CMSC 427
Computer Graphics[1]

David M. Mount
Department of Computer Science
University of Maryland
Fall 2013

Lecture 1: Introduction to Computer Graphics

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

Computer graphics has grown tremedously over the past 20–30 years with the advent of inexpensive interactive display technology. The availability of high resolution, highly dynamic, colored displays has enabled computer graphics to serve a role in *intelligence amplification*, where a human working in conjunction with a graphics-enabled computer can engage in creative activities that would be difficult or impossible without this enabling technology. An important aspect of this interaction is that vision is the sensory mode of highest bandwidth.

Because of the importance of vision and visual communication, computer graphics has found applications in numerous areas of science, engineering, and entertainment. These include:

Computer-Aided Design: The design of 3-dimensional manufactured objects such as appliances, planes, automobiles, and the parts that they are made of.

Drug Design: The design and analysis drugs based on their geometric interactions with molecules such as proteins and enzymes.

Architecture: Designing buildings by computer with the capability to perform virtual "fly throughs" of the structure and investigation of lighting properties.

Medical Imaging: Visualizations of the human body produced by 3-dimensional scanning technology.

Computational Simulations: Visualizations of physical simulations, such as airflow analysis in computational fluid dynamics or stresses on bridges.

Entertainment: Film production and computer games.

Interaction versus Realism: One of the most important tradeoffs faced in the design of interactive computer graphics systems is the balance between the speed of interactivity and degree of visual realism. To provide a feeling of interaction, images should be rendered at speeds of at least 20–30 frames (images) per second. However, producing a high degree of realism at these speeds for very complex scenes is difficult. This difficulty arises from a number of sources:

Large Geometric Models: Large-scale models, such as factories, city-scapes, forests and jungles, and crowds of people, can involve vast numbers of geometric elements.

Complex Geometry: Many natural objects (such as hair, fur, trees, plants, water, and clouds) have very sophisticate geometric structure, and they move and interact in complex manners.

Complex Illumination: Many natural objects (such as human hair and skin, plants, and water) reflect light in complex and subtle ways.

The Scope of Computer Graphics: Graphics is both fun and challenging. The challenge arises from the fact that computer graphics draws from so many different areas, including:

Mathematics and Geometry: Modeling geometric objects. Representing and manipulating surfaces and shapes. Describing 3-dimensional transformations such as translation and rotation.

Physics (Kinetics): Understanding how physical objects behave when acted upon by various forces.

Physics (Illumination): Understanding how physical objects reflect light.

Computer Science: The design of efficient algorithms and data structures for rendering.

Software Engineering: Software design and organization for large and complex systems, such as computer games.

High-Performance Computing: Interactive systems, like computer games, place high demands on the efficiency of processing, which relies on parallel programming techniques. It is necessary to understand how graphics processors work in order to produce the most efficient computation times.

The Scope of this Course: There has been a great deal of software produced to aid in the generation of large-scale software systems for computer graphics. Our focus in this course will *not* be on how to use these systems to produce these images. (If you are interested in this topic, you should take courses in the art technology department). As in other computer science courses, our interest is not in how to use these tools, but rather in understanding how these systems are constructed and how they work.

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

In this course we will deal primarily with the task of producing a both single images and animations from a 2- or 3-dimensional scene models. Over the course of the semester, we will build from a simple basis (e.g., drawing a triangle in 3-dimensional space) all the way to complex methods, such as lighting models, texture mapping, motion blur, morphing and blending, anti-aliasing.

Let us begin by considering the process of drawing (or *rendering*) a single image of a 3-dimensional scene. This is crudely illustrated in the figure below. The process begins by producing a mathematical model of the object to be rendered. Such a model should describe not only the shape of the object but its color, its surface finish (shiny, matte, transparent,

fuzzy, scaly, rocky). Producing realistic models is extremely complex, but luckily it is not our main concern. We will leave this to the artists and modelers. The scene model should also include information about the location and characteristics of the light sources (their color, brightness), and the atmospheric nature of the medium through which the light travels (is it foggy or clear). In addition we will need to know the location of the viewer. We can think of the viewer as holding a "synthetic camera", through which the image is to be photographed. We need to know the characteristics of this camera (its focal length, for example).

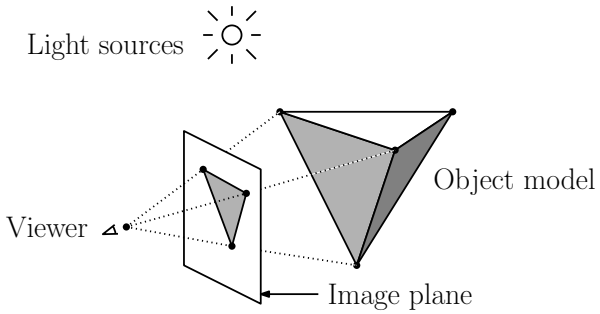


Fig. 1: A typical rendering situation.

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

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

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

Surface Detail: Are the surfaces textured, either with color (as in a wood-grain pattern) or surface irregularities (such as bumpiness).

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

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

By the end of the semester, you should have a basic understanding of how each of the steps is performed. Of course, a detailed understanding of most of the elements that are important to computer graphics will beyond the scope of this one-semester course. But by combining what you have learned here with other resources (from books or the Web) you will know enough to, say, write a simple video game, write a program to generate highly realistic images, or produce a simple animation.

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

Basics:

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

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

Implementation Issues: Rasterization, clipping.

Modeling:

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

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

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

Projection:

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

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

Issues in Realism:

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

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

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

Although this order represents a "reasonable" way in which to present the material. We will present the topics in a different order, mostly to suit our need to get material covered before major programming assignments.

Lecture 2: Basics of Graphics Systems and Architectures

Elements of 2-dimensional Graphics: Computer graphics is all about rendering images (either realistic or stylistic) by computer. The process of producing such images involves a number of elements. The most basic of these is the generation of the simplest two-dimensional elements, from which complex scenes are then constructed. Let us begin our exploration of computer graphics by discussing these simple two-dimensional primitives. Examples of the primitive drawing elements include *line segments*, *polylines*, *curves*, *filled regions*, and *text*.

Polylines: A polyline (or more properly a *polygonal curve*) is a finite sequence of line segments joined end to end. These line segments are called *edges*, and the endpoints of the line segments are called *vertices*. A single line segment is a special case. A polyline is *closed* if it ends where it starts (see Fig. 2). It is *simple* if it does not self-intersect. Self-intersections include such things as two edge crossing one another, a vertex intersecting in the interior of an edge, or more than two edges sharing a common vertex. A simple, closed polyline is also called a *simple polygon*.

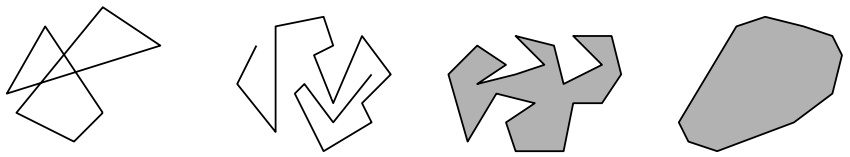


Fig. 2: Polylines.

If a polygon is simple and all its internal angle are at most 180°, then it is a *convex polygon*. Convex polygons are important structures, because they have particularly nice mathematical properties. For example, the intersection of any two convex polygons (if not empty) is a convex polygon.

The geometry of a polyline in the plane can be represented simply as a sequence of the (x, y) coordinates of its vertices. The way in which the polyline is rendered is determined by a set of properties called *graphical attributes*. These include elements such as *color*, *line width*, and *line style* (solid, dotted, dashed). Polyline attributes also include how consecutive segments are joined. For example, when two line segments come together at a sharp angle, do we round the corner between them, square it off, or leaving it pointed?

Curves: Curves consist of various common shapes, such as circles, ellipses, circular arcs. It also includes special free-form curves. Later in the semester we will discuss other types of curves, such as *Bézier curves*, *B-splines*, and their variants. These are curves that are defined by a collection of *control points*, which a designer can manipulate to affect the curve's shape.

Filled regions: Any simple, closed polyline in the plane defines a region consisting of an inside and outside. (This is a typical example of an utterly obvious fact from topology that is notoriously hard to prove. It is called the *Jordan curve theorem*.) We can fill any such region with a color or repeating pattern. In some cases it is desired to draw both the bounding polyline and the filled region, and in other cases just the filled region is to be drawn (see Fig. 3).

If a closed polyline is not simple, it is still possible to define the notions of "inside" and "outside," but the definitions are not obvious. There are a couple of ways of doing this. One way is to shoot a ray from a given point p to infinity and count the number of intersections with the curve's boundary. If this number is odd, the point is defined to be inside, and otherwise it is outside. The other common method is based on the *winding number*. Here is the idea. Walk around the curve's boundary and measure the total angle swept around p. Here angle is a signed quantity, and clockwise swept angles cancel out counterclockwise swept angles. This is called the winding number with respect to p. If the point p lies outside the object, the clockwise and counterclockwise contributions will cancel out, and the winding number will be zero. If p lies inside the object, the count will be 2π, assuming that the polyline is simple. In general, if p lies inside the winding number will be some nonzero multiple of 2π. Thus, by computing the winding number, we can determine whether p is inside the object.

Text: Although we do not normally think of text as a graphical output, it occurs frequently within graphical images such as engineering diagrams. Text can be thought of as a

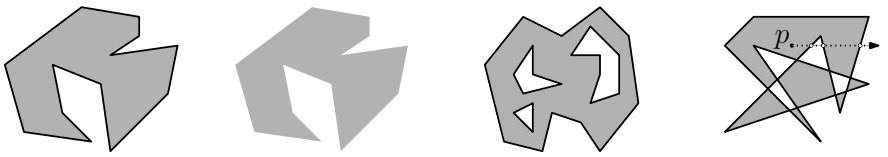


Fig. 3: Filled regions defined by closed polylines.

sequence of characters in some *font*. As with polylines there are numerous attributes which affect how the text appears. This includes the font's *face* (Times-Roman, Helvetica, Arial, Courier, for example), its *weight* (normal, bold, light), its *style* or *slant* (normal, slanted, italic, for example), its *size* (which is usually measured in *points*, a printer's unit of measure equal to 1/72-inch), and its *color* (see Fig. 4).

Face (family)	Weight	Style (shape)	Size
Times Roman	Normal	Normal	6 point
Helvetica	**Bold**	*Slanted*	8 point
`Courier`		*Italic*	10 point

Fig. 4: Text font properties. (Font faces and sizes are approximate.)

Text generation, or more accurately, *typography*, is a remarkably rich and complex topic (which may explain why even relatively expensive printers have difficulty printing documents reliably and consistently). Some of the differences are due to the fact that different written languages have very different character sets and punctuation symbols, are oriented differently on the page (left-to-right or right-to-left or top-to-bottom), and different rules for spacing, accents and other markings, and so on.

An interesting example of this in many fonts is that certain sequences of letters are often combined, for the sake of aesthetics, into a single character, called a *ligature*. In English, the common ligatures include "fi", "ff", "fl", "ffi", and "ffl" (see Fig. 5).

flying fish → flying fish

Fig. 5: Example of the "fl" and "fi" ligatures in the Times-Roman font.

Mathematical typesetting also poses many challenges. Consider for example the difference in spacing between "−" and "b" in the expressions "$a - b$" and "$-b$".

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

An important characteristic of pixel maps is the number of bits per pixel, called its *depth*. The simplest example is an image made up of black and white pixels (depth 1), each represented by a single bit (e.g., 0 for black and 1 for white). This is called a *bitmap*. Typical gray-scale (or *monochrome*) images can be represented as a pixel map

of depth 8, in which each pixel is represented by assigning it a numerical value over the range 0 to 255. More commonly, full color is represented using a pixel map of depth 24, where 8 bits each are used to represented the components of red, green and blue. We will frequently use the term *RGB* when referring to this representation.

Interactive 3-dimensional Graphics: Anyone who has played a computer game is accustomed to interaction with a graphics system in which the principal mode of rendering involves 3-dimensional scenes. Producing highly realistic, complex scenes at interactive frame rates (30–60 frames per second, say) is made possible with the aid of a hardware device called a *graphics processing unit*, or *GPU* for short. GPUs are very complex things, and we will only be able to provide a general outline of how they work.

Like the CPU (central processing unit), the GPU is a critical part of modern computer systems. It has its own memory, separate from the CPU's memory, in which it stores the various graphics objects (e.g., object coordinates and texture images) that it needs in order to do its job. Part of this memory is called the *frame buffer*, which is a dedicated chunk of memory where the pixels associated with your monitor are stored. Another entity, called the *video controller*, reads the contents of the frame buffer and generates the actual image on the monitor. This process is illustrated in schematic form in Fig. 6.

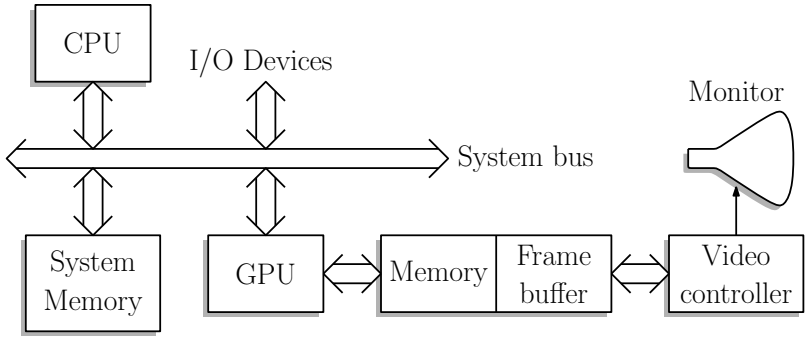


Fig. 6: Architecture of a simple GPU-based graphics system.

Traditionally, GPUs are designed to perform a relatively limited fixed set of operations, but with blazing speed and a high degree of parallelism. Modern GPUs are much *programmable*, in that they provide the user the ability to program various elements of the graphics process. For example, modern GPUs support programs called *vertex shaders* and *fragment shaders*, which provide the user with the ability to fine-tune the colors assigned to vertices and fragments. Recently there has been a trend towards what are called *general purpose GPUs*, which can perform not just graphics rendering, but general scientific calculations on the GPU. Since we are interested in graphics here, we will focus on the GPUs traditional role in the rendering process.

The Graphics Pipeline: A key concept behind all GPUs is the notion of the *graphics pipeline*. This is conceptual tool, where your user program sits at one end sending graphics commands to the GPU, and the frame buffer sits at the other end. A typical command from your program might be "draw a triangle in 3-dimensional space at these coordinates." The job of

the graphics system is to convert this simple request to that of coloring a set of pixels on your display. The process of doing this is quite complex, and involves a number of stages. Each of these stages is performed by some part of the pipeline, and the results are then fed to the next stage of the pipeline, until the final image is produced at the end.

Broadly speaking the pipeline can be viewed as involving four major stages. (This is mostly a conceptual aid, since the GPU architecture is not divided so cleanly.) The process is illustrated in Fig. 7.

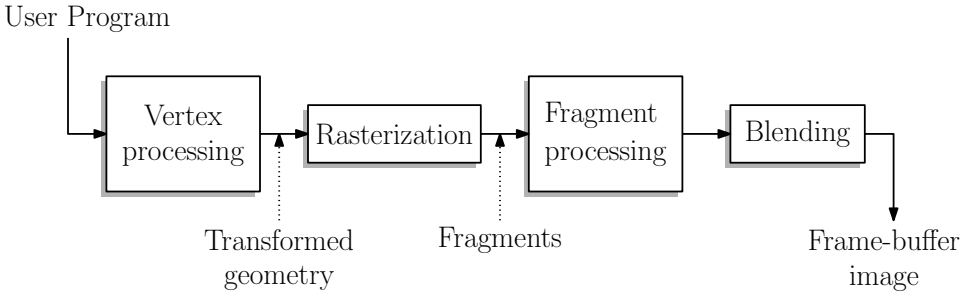


Fig. 7: Stages of the graphics pipeline.

Vertex Processing: Geometric objects are introduced to the pipeline from your program. Objects are described in terms of vectors in 3-dimensional space (for example, a triangle might be represented by three such vectors, one per vertex). Surfaces are represented as a *polygonal mesh*, in which a collection of polygons are glued together at a set of *vertices* (see Fig. 8). In the vertex processing stage, the graphics system *transforms* these coordinates into a coordinate system that is more convenient to the graphics system. For the purposes of this high-level overview, you might imagine that the transformation projects the vertices of the three-dimensional triangle onto the 2-dimensional coordinate system of your screen, called *screen space*.

In order to know how to perform this transformation, your program sends a command to the GPU specifying the location of the camera and its projection properties. The output of this stage is called the *transformed geometry*.

This stage involves other tasks as well. For one, *clipping* is performed to snip off any parts of your geometry that lie outside the viewing area of the window on your display. Another operation is *lighting*, where computations are performed to determine the colors and intensities of the vertices of your objects. (How the lighting is performed depends on commands that you send to the GPU, indicating where the light sources are and how bright they are.)

Rasterization/assembly: The job of the rasterizer is to convert the geometric shape given in terms of its screen coordinates into individual pixels, called *fragments* (see Fig. 8).

Fragment Processing: Each fragment is then run through various computations. First, it must be determined whether this fragment is *visible*, or whether it is hidden behind some other fragment. If it is visible, it will then be subjected to coloring. This may involve applying various coloring textures to the fragment and/or color blending from the vertices, in order to produce the effect of smooth shading.

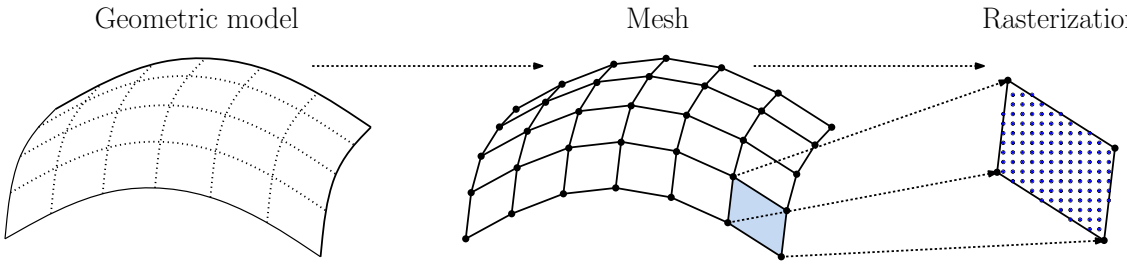


Fig. 8: The rendering process, from surface geometry, to a polygonal mesh, to a set of fragments.

Blending: Generally, there may be a number of fragments that affect the color of a given pixel. (This typically results from translucence or other special effects like motion blur.) The colors of these fragments are then blended together to produce the final pixel color. The final output of this stage is the *frame-buffer image*.

When GPUs were first invented, the graphics pipeline had a fairly rigid structure, which is sometimes referred to as the *fixed functionality pipeline*. Early GPUs allowed users to modify various parameters, which affected how rendering stapges, such as lighting and texturing, are to be applied, but the user did not have direct control of the underlying computation. Over time, GPUs were capable of being programmed, which meant that, rather than using the standard fixed functions, users could invent their own lighting and texturing algorithms and load these programs directly into the GPU for execution. These programs are called *shaders*. This semester, we will begin by presenting the standard fixed functionality pipeline (for the purpose of establishing an easily accessible set of basic operations), but we will also discuss the design of shaders as well.

Graphics Libraries: Let us consider programming a 3-dimensional interactive graphics system, as described above. The challenge is that your program needs to specify, at the rate of over 30 frames per second, what image is to be drawn. We call each such redrawing a *display cycle* or a *refresh cycle*, since your program is refresh the current contents of the image. Your program communicates with the graphics system through a library, or more formally, an *application programmer's interface* or API. There are a number of different APIs used in modern graphics systems, each providing some features relative to the others. Broadly speaking, graphics APIs are classified into two general classes:

Retained Mode: The library maintains the state of the computation in its own internal data structures. With each refresh cycle, this data is transmitted to the GPU for rendering.

- Because it knows the full state of the scene, the library can perform global optimizations automatically.
- This method is less well suited to time-varying data sets, since the internal representation of the data set needs to be updated frequently.
- This is functionally analogous to program compilation.
- Examples: Java3d, Ogre, Open Scenegraph.

Immediate Mode: The application provides all the primitives with each display cycle. In other words, your program transmits commands directly to the GPU for execution.

- The library can only perform local optimizations, since it does not know the global state. It is the responsibility of the user program to perform global optimizations.
- This is well suited to highly dynamic scenes.
- This is functionally analogous to program interpretation.
- Examples: OpenGL, DirectX.

OpenGL: OpenGL is a widely used industry standard graphics API. It has been ported to virtually all major systems, and can be accessed from a number of different programming languages (C, C++, Java, Python, ...). Because it works across many different platforms, it is very general. (This is in contrast to DirectX, which has been desired to work primarily on Microsoft systems.)

For the most part, OpenGL operates in *immediate mode*, which means that each function call results in a command being sent directly to the GPU. There are some retained elements, however. For example, transformations, lighting, and texturing need to be set up, so that they can be applied later in the computation.

Because of the design goal of being independent of the window system and operating system, OpenGL does *not* provide capabilities for windowing tasks or user input and output. For example, there are no commands in OpenGL to create a window, to resize a window, to determine the current mouse coordinates, or to detect whether a keyboard key has been hit. Everything is focused just on the process of generating an image. In order to achieve these other goals, it is necessary to use an additional toolkit. There are a number of different toolkits, which provide various capabilities. We will cover a very simple one in this class, called *GLUT*, which stands for the *GL Utility Toolkit*. GLUT has the virtue of being very simple, but it does not have a lot of features. To get these features, you will need to use a more sophisticated toolkit.

There are many, many tasks needed in a typical large graphics system. As a result, there are a number of software systems available that provide utility functions. For example, suppose that you want to draw a sphere. OpenGL does not have a command for drawing spheres, but it can draw triangles. What you would like is a utility function which, given the center and radius of a sphere, will produce a collection of triangles that approximate the sphere's shape. OpenGL provides a simple collection of utilities, called the *GL Utility Library* or *GLU* for short.

Since we will be discussing a number of the library functions for OpenGL, GLU, and GLUT during the next few lectures, let me mention that it is possible to determine which library a function comes from by its prefix. Functions from the OpenGL library begin with "gl" (as in "glTriangle"), functions from GLU begin with "glu" (as in "gluLookAt"), and functions from GLUT begin with "glut" (as in "glutCreateWindow").

We have described some of the basic elements of graphics systems. Next time, we will discuss OpenGL in greater detail.

Lecture 3: Basic Elements of OpenGL and GLUT

The OpenGL API: Before getting to the topic of how graphics are generated, let us begin with a discussion of the graphics API that we will be using this semester, OpenGL. We will also discuss two related libraries, GLU (the OpenGL utility library) and GLUT (the OpenGL Utility Toolkit). OpenGL is designed to be a machine-independent graphics library, but one that can take advantage of the structure of typical hardware accelerators for computer graphics.

The Main Program: Before discussing how to draw shapes, we will begin with the basic elements of how to create a window. OpenGL was intentionally designed to be independent of any specific window system. Consequently, a number of the basic window operations are not provided. For this reason, a separate library, called *GLUT* or *OpenGL Utility Toolkit*, was created to provide these functions. It is the GLUT toolkit which provides the necessary tools for requesting that windows be created and providing interaction with I/O devices.

Let us begin by considering a typical main program. Throughout, we will assume that programming is done in C++, but most our examples will compile in C as well. (Do not worry for now if you do not understand the meanings of the various calls. Later we will discuss the various elements in more detail.) This program creates a window that is 400 pixels wide and 300 pixels high, located in the upper left corner of the display.

Typical OpenGL/GLUT Main Program

```
#include <GL/glut.h>                          // GLUT, GLU, and OpenGL defs
int main(int argc, char** argv)               // program arguments
{
    glutInit(&argc, argv);                    // initialize glut and gl
                                              // double buffering and RGB
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA);
    glutInitWindowSize(400, 300);             // initial window size
    glutInitWindowPosition(0, 0);             // initial window position
    glutCreateWindow(argv[0]);                // create window

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

    myInit();                                 // your own initializations
    glutMainLoop();                           // turn control over to glut
    return 0; // we never return here; this just keeps the compiler happy
}
```

The call to glutMainLoop turns control over to the system. After this, the only return to your program will occur due to various callbacks. (The final "return 0" is only there to keep the compiler from issuing a warning.) Here is an explanation of the first five function calls.

glutInit: The arguments given to the main program (argc and argv) are the command-line arguments supplied to the program. This assumes a typical Unix environment, in which the program is invoked from a command line. We pass these into the main initialization procedure, glutInit. This procedure must be called before any others. It processes (and removes) command-line arguments that may be of interest to GLUT and the window system and does general

initialization of GLUT and OpenGL. Any remaining arguments are then left for the user's program to interpret, if desired.

glutInitDisplayMode: The next procedure, glutInitDisplayMode, performs initializations by informing OpenGL how to set up its various buffers. Recall that the frame buffer is a special 2-dimensional array in memory where the graphical image is stored. OpenGL maintains an enhanced version of the frame buffer with additional information. For example, this includes depth information for hidden surface removal. The system needs to know how we are representing colors of our general needs in order to determine the depth (number of bits) to assign for each pixel in the frame buffer. The argument to glutInitDisplayMode is a logical-or (using the operator "|") of a number of possible options. A partial list of possible arguments is given in Table 1.

Display Mode	Meaning
GLUT_RGB	Use RGB colors
GLUT_RGBA	Use RGB plus α (recommended)
GLUT_INDEX	Use colormapped colors (not recommended)
GLUT_DOUBLE	Use double buffering (recommended)
GLUT_SINGLE	Use single buffering (not recommended)
GLUT_DEPTH	Use depth buffer (needed for hidden surface removal)

Table 1: Partial list of arguments to glutInitDisplayMode. (Constants defined in glut.h. Other arguments will be discussed in later lectures.)

.

Color: First off, we need to tell the system how colors will be represented. There are three methods, of which two are fairly commonly used: GLUT_RGB or GLUT_RGBA. The first uses standard RGB colors (24-bit color, consisting of 8 bits of red, green, and blue), and is the default. The second requests RGBA coloring. In this color system there is a fourth component (A or α), which indicates the opaqueness of the color (1 = fully opaque, 0 = fully transparent). This is useful in creating transparent effects. We will discuss how this is applied later this semester. It turns out that there is no advantage in trying to save space using GLUT_RGB over GLUT_RGBA, since (according to the documentation), both are treated the same.

Single or Double Buffering: The next option specifies whether single or double buffering is to be used, GLUT_SINGLE or GLUT_DOUBLE, respectively. To explain the difference, we need to understand a bit more about how the frame buffer works. In raster graphics systems, whatever is written to the frame buffer is immediately transferred to the display. This process is repeated frequently, say 30–60 times a second. To do this, the typical approach is to first erase the old contents by setting all the pixels to some background color, say black. After this, the new contents are drawn. However, even though it might happen very fast, the process of setting the image to black and then redrawing everything produces a noticeable flicker in the image.

Double buffering is a method to eliminate this flicker. In double buffering, the system maintains two separate frame buffers. The *front buffer* is the one which is displayed,

and the *back buffer* is the other one. Drawing is always done to the back buffer. Then to update the image, the system simply swaps the two buffers. The swapping process is very fast, and appears to happen instantaneously (with no flicker). Double buffering requires twice the buffer space as single buffering, but since memory is relatively cheap these days, it is the preferred method for interactive graphics.

It is possible go even further. Triple buffering is sometimes used in very high performance graphics systems. Once one buffer is drawn, it is possible for the program to start drawing the next, without waiting for the next refresh cycle to start. Quadruple buffering is used to achieve the benefits of double buffering in stereoscopic systems.

Depth Buffer: One other option that we will need later with 3-dimensional graphics will be hidden surface removal. Virtually all raster-based interactive graphics systems perform hidden surface removal by an approach called the *depth-buffer* or *z-buffer*. In such a system, each fragment stores its distance from the eye. When fragments are rendered as pixels only the closest is actually drawn. The depth buffer is enabled with the option GLUT_DEPTH. For this program it is not needed, and so has been omitted. When there is no depth buffer, the last pixel to be drawn is the one that you see.

glutInitWindowSize: This command specifies the desired width and height of the graphics window. The general form is glutInitWindowSize(int width, int height). The values are given in numbers of pixels.

glutInitPosition: This command specifies the location of the upper left corner of the graphics window. The form is glutInitWindowPosition(int x, int y) where the (x, y) coordinates are given relative to the upper left corner of the display. Thus, the arguments $(0, 0)$ places the window in the upper left corner of the display.

Note that glutInitWindowSize and glutInitWindowPosition are both considered to be only *suggestions* to the system as to how to where to place the graphics window. Depending on the window system's policies, and the size of the display, it may not honor these requests.

glutCreateWindow: This command actually creates the graphics window. The general form of the command is glutCreateWindow(char* title), where title is a character string. Each window has a title, and the argument is a string which specifies the window's title. We pass in argv[0]. In Unix argv[0] is the name of the program (the executable file name) so our graphics window's name is the same as the name of our program.

Beware: Note that the call glutCreateWindow does not really create the window, but rather merely sends a request to the system that the window be created. Why do you care? In some OpenGL implementations, certain operations cannot be performed unless the window (and the associated graphics context) exists. Such operations should be performed only after notification has been received that the window really exists. Your program is informed of this event through an event callback, either *reshape* or *display*. We will discuss them below.

The general structure of an OpenGL program using GLUT is shown in Fig. 9. (Don't worry if some elements are unfamiliar. We will discuss them below.)

Event-driven Programming and Callbacks: Virtually all interactive graphics programs are *event driven*. Unlike traditional programs that read from a standard input file, a graphics

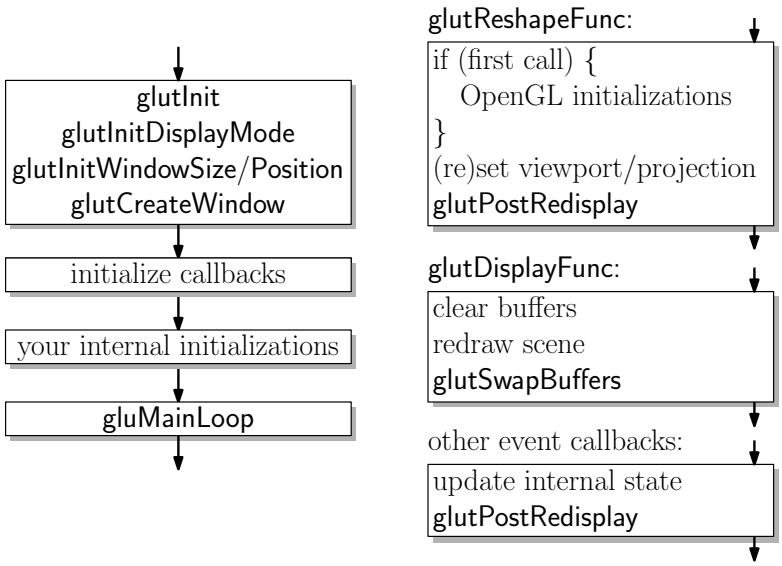


Fig. 9: General structure of an OpenGL program using GLUT.

program must be prepared at any time for input from any number of sources, including the mouse, or keyboard, or other graphics devises such as trackballs and joysticks.

In OpenGL this is done through the use of *callbacks*. The graphics program instructs the system to invoke a particular procedure whenever an event of interest occurs, say, the mouse button is clicked. The graphics program indicates its interest, or *registers*, for various events. This involves telling the window system which event type you are interested in, and passing it the name of a procedure you have written to handle the event.

Note: If you program in C++, note that the Glut callback functions you define must be "standard" procedures; they cannot be class member functions.

Types of Callbacks: Callbacks are used for two purposes, *user input events* and *system events*. User input events include things such as mouse clicks, the motion of the mouse (without clicking) also called *passive motion*, keyboard hits. Note that your program is only signaled about events that happen to your window. For example, entering text into another window's dialogue box will not generate a keyboard event for your program.

There are a number of different events that are generated by the system. There is one such special event that every OpenGL program must handle, called a *display event*. A display event is invoked when the system senses that the contents of the window need to be redisplayed, either because:

- the graphics window has completed its initial creation,
- an obscuring window has moved away, thus revealing all or part of the graphics window,
- the program explicitly requests redrawing, for example, because the internal state has changed in a way that affects the scene, by calling glutPostRedisplay.

Recall from above that the command glutCreateWindow does not actually create the window, but merely requests that creation be started. In order to inform your program that the

creation has completed, the system generates a display event. This is how you know that you can now start drawing into the graphics window.

Another type of system event is a *reshape event*. This happens whenever the window's size is altered. The callback provides information on the new size of the window. Recall that your initial call to glutInitWindowSize is only taken as a suggestion of the actual window size. When the system determines the actual size of your window, it generates such a callback to inform you of this size. Typically, the first two events that the system will generate for any newly created window are a reshape event (indicating the size of the new window) followed immediately by a display event (indicating that it is now safe to draw graphics in the window).

Often in an interactive graphics program, the user may not be providing any input at all, but it may still be necessary to update the image. For example, in a flight simulator the plane keeps moving forward, even without user input. To do this, the program goes to sleep and requests that it be awakened in order to draw the next image. There are two ways to do this, a *timer event* and an *idle event*. An idle event is generated every time the system has nothing better to do. This is often fine, since it means that your program wastes no cycles.

Often, you want to have more precise control of timing (e.g., when trying to manage parallel threads such as artificial intelligence and physics modeling). If so, an alternate approach is to request a timer event. In a timer event you request that your program go to sleep for some period of time and that it be "awakened" by an event some time later, say 1/50 of a second later. In glutTimerFunc the first argument gives the sleep time as an integer in milliseconds and the last argument is an integer identifier, which is passed into the callback function. Various input and system events and their associated callback function prototypes are given in Table 2.

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

Table 2: Common callbacks and the associated registration functions.

For example, the following code fragment shows how to register for the following events: display events, reshape events, mouse clicks, keyboard strikes, and timer events. The functions like myDraw and myReshape are supplied by the user, and will be described later.

Most of these callback registrations simply pass the name of the desired user function to be called for the corresponding event. The one exception is glutTimeFunc whose arguments are the number of milliseconds to wait (an unsigned int), the user's callback function, and an integer identifier. The identifier is useful if there are multiple timer callbacks requested (for different times in the future), so the user can determine which one caused this particular

Typical Callback Setup

```
int main(int argc, char** argv)
{
    ...
    glutDisplayFunc(myDraw);                // set up the callbacks
    glutReshapeFunc(myReshape);
    glutMouseFunc(myMouse);
    glutKeyboardFunc(myKeyboard);
    glutTimerFunc(20, myTimeOut, 0);        // timer in 20/1000 seconds
    ...
}
```

event.

Callback Functions: What does a typical callback function do? This depends entirely on the application that you are designing. Some examples of general form of callback functions is shown below.

Examples of Callback Functions for System Events

```
void myDraw() {                         // called to display window
   // ...insert your drawing code here ...
}
void myReshape(int w, int h) {          // called if reshaped
    windowWidth = w;                    // save new window size
    windowHeight = h;
    // ...may need to update the projection ...
    glutPostRedisplay();                // request window redisplay
}
void myTimeOut(int id) {                // called if timer event
    // ...advance the state of animation incrementally...
    glutPostRedisplay();                // request redisplay
    glutTimerFunc(20, myTimeOut, 0);    // schedule next timer event
}
```

Note that the timer callback and the reshape callback both invoke the function glutPostRedisplay. This procedure informs OpenGL that the state of the scene has changed and should be redrawn (by calling your drawing procedure). This might be requested in other callbacks as well.

Note that each callback function is provided with information associated with the event. For example, a reshape event callback passes in the new window width and height. A mouse click callback passes in four arguments, which button was hit (b: left, middle, right), what the buttons new state is (s: up or down), the (x, y) coordinates of the mouse when it was clicked (in pixels). The various parameters used for b and s are described in Table 3. A keyboard event callback passes in the character that was hit and the current coordinates of the mouse. The timer event callback passes in the integer identifier, of the timer event which caused the callback. Note that each call to glutTimerFunc creates only one request for a timer event. (That is, you do not get automatic repetition of timer events.) If you want to generate

```
                                        // called if mouse click
void myMouse(int b, int s, int x, int y) {
    switch (b) {                        // b indicates the button
        case GLUT_LEFT_BUTTON:
            if (s == GLUT_DOWN)         // button pressed
                // ...
            else if (s == GLUT_UP)      // button released
                // ...
            break;
        // ...                          // other button events
    }
}
                                        // called if keyboard key hit
void myKeyboard(unsigned char c, int x, int y) {
   switch (c) {                         // c is the key that is hit
      case 'q':                         // 'q' means quit
          exit(0);
          break;
      // ...                            // other keyboard events
   }
}
```

events on a regular basis, then insert a call to glutTimerFunc from within the callback function to generate the next one.

GLUT Parameter Name	Meaning
GLUT_LEFT_BUTTON	left mouse button
GLUT_MIDDLE_BUTTON	middle mouse button
GLUT_RIGHT_BUTTON	right mouse button
GLUT_DOWN	mouse button pressed down
GLUT_UP	mouse button released

Table 3: GLUT parameter names associated with mouse events. (Constants defined in glut.h)

Lecture 4: More about OpenGL and GLUT

Basic Drawing: In the previous lecture, we showed how to create a window in GLUT, how to get user input, but we have not discussed how to get graphics to appear in the window. Here, we begin discussion of how to use OpenGL to draw objects.

Before being able to draw a scene, OpenGL needs to know the following information: what are the *objects* to be drawn, how is the image to be *projected* onto the window, and how *lighting* and *shading* are to be performed. To begin with, we will consider a very the simple case. There are only 2-dimensional objects, no lighting or shading. Also we will consider only relatively little user interaction.

Because we generally do not have complete control over the window size, it is a good idea to think in terms of drawing on a rectangular *idealized drawing region*, whose size and shape are completely under our control. Then we will scale this region to fit within the actual graphics window on the display. There are many reasons for doing this. For example, if you design your program to work specifically for 400×400 window, but then the user resizes the window, what do you do? It is a much better idea to assume a window of arbitrary size. For example, you could define two variables w and h, that indicate the width and height of the window, respectively. The values w and h could be measured in whatever units are most natural to your application: pixels, inches, light years, microns, furlongs or fathoms.

OpenGL allows for the grahics window to be broken up into smaller rectangular subwindows, called *viewports*. We will then have OpenGL scale the image drawn in the idealized drawing region to fit within the viewport. The main advantage of this approach is that it is very easy to deal with changes in the window size.

We will consider a simple drawing routine for the picture shown in the figure. We assume that our idealized drawing region is a unit square over the real interval $[0, 1] \times [0, 1]$. (Throughout the course we will use the notation $[a, b]$ to denote the interval of real values z such that $a \leq z \leq b$. Hence, $[0, 1] \times [0, 1]$ is a unit square whose lower left corner is the origin.) This is illustrated in Fig. 10.

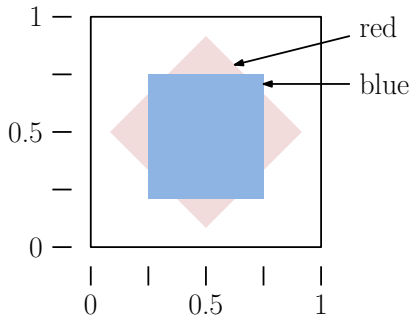


Fig. 10: Drawing produced by the simple display function.

GLUT uses the convention that the origin is in the upper left corner and coordinates are given as integers. This makes sense for GLUT, because its principal job is to communicate with the window system, and most window systems use this convention. On the other hand, OpenGL uses the convention that coordinates are (generally) floating point values and the origin is in the lower left corner. Recalling the OpenGL goal is to provide us with an idealized drawing surface, this convention is mathematically more elegant.

The Display Callback: Recall that the *display callback function* is the function that is called whenever it is necessary to redraw the image, which arises for example:

- The initial creation of the window,
- Whenever the window is uncovered by the removal of some overlapping window (assuming that your window system does not automatically save the contents of covered windows),
- Whenever your program requests that it be redrawn (through the use of glutPostRedisplay() function, as in the case of an animation, where this would happen continuously.

The display callback function for our program is shown below. We first erase the contents of the image window, then do our drawing, and finally swap buffers so that what we have drawn becomes visible. (Recall double buffering from the previous lecture.) This function first draws a red diamond and then (on top of this) it draws a blue rectangle. Let us assume double buffering is being performed, and so the last thing to do is invoke glutSwapBuffers() to make everything visible.

Let us present the code, and we will discuss the various elements of the solution in greater detail below.

Sample Display Function

```
void myDisplay() {                              // display function
    glClear(GL_COLOR_BUFFER_BIT);               // clear the window

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

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

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

Clearing the Window: The command glClear() clears the window, by overwriting it with the background color. The background color is black by default, but generally it may be set by the call:

glClearColor(GLfloat Red, GLfloat Green, GLfloat Blue, GLfloat Alpha).

The type GLfloat is OpenGL's redefinition of the standard float. To be correct, you should use the approved OpenGL types (e.g. GLfloat, GLdouble, GLint) rather than the obvious counterparts (float, double, and int). Typically the GL types are the same as the corresponding native types, but not always.

Colors components are given as floats in the range from 0 to 1, from dark to light. Recall that the A (or α) value is used to control transparency. For opaque colors A is set to 1. Thus to set the background color to black, we would use glClearColor(0.0, 0.0, 0.0, 1.0), and to set it to blue use glClearColor(0.0, 0.0, 1.0, 1.0). (**Tip:** When debugging your program, it is often a good idea to use an uncommon background color, like a random shade of pink, since black can arise as the result of many different bugs.) Since the background color is usually independent of drawing, the function glClearColor() is typically set in one of your initialization procedures, rather than in the drawing callback function.

Clearing the window involves resetting information within the drawing buffer. As we mentioned before, the drawing buffer may store different types of information. This includes color

information, of course, but depth or distance information is used for hidden surface removal. Typically when the window is cleared, we want to clear everything. (Occasionally it is useful to achieve certain special effects by clearing only some of the buffers.) The glClear() command allows the user to select which buffers are to be cleared. In this case we only have color in the depth buffer, which is selected by the option GL_COLOR_BUFFER_BIT. If we had a depth buffer to be cleared it as well we could do this by combining these using a "bitwise or" operation:

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)

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

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

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

But that is not all! The three argument versions assume RGB color. If we were using RGBA color instead, we would use glColor4d() variant instead. Here "4" signifies four arguments. (Recall that the A or alpha value is used for various effects, such an transparency. For standard (opaque) color we set $A = 1.0$.)

In some cases it is more convenient to store your colors in an array with three elements. The suffix "v" means that the argument is a vector. For example glColor3dv() expects a single argument, a vector containing three GLdouble's. (Note that this is a standard C/C++ style array, not the class vector from the C++ Standard Template Library.) Using C's convention that a vector is represented as a pointer to its first element, the corresponding argument type would be "const GLdouble*".

Whenever you look up the prototypes for OpenGL commands, you often see a long list with various suffixes to indicate the argument types. Here are some examples for glColor:

```
void glColor3d(GLdouble r, GLdouble g, GLdouble b)
void glColor3f(GLfloat r, GLfloat g, GLfloat b)
void glColor3i(GLint r, GLint g, GLint b)
... (and forms for byte, short, unsigned byte and unsigned short) ...

void glColor4d(GLdouble r, GLdouble g, GLdouble b, GLdouble a)
```

```
... (and 4-argument forms for all the other types) ...

void glColor3dv(const GLdouble *v)
... (and other 3- and 4-argument forms for all the other types) ...
```

Drawing commands: OpenGL supports drawing of a number of different types of objects. The simplest is glRectf(), which draws a filled rectangle. All the others are complex objects consisting of a (generally) unpredictable number of elements. This is handled in OpenGL by the constructs glBegin(*mode*) and glEnd(). Between these two commands a list of vertices is given, which defines the object. The sort of object to be defined is determined by the *mode* argument of the glBegin() command. Some of the possible modes are illustrated in Fig. 11. For details on the semantics of the drawing methods, see the reference manuals.

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

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

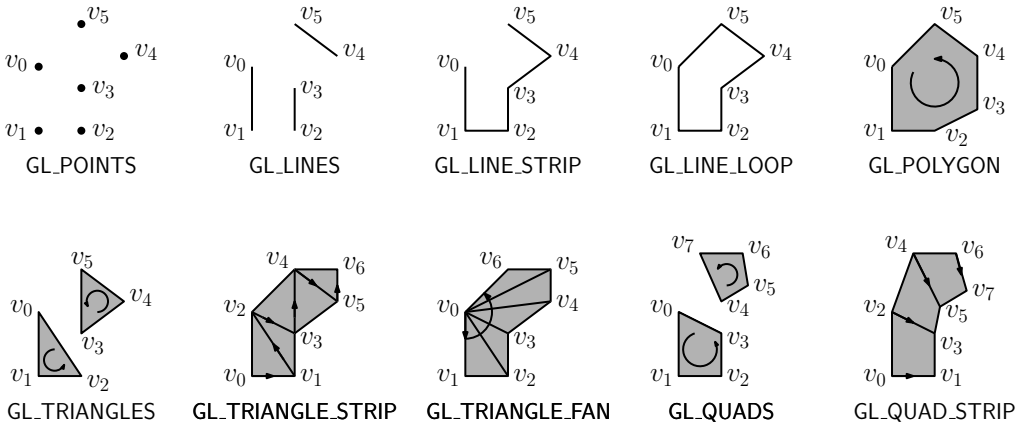


Fig. 11: Some OpenGL object definition modes. It is a good idea to draw primitives using a consistent direction, say counterclockwise.

In the example above we only defined the x- and y-coordinates of the vertices. How does OpenGL know whether our object is 2-dimensional or 3-dimensional? The answer is that it does not know. OpenGL represents all vertices as 3-dimensional coordinates internally. This may seem wasteful, but remember that OpenGL is designed primarily for 3-d graphics. If you do not specify the z-coordinate, then it simply sets the z-coordinate to 0.0. By the way, glRectf() always draws its rectangle on the $z = 0$ plane.

Between any glBegin()...glEnd() pair, there is a restricted set of OpenGL commands that may be given. This includes glVertex() and also other command attribute commands, such as glColor3f(). At first it may seem a bit strange that you can assign different colors to the

different vertices of an object, but this is a very useful feature. Depending on the shading model, it allows you to produce shapes whose color blends smoothly from one end to the other.

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

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

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

The size of the viewport depends on the dimensions of our window. Thus, every time the window is resized (and this includes when the window is created originally) we need to readjust the viewport to ensure proper transformation of the graphics. For example, in the typical case, where the graphics are drawn to the entire window, the reshape callback would contain the following call which resizes the viewport, whenever the window is resized.

Setting the Viewport in the Reshape Callback

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

The other thing that might typically go in the myReshape() function would be a call to glutPostRedisplay(), since you will need to redraw your image after the window changes size.

The general form of the command is

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

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

For example, suppose you had a $w \times h$ window, which you wanted to split in half by a vertical line to produce two different drawings. You could do the following.

```
glClear(GL_COLOR_BUFFER_BIT);       // clear the window
glViewport (0, 0, w/2, h);          // set viewport to left half
  // ...drawing commands for the left half of window
glViewport (w/2, 0, w/2, h);        // set viewport to right half
  // ...drawing commands for the right half of window
glutSwapBuffers();                  // swap buffers
```

Projection Transformation: In the simple drawing procedure, we said that we were assuming that the "idealized" drawing area was a unit square over the interval $[0, 1]$ with the origin in the lower left corner. The transformation that maps the idealized drawing region (in 2- or 3-dimensions) to the window is called the *projection*. We did this for convenience, since otherwise we would need to explicitly scale all of our coordinates whenever the user changes the size of the graphics window.

However, we need to inform OpenGL of where our "idealized" drawing area is so that OpenGL can map it to our viewport. This mapping is performed by a transformation matrix called the *projection matrix*, which OpenGL maintains internally. (In future lectures, we will discuss OpenGL's transformation mechanism in greater detail. In the mean time some of this may seem a bit arcane.)

Since matrices are often cumbersome to work with, OpenGL provides a number of relatively simple and natural ways of defining this matrix. For our 2-dimensional example, we will do this by simply informing OpenGL of the rectangular region of two dimensional space that makes up our idealized drawing region. This is handled by the command

gluOrtho2D(left, right, bottom, top).

First note that the prefix is "glu" and not "gl", because this procedure is provided by the GLU library. Also, note that the "2D" designator in this case stands for "2-dimensional." (In particular, it does not indicate the argument types, as with, say, glColor3f()).

All arguments are of type GLdouble. The arguments specify the x-coordinates (*left* and *right*) and the y-coordinates (*bottom* and *top*) of the rectangle into which we will be drawing. Any drawing that we do outside of this region will automatically be clipped away by OpenGL. The code to set the projection is given below.

Setting a Two-Dimensional Projection

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

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

Where does this code fragment go? It depends on whether the projection will change or not. If we make the simple assumption that are drawing will always be done relative to the $[0,1]^2$ unit square, then this code can go in some initialization procedure. If our program decides to change the drawing area (for example, growing the drawing area when the window is increased in size) then we would need to repeat the call whenever the projection changes.

At first viewports and projections may seem confusing. Remember that the viewport is a rectangle within the actual graphics window on your display, where you graphics will appear. The projection defined by gluOrtho2D() simply defines a rectangle in some "ideal" coordinate system, which you will use to specify the coordinates of your objects. It is the job of OpenGL to map everything that is drawn in your ideal window to the actual viewport on your screen. This is illustrated in Fig. 12.

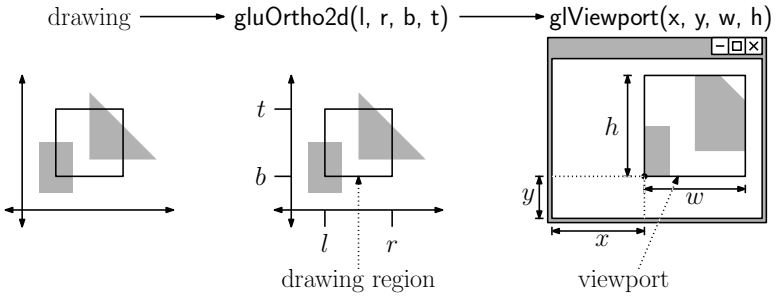


Fig. 12: Projection and viewport transformations.

The complete program is shown in Fig. 13.

Lecture 5: Geometry and Geometric Programming

Geometric Programming: We are going to leave our discussion of OpenGL for a while, and discuss some of the basic elements of geometry, which will be needed for the rest of the course. There are many areas of computer science that involve computation with geometric entities. This includes not only computer graphics, but also areas like computer-aided design, robotics, computer vision, and geographic information systems. In this and the next few lectures we will consider how this can be done, and how to do this in a reasonably clean and painless way.

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

Transformations: You are asked to render a twirling boomerang flying through the air. How would you represent the boomerang's rotation and translation over time in 3-dimensional space? How would you compute its exact position at a particular time?

Geometric Intersections: Given the same boomerang, how would you determine whether it has hit another object?

```
#include <cstdlib>                      // standard definitions
#include <iostream>                     // C++ I/O

#include <GL/glut.h>                    // GLUT (also loads gl.h and glu.h)

void myReshape(int w, int h) {                  // window is reshaped
    glViewport (0, 0, w, h);                    // update the viewport
    glMatrixMode(GL_PROJECTION);                // update projection
    glLoadIdentity();
    gluOrtho2D(0.0, 1.0, 0.0, 1.0);             // map unit square to viewport
    glMatrixMode(GL_MODELVIEW);
    glutPostRedisplay();                        // request redisplay
}

void myDisplay(void) {                          // (re)display callback
    glClearColor(0.5, 0.5, 0.5, 1.0);           // background is gray
    glClear(GL_COLOR_BUFFER_BIT);               // clear the window
    glColor3f(1.0, 0.0, 0.0);                   // set color to red
    glBegin(GL_POLYGON);                        // draw the diamond
        glVertex2f(0.90, 0.50);
        glVertex2f(0.50, 0.90);
        glVertex2f(0.10, 0.50);
        glVertex2f(0.50, 0.10);
    glEnd();
    glColor3f(0.0, 0.0, 1.0);                   // set color to blue
    glRectf(0.25, 0.25, 0.75, 0.75);            // draw the rectangle
    glutSwapBuffers();                          // swap buffers
}

int main(int argc, char** argv) {               // main program
    glutInit(&argc, argv);                      // OpenGL initializations
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA);// double buffering and RGB
    glutInitWindowSize(400, 400);               // create a 400x400 window
    glutInitWindowPosition(0, 0);               // ...in the upper left
    glutCreateWindow(argv[0]);                  // create the window

    glutDisplayFunc(myDisplay);                 // setup callbacks
    glutReshapeFunc(myReshape);
    glutMainLoop();                             // start it running
    return 0;                                   // ANSI C expects this
}
```

Fig. 13: Sample OpenGL Program: Header and Main program.

Orientation: You have been asked to design the AI for a non-player agent in a flight combat simulator. You detect the presence of a enemy aircraft in a certain direction. How should you rotate your aircraft to either attack (or escape from) this threat?

Change of coordinates: We know the position of an object on a table with respect to a coordinate system associated with the table. We know the position of the table with respect to a coordinate system associated with the room. What is the position of the object with respect to the coordinate system associated with the room?

Reflection and refraction: A wine glass filled with red wine sits on a table. A bright light illuminates the glass from one side. The light passes through the glass and the liquid, and casts an interesting pattern of light and dark regions on the table on the far side. Can you simulate this effect? The wine glass is replaced with a shiny metal sculpture, and now, rather than passing through, the light is reflected off the various surfaces of the sculpture and is cast onto the table.

Such basic geometric problems are fundamental to computer graphics, and over the next few lectures, our goal will be to present the tools needed to answer these sorts of questions. (By the way, a good source of information on how to solve these problems is the series of books entitled "Graphics Gems". Each book is a collection of many simple graphics problems and provides algorithms for solving them.)

There are various formal geometric systems that arise naturally in computer graphics applications. The principal ones are:

Affine Geometry: The geometry of simple "flat things": points, lines, planes, line segments, triangles, etc. There is no defined notion of distance, angles, or orientations, however.

Euclidean Geometry: The geometric system that is most familiar to us. It enhances affine geometry by adding notions such as distances, angles, and orientations (such as clockwise and counterclockwise).

Projective Geometry: In Euclidean geometry, there is no notion of infinity (in the same way that in standard arithmetic, you cannot divide by zero). But in graphics, we often need to deal with infinity. (For example, two parallel lines in 3-dimensional space can meet at a common *vanishing point* in a perspective rendering. Think of the point in the distance where two perfectly straight train tracks appear to meet. Computing this vanishing point requires us to consider points at infinity.) Projective geometry permits this.

You might wonder where linear algebra enters. We will make use of linear algebra as a concrete reprensentational basis for these abstract geometric systems (in much the same way that a concrete structure like an array is used to represent an abstract structure like a stack in object-oriented programming). We will describe these systems, starting with the simplest, affine geometry.

Affine Geometry: The basic elements of *affine geometry* are:

- *scalars*, which we can just think of as being real numbers
- *points*, which define locations in space

- *free vectors* (or simply *vectors*), which are used to specify direction and magnitude, but have no fixed position.

The term "free" means that vectors do not necessarily emanate from some position (like the origin), but float freely about in space. There is a special vector called the *zero vector*, $\vec{0}$, that has no magnitude, such that $\vec{v} + \vec{0} = \vec{0} + \vec{v} = \vec{v}$.

Note that we did *not* define a *zero point* or "origin" for affine space. This is an intentional omission. No point special compared to any other point. (We will eventually have to break down and define an origin in order to have a coordinate system for our points, but this is a purely representational necessity, not an intrinsic feature of affine space.)

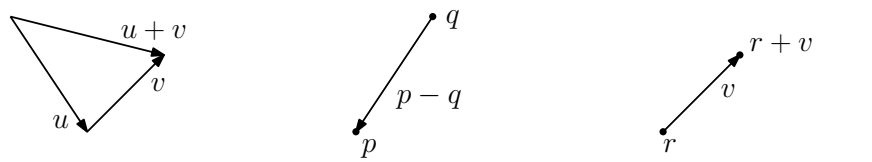
You might ask, why make a distinction between points and vectors? Although both can be represented in the same way as a list of coordinates, they represent very different concepts. For example, points would be appropriate for representing a vertex of a mesh, the center of mass of an object, the point of contact between two colliding objects. In contrast, a vector would be appropriate for representing the velocity of a moving object, the vector normal to a surface, the axis about which a rotating object is spinning. (As computer scientists the idea of different abstract objects sharing a common representation should be familiar. For example, stacks and queues are two different abstract data types, but they can both be represented as a 1-dimensional array.)

Because points and vectors are conceptually different, it is not surprising that the operations that can be applied to them are different. For example, it makes perfect sense to multiply a vector and a scalar. Geometrically, this corresponds to stretching the vector by this amount. It also makes sense to add two vectors together. This involves the usual head-to-tail rule, which you learn in linear algebra. It is not so clear, however, what it means to multiply a point by a scalar. (For example, the top of the Washington monument is a point. What would it mean to multiply this point by 2?) On the other hand, it does make sense to add a vector to a point. For example, if a vector points straight up and is 3 meters long, then adding this to the top of the Washington monument would naturally give you a point that is 3 meters above the top of the monument.

We will use the following notational conventions. Points will usually be denoted by lower-case Roman letters such as p, q, and r. Vectors will usually be denoted with lower-case Roman letters, such as u, v, and w, and often to emphasize this we will add an arrow (e.g., $\vec{u}$, $\vec{v}$, $\vec{w}$). Scalars will be represented as lower case Greek letters (e.g., α, β, γ). In our programs, scalars will be translated to Roman (e.g., a, b, c). (We will sometimes violate these conventions, however. For example, we may use c to denote the center point of a circle or r to denote the scalar radius of a circle.)

Affine Operations: The table below lists the valid combinations of scalars, points, and vectors. The formal definitions are pretty much what you would expect. Vector operations are applied in the same way that you learned in linear algebra. For example, vectors are added in the usual "tail-to-head" manner (see Fig. 14). The difference $p - q$ of two points results in a free vector directed from q to p. Point-vector addition $r + \vec{v}$ is defined to be the translation of r by displacement $\vec{v}$. Note that some operations (e.g. scalar-point multiplication, and addition of points) are explicitly not defined.

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



Vector addition Point subtraction Point-vector addition

Fig. 14: Affine operations.

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

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

$$r = p + \alpha(q - p),$$

(see Fig. 15(a)). Another way to think of this point r is as a *weighted average* of the endpoints p and q. Thinking of r in these terms, we might be tempted to rewrite the above formula in the following (technically illegal) manner:

$$r = (1 - \alpha)p + \alpha q,$$

(see Fig. 15(b)). Observe that as α ranges from 0 to 1, the point r ranges along the line segment from p to q. In fact, we may allow to become negative in which case r lies to the left of p, and if $\alpha > 1$, then r lies to the right of q (see Fig. 15(c)). The special case when $0 \le \alpha \le 1$, this is called a *convex combination*.

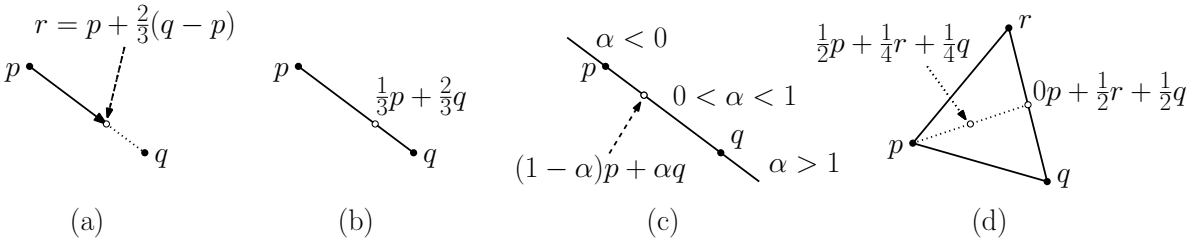


Fig. 15: Affine combinations.

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

Affine combination: Given a sequence of points $p_1, p_2, \ldots, p_n$, an affine combination is any sum of the form

$$\alpha_1 p_1 + \alpha_2 p_2 + \ldots + \alpha_n p_n,$$

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

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

Affine and convex combinations have a number of nice uses in graphics. For example, any three noncollinear points determine a plane. There is a 1–1 correspondence between the points on this plane and the affine combinations of these three points. Similarly, there is a 1–1 correspondence between the points in the triangle determined by the these points and the convex combinations of the points (see Fig. 15(d)). In particular, the point $(1/3)p + (1/3)q + (1/3)r$ is the *centroid* of the triangle.

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

$$r = \frac{p+q}{2}.$$

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

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

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

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

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

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

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

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

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

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

Using the dot product we may define a number of concepts, which are not defined in regular affine geometry (see Fig. 16). Note that these concepts generalize to all dimensions.

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

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

Distance between points: $\text{dist}(p, q) = \|p - q\|$.

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

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

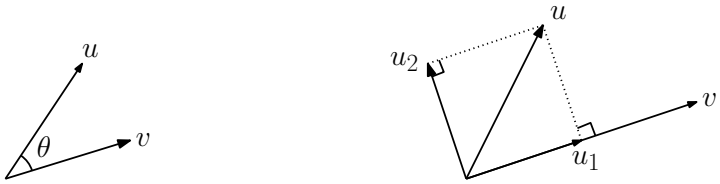
This is easy to derive from the law of cosines. Note that this does not provide us with a signed angle. We cannot tell whether $\vec{u}$ is clockwise our counterclockwise relative to $\vec{v}$. We will discuss signed angles when we consider the cross-product.

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

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

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

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



Angle between vectors Orthogonal projection and its complement

Fig. 16: The dot product and its uses.

Lecture 6: More on Geometry and Geometric Programming

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

The first question is how to represent points and vectors in affine space. We will begin by recalling how to do this in linear algebra, and generalize from there. We know from linear

algebra that if we have 2-linearly independent vectors, $\vec{u}_0$ and $\vec{u}_1$ in 2-space, then we can represent any other vector in 2-space uniquely as a *linear combination* of these two vectors (see Fig. 17(a)):

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

for some choice of scalars α_0, α_1. Thus, given any such vectors, we can use them to represent any vector in terms of a triple of scalars (α_0, α_1). In general d linearly independent vectors in dimension d is called a *basis*. The most convenient basis to work with consists of two vectors, each of unit length, that are orthogonal to each other. Such a collection of vectors is said to be *orthonormal*. The *standard basis* consisting of the x- and y-unit vectors is orthonormal (see Fig. 17(b)).

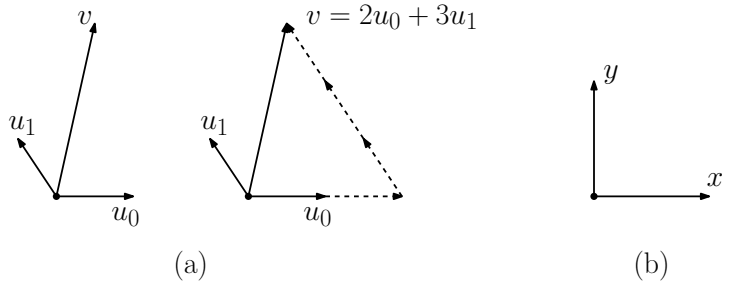


Fig. 17: Bases and linear combinations in linear algebra (a) and the standard basis (b).

Note that we are using the term "vector" in two different senses here, one as a geometric entity and the other as a sequence of numbers, given in the form of a row or column. The first is the object of interest (i.e., the abstract data type, in computer science terminology), and the latter is a representation. As is common in object oriented programming, we should "think" in terms of the abstract object, even though in our programming we will have to get dirty and work with the representation itself.

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

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

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

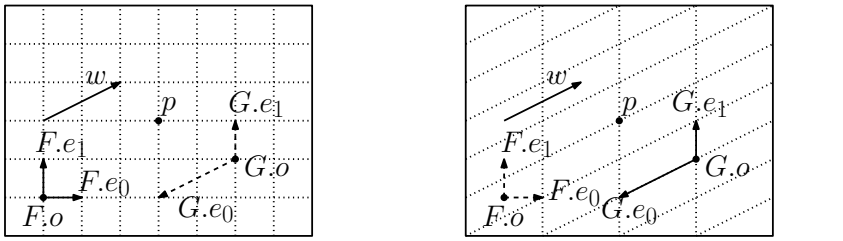
Definition: A *coordinate frame* for a d-dimensional affine space consists of a point, called the *origin* (which we will denote o) of the frame, and a set of d linearly independent *basis vectors*.

In Fig. 18 we show a point p and vector $\overrightarrow{w}$. We have also given two coordinate frames, F and G. Observe that p and $\overrightarrow{w}$ can be expressed as functions of F and G as follows:

$$\begin{aligned} p &= 3 \cdot F.\vec{e}_0 + 2 \cdot F.\vec{e}_1 + F.o \\ \overrightarrow{w} &= 2 \cdot F.\vec{e}_0 + 1 \cdot F.\vec{e}_1 \end{aligned}$$

$$\begin{aligned} p &= 1 \cdot G.\vec{e}_0 + 2 \cdot G.\vec{e}_1 + G.o \\ \overrightarrow{w} &= -1 \cdot G.\vec{e}_0 + 0 \cdot G.\vec{e}_1 \end{aligned}$$

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



$p_{[F]} = (3, 2, 1)$ $w_{[F]} = (2, 1, 0)$ $p_{[G]} = (1, 2, 1)$ $w_{[F]} = (-1, 0, 0)$

Fig. 18: Coordinate Frames.

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

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

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

$$\begin{aligned} p &= 3 \cdot F.\vec{e}_0 + 2 \cdot F.\vec{e}_1 + 1 \cdot F.o \\ \overrightarrow{w} &= 2 \cdot F.\vec{e}_0 + 1 \cdot F.\vec{e}_1 + 0 \cdot F.o \end{aligned}$$

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

$$p_{[F]} = \begin{pmatrix} 3 \\ 2 \\ 1 \end{pmatrix} \quad \text{and} \quad \overrightarrow{w}_{[F]} = \begin{pmatrix} 2 \\ 1 \\ 0 \end{pmatrix}.$$

We will call these *homogeneous coordinates* relative to frame F. In some linear algebra conventions, vectors are written as row vectors and some as column vectors. We will stick

with OpenGL's conventions, of using column vectors, but we may be sloppy from time to time.

As we said before, the term "vector" has two meanings: one as a *free vector* in an affine space, and now as a *coordinate vector*. Usually, it will be clear from context which meaning is intended.

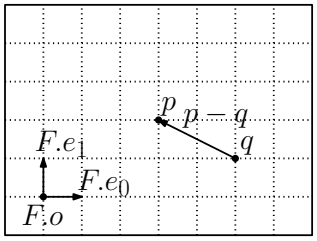
In general, to represent points and vectors in d-space, we will use coordinate vectors of length $d+1$. Points have a last coordinate of 1, and vectors have a last coordinate of 0. Some authors put the homogenizing coordinate first rather than last. There are actually good reasons for doing this. But we will stick with standard engineering conventions and place it last.

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

For example, consider two points p and q whose coordinate representations relative to some frame F are $p_{[F]} = (3, 2, 1)^\mathsf{T}$ and $q_{[F]} = (5, 1, 1)^\mathsf{T}$, respectively. Consider the vector

$$\vec{v} = p - q.$$

If we apply the difference rule that we defined last time for points, and then convert this vector into it coordinates relative to frame F, we find that $\vec{v}_{[F]} = (-2, 1, 0)^\mathsf{T}$. Thus, to compute the coordinates of $p - q$ we simply take the component-wise difference of the coordinate vectors for p and q. The 1-components nicely cancel out, to give a vector result (see Fig. 19).



$$p_{[F]} = \begin{pmatrix} 3 \\ 2 \\ 1 \end{pmatrix} \qquad q_{[F]} = \begin{pmatrix} 5 \\ 1 \\ 1 \end{pmatrix}$$

$$(p - q)_{[F]} = \begin{pmatrix} -2 \\ 1 \\ 0 \end{pmatrix}$$

Fig. 19: Coordinate arithmetic.

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

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

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

This suggests how one might do type checking for a coordinate-free geometry system. Points and vectors are stored using a common base type, which simply consists of a 4-element array of scalars. We allow the programmer to perform any combination of standard vector operations

on coordinates. Just prior to assignment, check that the last coordinate is either 0 or 1, appropriate to the type of variable into which you are storing the result. This allows much more flexibility in creating expressions, such as:

$$centroid \leftarrow \frac{p+q+r}{3},$$

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

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

The cross product is usually defined in standard linear 3-space (since it applies to vectors, not points). So we will ignore the homogeneous coordinate here. Given two vectors in 3-space, $\vec{u}$ and $\vec{v}$, their *cross product* is defined as follows (see Fig. 20(a)):

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

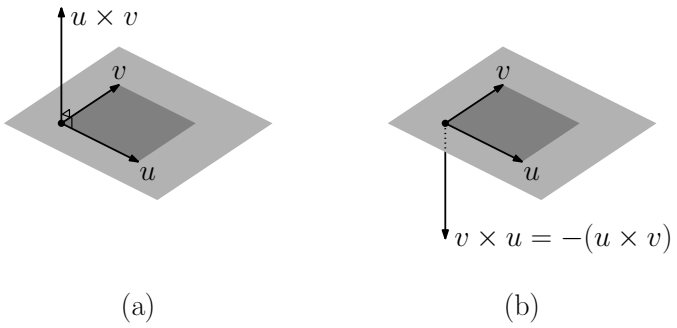


Fig. 20: Cross product.

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

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

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

Skew symmetric: $\vec{u} \times \vec{v} = -(\vec{v} \times \vec{u})$ (see Fig. 20(b)). It follows immediately that $\vec{u} \times \vec{u} = 0$ (since it is equal to its own negation).

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

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

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

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

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

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

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

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

The cross product is commonly used in computer graphics for generating coordinate frames. Given two basis vectors for a frame, it is useful to generate a third vector that is orthogonal to the first two. The cross product does exactly this. It is also useful for generating surface normals. Given two tangent vectors for a surface, the cross product generate a vector that is normal to the surface.

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

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

$$\text{Or}_2(p, q, r) = \text{sign det} \begin{pmatrix} 1 & 1 & 1 \\ p_x & q_x & r_x \\ p_y & q_y & r_y \end{pmatrix}, \qquad \text{Or}_3(p, q, r, s) = \text{sign det} \begin{pmatrix} 1 & 1 & 1 & 1 \\ p_x & q_x & r_x & s_x \\ p_y & q_y & r_y & s_y \\ p_z & q_z & r_z & s_z \end{pmatrix}.$$

What does orientation mean intuitively? The orientation of three points in the plane is $+1$ if the triangle PQR is oriented counter-clockwise, -1 if clockwise, and 0 if all three points are collinear (see Fig. 21). In 3-space, a positive orientation means that the points follow a right-handed screw, if you visit the points in the order $PQRS$. A negative orientation

means a left-handed screw and zero orientation means that the points are coplanar. Note that the order of the arguments is significant. The orientation of (p, q, r) is the negation of the orientation of (p, r, q). As with determinants, the swap of any two elements reverses the sign of the orientation.

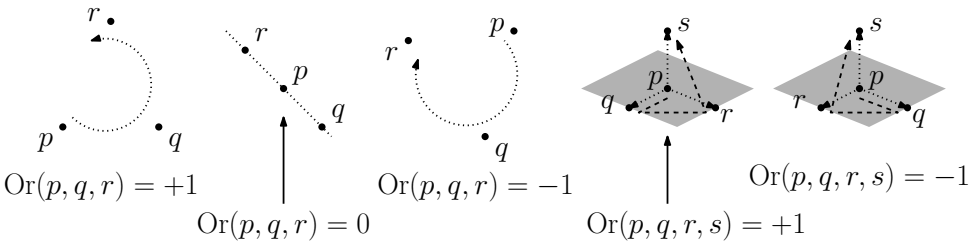


Fig. 21: Orientations in 2 and 3 dimensions.

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

The value of the determinant itself is the area of the parallelogram defined by the vectors $q - p$ and $r - p$, and thus this determinant is also handy for computing areas and volumes. Later we will discuss other methods.

Lecture 7: Drawing in OpenGL: Transformations

More about Drawing: So far we have discussed how to draw simple 2-dimensional objects using OpenGL. Suppose that we want to draw more complex scenes. For example, we want to draw objects that move and rotate or to change the projection. We could do this by computing (ourselves) the coordinates of the transformed vertices. However, this would be inconvenient for us. It would also be inefficient. OpenGL provides methods for downloading large geometric specifications directly to the GPU. However, if the coordinates of these object were changed with each display cycle, this would negate the benefit of loading them just once.

For this reason, OpenGL provides tools to handle transformations. Today we consider how this is done in 2-space. This will form a foundation for the more complex transformations, which will be needed for 3-dimensional viewing.

Transformations: Linear and affine transformations are central to computer graphics. Recall from your linear algebra class that a *linear transformation* is a mapping in a vector space that preserves linear combinations. Such transformations include rotations, scalings, shearings (which stretch rectangles into parallelograms), and combinations thereof.

As you might expect, *affine transformations* are transformations that preserve affine combinations. For example, if p and q are two points and m is their midpoint, and T is an affine transformation, then the midpoint of $T(p)$ and $T(q)$ is $T(m)$. Important features of

affine transformations include the facts that they map straight lines to straight lines, they preserve parallelism, and they can be implemented through matrix multiplication. They arise in various ways in graphics.

Moving Objects: As needed in animations.

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

Projection: Such transformations are used to project objects from the idealized drawing window to the viewport, and mapping the viewport to the graphics display window. (We shall see that perspective projection transformations are more general than affine transformations, since they may not preserve parallelism.)

Mapping between Surfaces: This is useful when textures are mapped onto object surfaces as part of texture mapping.

OpenGL has a very particular model for how transformations are performed. Recall that when drawing, it was convenient for us to first define the drawing attributes (such as color) and then draw a number of objects using that attribute. OpenGL uses much the same model with transformations. You specify a transformation *first*, and then this transformation is automatically applied to every object that is drawn *afterwards*, until the transformation is set again. It is important to keep this in mind, because it implies that you must always set the transformation prior to issuing drawing commands.

Because transformations are used for different purposes, OpenGL maintains three sets of matrices for performing various transformation operations. These are:

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

Projection matrix: Handles parallel and perspective projections. (Used for the third task above.)

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

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

Understanding how OpenGL maintains and manipulates transformations through these matrices is central to understanding how OpenGL and other modern immediate-mode rendering systems (such as DirectX) work.

Matrix Stacks: For each matrix type, OpenGL maintains a *stack* of matrices. The *current matrix* is the one on the top of the stack. It is the matrix that is being applied at any given time. The stack mechanism allows you to save the current matrix (by pushing the stack down) and

restoring it later (by popping the stack). We will discuss the entire process of implementing affine and projection transformations later in the semester. For now, we'll give just basic information on OpenGL's approach to handling matrices and transformations.

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

glMatrixMode(*mode*);

where *mode* is either GL_MODELVIEW, GL_PROJECTION, or GL_TEXTURE. The default mode is GL_MODELVIEW.

GL_MODELVIEW is by far the most common mode, the convention in OpenGL programs is to assume that you are always in this mode. If you want to modify the mode for some reason, you first change the mode to the desired mode (GL_PROJECTION or GL_TEXTURE), perform whatever operations you want, and then immediately change the mode back to GL_MODELVIEW.

Once the matrix mode is set, you can perform various operations to the stack. OpenGL has an unintuitive way of handling the stack. Note that most operations below (except glPushMatrix()) alter the contents of the matrix at the top of the stack.

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

glLoadMatrix*(M): Loads (copies) a given matrix over the current matrix. (The '*' can be either 'f' or 'd' depending on whether the elements of M are GLfloat or GLdouble, respectively.)

glMultMatrix*(M): Post-multiplies the current matrix by a given matrix and replaces the current matrix with this result. Thus, if C is the current matrix on top of the stack, it will be replaced with the matrix product $C \cdot M$. (As above, the '*' can be either 'f' or 'd' depending on M.)

glPushMatrix(): Pushes a copy of the current matrix on top the stack. (Thus the stack now has two copies of the top matrix.)

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

An example is shown in Fig. 22. We will discuss how matrices like M are presented to OpenGL later in the semester. There are a number of other matrix operations, which we will also discuss later.

Warning: OpenGL assumes that all matrices are 4×4 homogeneous matrices, stored in *column-major order*. (In contrast, most modern programming languages linearize 2-dimensional arrays by storing them in row-major order.) That is, a matrix is presented as an array of 16 values, where the first four values give column 0 (for x), then column 1 (for y), then column 2 (for z), and finally column 3 (for the homogeneous coordinate, usually called w). For example, given a matrix M and vector v, OpenGL assumes the following

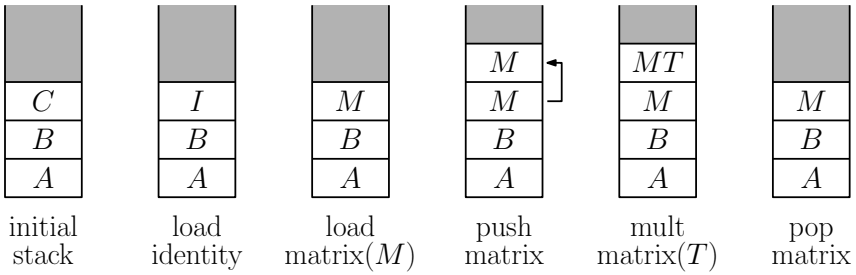


Fig. 22: Matrix stack operations.

representation:

$$M \cdot v \;=\; \begin{pmatrix} m[0] & m[4] & m[8] & m[12] \\ m[1] & m[5] & m[9] & m[13] \\ m[2] & m[6] & m[10] & m[14] \\ m[3] & m[7] & m[11] & m[15] \end{pmatrix} \begin{pmatrix} v[0] \\ v[1] \\ v[2] \\ v[3] \end{pmatrix}$$

Automatic Evaluation and the Transformation Pipeline: Now that we have described the matrix stack, the next question is how do we apply the matrix to some point that we want to transform? Understanding the answer is critical to understanding how OpenGL (and actually display processors) work. The answer is that it happens *automatically*. In particular, *every* vertex (and hence virtually every geometric object that is drawn) is passed through a series of matrices, as shown in Fig. 23. This may seem rather inflexible, but it is because of the simple uniformity of sending every vertex through this transformation sequence that makes graphics cards run so fast. As mentioned above, these transformations behave much like drawing attributes—you set them, do some drawing, alter them, do more drawing, etc.

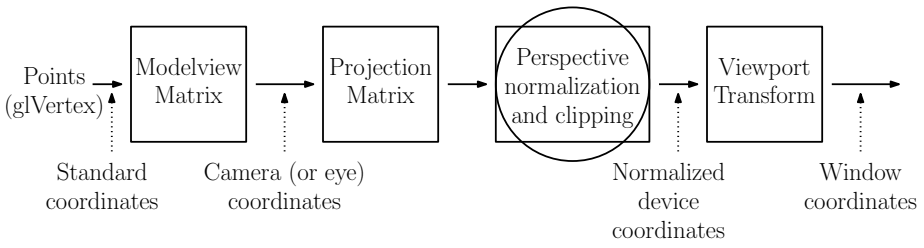


Fig. 23: Transformation pipeline.

A second important thing to understand is that OpenGL's transformations do not alter the state of the objects you are drawing. They simply modify things before they get drawn. For example, suppose that you draw a unit square ($U = [0,1] \times [0,1]$) and pass it through a matrix that scales it by a factor of 5. The square U itself has not changed; it is still a unit square. If you wanted to change the actual representation of U to be a 5×5 square, then you need to perform your own modification of U's representation.

You might ask, "what if I do *not* want the current transformation to be applied to some object?" The answer is, "tough luck." There are no exceptions to this rule (other than commands that act directly on the viewport). If you do not want a transformation to be

applied, then to achieve this, you load an identity matrix on the top of the transformation stack, then do your (untransformed) drawing, and finally pop the stack.

Example: Rotating a Rectangle (first attempt): The Modelview matrix is useful for applying transformations to objects, which would otherwise require you to perform your own linear algebra. Suppose that rather than drawing a rectangle that is aligned with the coordinate axes, you want to draw a rectangle that is rotated by 20 degrees (counterclockwise) and centered at some point (x, y). The desired result is shown in Fig. 24. Of course, as mentioned above, you could compute the rotated coordinates of the vertices yourself (using the appropriate trigonometric functions), but OpenGL provides a way of doing this transformation more easily.

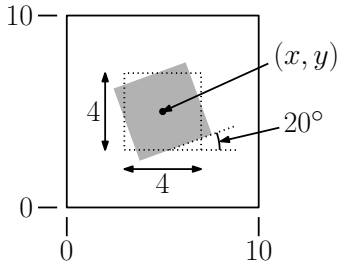


Fig. 24: Desired drawing. (Rotated rectangle is shaded).

Suppose that we are drawing within the square, $0 \leq x, y \leq 10$, and we have a 4×4 sized rectangle to be drawn centered at location (x, y). We could draw an unrotated rectangle with the following command:

```
glRectf(x - 2, y - 2, x + 2, y + 2);
```

Note that the arguments should be of type GLfloat (`2.0f` rather than `2`), but we will let the compiler cast the integer constants to floating point values for us.

Now let us draw a rotated rectangle. Let us assume that the matrix mode is `GL_MODELVIEW` (this is the default). Generally, there will be some existing transformation (call it M) currently present in the Modelview matrix. This usually represents some more global transformation, which is to be applied on top of our rotation. For this reason, we will compose our rotation transformation with this existing transformation.

Because the OpenGL rotation function destroys the contents of the Modelview matrix, we will begin by saving it, by using the command `glPushMatrix()`. Saving the Modelview matrix in this manner is not always required, but it is considered good form. Then we will compose the current matrix M with an appropriate rotation matrix R. Then we draw the rectangle (in upright form). Since all points are transformed by the Modelview matrix prior to projection, this will have the effect of rotating our rectangle. Finally, we will pop off this matrix (so future drawing is not rotated).

To perform the rotation, we will use the command `glRotatef(ang, x, y, z)`. All arguments are `GLfloat`'s. (Or, recalling OpenGL's naming convention, we could use `glRotated()` which takes

GLdouble arguments.) This command constructs a matrix that performs a rotation in 3-dimensional space counterclockwise by angle *ang* degrees, about the vector (x, y, z). It then *composes* (or multiplies) this matrix with the current Modelview matrix. In our case the angle is 20 degrees. To achieve a rotation in the (x, y) plane the vector of rotation would be the z-unit vector, $(0, 0, 1)$. Here is how the code might look (but beware, this conceals a subtle error).

Drawing an Rotated Rectangle (First Attempt)

```
glPushMatrix();                 // save the current matrix
  glRotatef(20, 0, 0, 1);       // rotate by 20 degrees CCW
  glRectf(x-2, y-2, x+2, y+2);  // draw the rectangle
glPopMatrix();                  // restore the old matrix
```

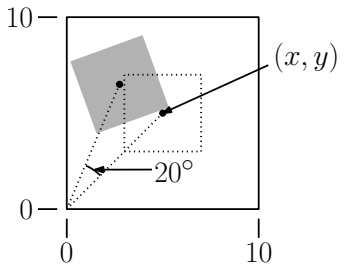


Fig. 25: The actual drawing produced by the previous example. (Rotated rectangle is shaded).

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

Example: Rotating a Rectangle (correct): Something is wrong with this example given above. What is it? The answer is that the rotation is performed about the origin of the coordinate system, not about the center of the rectangle and we want.

Fortunately, there is an easy fix. Conceptually, we will draw the rectangle centered at the origin, then rotate it by 20 degrees, and finally *translate* (or move) it by the vector (x, y). To do this, we will need to use the command glTranslatef(x, y, z). All three arguments are GLfloat's. (And there is version with GLdouble arguments.) This command creates a matrix which performs a translation by the vector (x, y, z), and then composes (or multiplies) it with the current matrix. Recalling that all 2-dimensional graphics occurs in the $z = 0$ plane, the desired translation vector is $(x, y, 0)$.

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

current Modelview matrix, then we want to compute the product

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

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

Drawing/Transformation Order in OpenGL's

First, conceptualize your intent by drawing about the origin and then applying the appropriate transformations to map your object to its desired location. Then implement this by applying transformations in *reverse order*, and do your drawing. It is always a good idea to enclose everything in a push-matrix and pop-matrix pair.

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

(1) Push the matrix stack,

(2) Apply (i.e., multiply) all the desired transformation matrices with the current matrix, but *in the reverse order* from which you would like them to be applied to your object,

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

(4) Pop the matrix stack.

The final and correct fragment of code for the rotation is shown in the code block below.

Drawing an Rotated Rectangle (Correct)

```
glPushMatrix();                 // save the current matrix (M)
  glTranslatef(x, y, 0);        // apply translation (T)
  glRotatef(20, 0, 0, 1);       // apply rotation (R)
  glRectf(-2, -2, 2, 2);        // draw rectangle at the origin
glPopMatrix();                  // restore the old matrix (M)
```

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

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

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

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

As an exercise in deriving linear transformations, let us consider doing this all in one shot. Let W denote the idealized drawing window and let V denote the viewport. Let w_r, w_l, w_b, and w_t denote the left, right, bottom and top of the window. Define v_r, v_l, v_b, and v_t similarly for the viewport. We wish to derive a linear transformation that maps a point (x, y) in window coordinates to a point (x', y') in viewport coordinates. See Fig. 26.

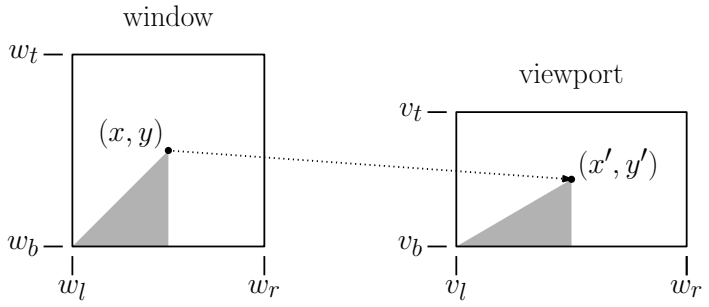


Fig. 26: Window to Viewport transformation.

Let $f(x, y)$ denote the desired transformation. Since the function is linear, and it operates on x and y independently, we have

$$(x', y') = f(x, y) = (s_x x + t_x, s_y y + t_y),$$

where s_x, t_x, s_y and t_y, depend on the window and viewport coordinates. Let's derive what s_x and t_x are using simultaneous equations. We know that the x-coordinates for the left and right sides of the window (w_l and w_r) should map to the left and right sides of the viewport (v_l and v_r). Thus we have

$$s_x w_l + t_x = v_l \qquad \text{and} \qquad s_x w_r + t_x = v_r.$$

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

$$s_x = \frac{v_r - v_l}{w_r - w_l}.$$

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

$$t_x \;=\; v_l - s_x w_l \;=\; v_l - \frac{v_r - v_l}{w_r - w_l} w_l \;=\; \frac{v_l w_r - v_r w_l}{w_r - w_l}.$$

A similar derivation for s_y and t_y yields

$$s_y = \frac{v_t - v_b}{w_t - w_b} \qquad t_y = \frac{v_b w_t - v_t w_b}{w_t - w_b}.$$

These four formulas give the desired final transformation.

$$f(x,y) = \left(\frac{(v_r - v_l)x + (v_l w_r - v_r w_l)}{w_r - w_l}, \frac{(v_t - v_b)y + (v_b w_t - v_t w_b)}{w_t - w_b} \right).$$

This can be expressed in matrix form as

$$\begin{pmatrix} \frac{v_r - v_l}{w_r - w_l} & 0 & \frac{v_l w_r - v_r w_l}{w_r - w_l} \\ 0 & \frac{v_t - v_b}{w_t - w_b} & \frac{v_b w_t - v_t w_b}{w_t - w_b} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix},$$

which is essentially what OpenGL stores internally.

Lecture 8: Affine Transformations

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

We will consider a limited, but interesting class of transformations, called *affine transformations*. These include (among others) the following transformations of space: translations, rotations, uniform and nonuniform scalings (stretching the axes by some constant scale factor), reflections (flipping objects about a line) and shearings (which deform squares into parallelograms). They are illustrated in Fig. 27.

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

You might ask, since OpenGL provides commands such as glTranslate and glRotate, why do we need to know how to implement these in linear algebra? There are two reasons that come to mind. First, sometimes it is not sufficient to perform these transformations just for the purposes of drawing. You actually need to modify the internal representation of the geometric data itself. (For example, if you are passing this information to a module that performs

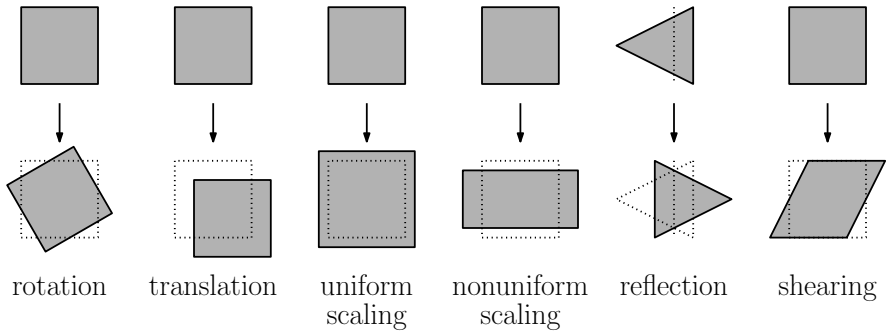


Fig. 27: Examples of affine transformations.

collision detection.) The second is that in the latest versions of OpenGL, the aforementioned transformation commands are not provided. You will need to implement them yourself (or at least, find a linear algebra library that will do them for you).

Recall that an *affine transformation* is a mapping from one affine space to another (typically the same space) that preserves affine combinations. For example, this implies that given any affine transformation T and two points p and q, and any scalar α,

$$r \;=\; (1-\alpha)p + \alpha q \qquad \Rightarrow \qquad T(r) \;=\; (1-\alpha)T(p) + \alpha T(q).$$

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

Matrix Representation of Affine Transformations: Let us concentrate on transformations in 3-space. Let F denote a coordinate frame in $\mathbb{R}^3$. Recall that such a coordinate frame is defined by an origin point, denoted $F.o$, and three basis vectors, denoted $F.\vec{e}_0$, $F.\vec{e}_1$, and $F.\vec{e}_2$. An important consequence of the preservation of affine relations is the following, where r is any point or free vector:

$$\begin{aligned} r \;&=\; \alpha_0 F.\vec{e}_0 + \alpha_1 F.\vec{e}_1 + \alpha_2 F.\vec{e}_2 + \alpha_3 F.o \\ &\Rightarrow \\ T(r) \;&=\; \alpha_0 T(F.\vec{e}_0) + \alpha_1 T(F.\vec{e}_1) + \alpha_2 T(F.\vec{e}_2) + \alpha_3 T(F.o). \end{aligned}$$

Recall that α_3 is either 0 (if r is a vector) or 1 (if r is a point). The equation on the left is the representation of a point or vector r in terms of the coordinate frame F. This implication shows that if we know the image of the frame elements under the transformation, then we can determine the image r under the transformation.

Recall that the homogeneous coordinate representation of r relative to frame F, which we denote by $r[F]$, is the 4-element column vector $(\alpha_0, \alpha_1, \alpha_2, \alpha_3)^{\mathsf{T}}$. (Recall that the superscript "$(\ldots)^{\mathsf{T}}$" in this context means to transpose this row vector into a column vector, and should not be confused with the transformation T.) Applying T to the ith unit vector in F's frame is denoted by $T(F.\vec{e}_i)$. This is a just a free vector. Its representation as a homogeneous (column) vector in F's frame $T(F.\vec{e}_i)[F]$. Thus, we can express the above relationship in the

following matrix form.

$$\begin{aligned} T(r)[F] &= \alpha_0 T(F.\vec{e}_0)[F] + \alpha_1 T(F.\vec{e}_1)[F] + \alpha_2 T(F.\vec{e}_2)[F] + \alpha_3 T(F.o)[F] \\ &= \left(T(F.\vec{e}_0)[F] \;\middle|\; T(F.\vec{e}_1)[F] \;\middle|\; T(F.\vec{e}_2)[F] \;\middle|\; T(F.o)[F] \right) \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{pmatrix}. \end{aligned}$$

This is a lot of notation, but its meaning is pretty simple. The columns of the above matrix are the representation (relative to frame F) of the images of the basis elements of the frame under T. This implies that applying an affine transformation (in coordinate form) is equivalent to multiplying the coordinates by a matrix. In dimension d this is a $(d+1) \times (d+1)$ matrix.

If this all seems a bit abstract. In the remainder of the lecture we will give a number of concrete examples of how this applies to various transformations. Rather than considering this in the context of 2-dimensional transformations, let's consider it in the more general setting of 3-dimensional transformations. The two dimensional cases can be extracted by just ignoring the rows and columns for the z-coordinates.

Translation: Translation by a fixed vector $\vec{v}$ maps any point p to $p+\vec{v}$. Note that, since free vectors have no position in space, they are not altered by translation (see Fig. 28(a)). Suppose that relative to the standard frame, $v[F] = (\alpha_x, \alpha_y, \alpha_z, 0)^\mathsf{T}$ are the homogeneous coordinates of $\vec{v}$. The three unit vectors are unaffected by translation, and the origin is mapped to $o+\vec{v}$, whose homogeneous coordinates are $(\alpha_x, \alpha_y, \alpha_z, 1)$. Thus, by the rule given earlier, the homogeneous matrix representation for this translation transformation is

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

This is the matrix used by OpenGL in the call glTranslatef$(\alpha_x, \alpha_y, \alpha_z)$.

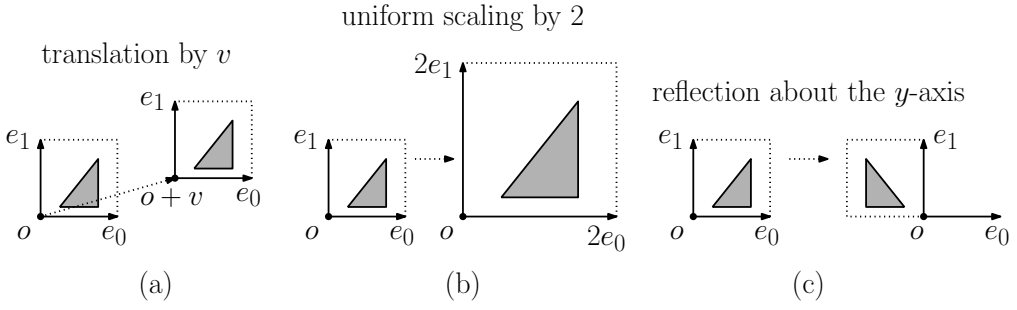


Fig. 28: Derivation of transformation matrices.

Scaling: *Uniform scaling* is a transformation which is performed relative to some central fixed point. We will assume that this point is the origin of the standard coordinate frame. (We will leave the general case as an exercise.) Given a scalar β, this transformation maps

the object (point or vector) with coordinates $(\alpha_x, \alpha_y, \alpha_z, \alpha_w)^\mathsf{T}$ to $(\beta\alpha_x, \beta\alpha_y, \beta\alpha_z, \alpha_w)^\mathsf{T}$ (see Fig. 28(b)).

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

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

Observe that both points and vectors are altered by scaling. This is the matrix used by OpenGL in the call $\mathsf{glScalef}(\beta_x, \beta_y, \beta_z)$.

Reflection: A reflection in the plane is given a line and maps points by flipping the plane about this line. A reflection in 3-space is given a plane, and flips points in space about this plane. In this case, reflection is just a special case of scaling, but where the scale factor is negative. For example, to reflect points about the yz-coordinate plane, we want to scale the x-coordinate by -1 (see Fig. 28(c)). Using the scaling matrix above, we have the following transformation matrix:

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

The cases for the other two coordinate frames are similar. Reflection about an arbitrary line or plane is left as an exercise.

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

Consider the rotation about the z-axis. The z-unit vector and origin are unchanged. The x-unit vector is mapped to $(\cos\theta, \sin\theta, 0, 0)^\mathsf{T}$, and the y-unit vector is mapped to $(-\sin\theta, \cos\theta, 0, 0)^\mathsf{T}$ (see Fig. 29(a)). Thus the rotation matrix is:

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

Observe that both points and vectors are altered by rotation. This is the matrix used by OpenGL in the call $\mathsf{glRotatef}(\theta \cdot 180/\pi, 0, 0, 1)$.

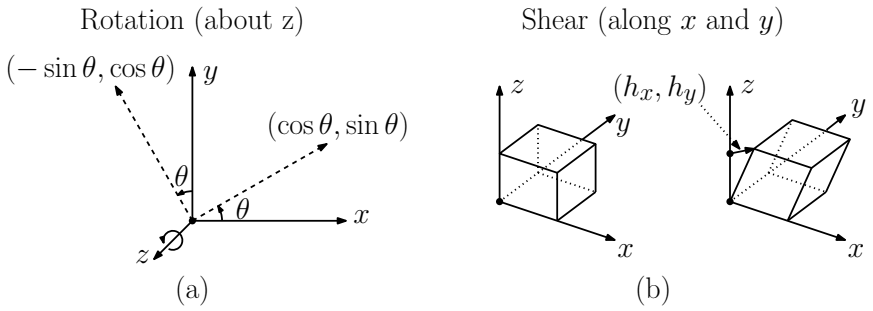


Fig. 29: Rotation and shearing.

For the other two axes we have

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

Shearing: A shearing transformation is perhaps the hardest of the group to visualize. Think of a shear as a transformation that maps a square into a parallelogram by sliding one side parallel to itself while keeping the opposite side fixed. In 3-dimensional space, it maps a cube into a parallelepiped by sliding one face parallel while keeping the opposite face fixed (see Fig. 29(b)). We will consider the simplest form, in which we start with a unit cube whose lower left corner coincides with the origin. Consider one of the axes, say the z-axis. The face of the cube that lies on the xy-coordinate plane does not move. The face that lies on the plane $z = 1$, is translated by a vector (h_x, h_y). In general, a point $p = (p_x, p_y, p_z, 1)$ is translated by the vector $p_z(h_x, h_y, 0, 0)$. This vector is orthogonal to the z-axis, and its length is proportional to the z-coordinate of p. This is called an *xy-shear*. (The yz- and xz-shears are defined analogously.)

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

$$H_{xy}(h_x, h_y) = \begin{pmatrix} 1 & 0 & h_x & 0 \\ 0 & 1 & h_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Shears involving any other pairs of axes are defined analogously.

$$H_{yz}(h_y, h_z) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ h_y & 1 & 0 & 0 \\ h_z & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad H_{zx}(h_z, h_x) = \begin{pmatrix} 1 & h_x & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & h_z & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Lecture 9: 3-d Viewing and Projective Geometry

Viewing in OpenGL: For the next couple of lectures we will discuss how viewing and perspective transformations are handled for 3-dimensional scenes. In OpenGL, and most similar graphics

systems, the process involves the following basic steps, of which the perspective transformation is just one component. We assume that all objects are initially represented relative to a standard 3-dimensional coordinate frame, in what are called *world coordinates*.

Modelview transformation: Maps objects (actually vertices) from their world-coordinate representation to one that is centered around the viewer. The resulting coordinates are variously called *camera coordinates*, *view coordinates*, or *eye coordinates*. (Specified by the OpenGL command gluLookAt.)

Projection transformation: This projects points in 3-dimensional eye-coordinates to points on a plane called the *image plane*. This projection process consists of three separate parts: the projection transformation, clipping, and perspective normalization. Each will be discussed below. The output coordinates are called *normalized device coordinates*. (Specified by the OpenGL commands such as gluOrtho2D, glOrtho, glFrustum, and gluPerspective.)

Mapping to the viewport: Convert the point from these idealized normalized device coordinates to the viewport. The coordinates are called *window coordinates* or *viewport coordinates*. (Specified by the OpenGL command glViewport.)

We have ignored a number of issues, such as lighting and hidden surface removal. These will be considered separately later. The process is illustrated in Fig. 30. We have already discussed the viewport transformation, so it suffices to discuss the first two transformations.

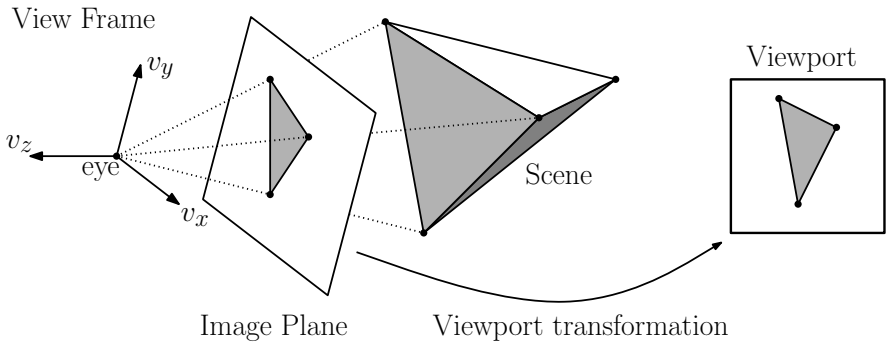


Fig. 30: OpenGL Viewing Process.

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

In OpenGL, there is a nice utility for doing this. The procedure gluLookAt generates the desired transformation to perform this change of coordinates and multiplies it times the

transformation at the top of the current transformation stack. (Recall OpenGL's transformation structure from the previous lecture on OpenGL transformations.) This should be done in Modelview mode.

Conceptually, this change of coordinates is performed *last*, after all other Modelview transformations are performed, and immediately before the projection. By the "reverse rule" of OpenGL transformations, this implies that this change of coordinates transformation should be the *first* transformation on the Modelview transformation matrix stack. Thus, it is almost always preceded by loading the identity matrix. Here is the typical calling sequence. This should be called when the camera position is set initially, and whenever the camera is (conceptually) repositioned in space.

Typical Structure of Redisplay Callback

```
void myDisplay() {
                                        // clear the buffer
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glLoadIdentity();                     // start fresh
                                        // set up camera frame
  gluLookAt(eyeX, eyeY, eyeZ, atX, atY, atZ, upX, upY, upZ);
  myWorld.draw();                       // draw your scene
  glutSwapBuffers();                    // make it all appear
}
```

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 $eye = (e_x, e_y, e_z)^\mathsf{T}$ is the *viewpoint*, that is the location of they viewer (or the camera). To indicate the direction that the camera is pointed, a central point at which the camera is directed is given by $at = (a_x, a_y, a_z)^\mathsf{T}$. The "at" point is significant only in that it defines the *viewing vector*, which indicates the direction that the viewer is facing. It is defined to be $at - eye$ (see Fig. 31).

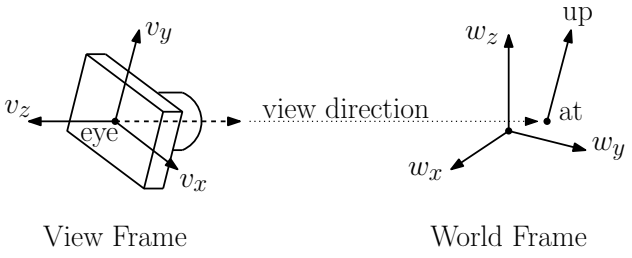


Fig. 31: The world frame, parameters to gluLookAt, and the camera frame.

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

The Camera Frame: OpenGL uses the arguments to gluLookAt to construct a coordinate frame centered at the viewer. The x- and y-axes are directed to the right and up, respectively, relative to the viewer. It might seem natural that the z-axes be directed in the direction that the viewer is facing, but this is not a good idea.

To see why, we need to discuss the distinction between right-handed and left-handed coordinate systems. Consider an orthonormal coordinate system with basis vectors $\vec{v}_x$, $\vec{v}_y$ and $\vec{v}_z$. This system is said to be *right-handed* if $\vec{v}_x \times \vec{v}_y = \vec{v}_z$, and left-handed otherwise ($\vec{v}_x \times \vec{v}_y = -\vec{v}_z$). Right-handed coordinate systems are used by default throughout mathematics. (Otherwise computation of orientations is all screwed up.) Given that the x- and y-axes are directed right and up relative to the viewer, if the z-axis were to point in the direction that the viewer is facing, this would result in left-handed coordinate system. The designers of OpenGL wisely decided to stick to a right-handed coordinate system, which requires that the z-axes is directed opposite to the viewing direction.

Building the Camera Frame: How does OpenGL implement this change of coordinate transformation? This turns out to be a nice exercise in geometric computation, so let's try it. We want to construct an orthonormal frame whose origin is the point *eye*, whose $(-z)$-basis vector is parallel to the view vector, and such that the $\overrightarrow{up}$ vector projects to the up direction in the final projection. (This is illustrated in the Fig. 32, where the x-axis is pointing outwards from the page.)

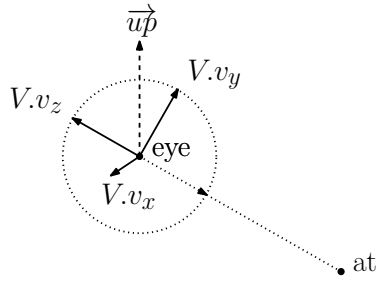


Fig. 32: The camera frame.

Let $V = (V.\vec{v}_x, V.\vec{v}_y, V.\vec{v}_z, V.o)^\mathsf{T}$ denote this frame, where $(\vec{v}_x, \vec{v}_y, \vec{v}_z)^\mathsf{T}$ are the three unit vectors for the frame and o is the origin. Clearly $V.o = eye$. As mentioned earlier, the view vector $\overrightarrow{view}$ is directed from *eye* to *at*. The z-basis vector is the normalized negation of this vector.

$$\begin{aligned} \overrightarrow{view} &= \text{normalize}(at - eye) \\ V.\vec{v}_z &= -\overrightarrow{view} \end{aligned}$$

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

Next, we want to select the x-basis vector for our camera frame. It should be orthogonal to the viewing direction, it should be orthogonal to the up vector, and it should be directed to the camera's right. Recall that the cross product will produce a vector that is orthogonal to

any pair of vectors, and directed according to the right hand rule. Also, we want this vector to have unit length. Thus we choose

$$V.\vec{v}_x = \text{normalize}(\overrightarrow{view} \times \overrightarrow{up}).$$

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

$$V.\vec{v}_y = (V.\vec{v}_z \times V.\vec{v}_x).$$

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

Camera Transformation Matrix (Optional): Now, all we need to do is to construct the change of coordinates matrix from the standard world frame W to our camera frame V. We will not dwell on the linear algebra details, but the change of coordinate matrix is formed by considering the matrix M whose columns are the basis elements of V relative to W, and then *inverting*[2] this matrix. The matrix before inversion is:

$$M = \left((V.\vec{v}_x)_{[W]} \;\middle|\; (V.\vec{v}_y)_{[W]} \;\middle|\; (V.\vec{v}_z)_{[W]} \;\middle|\; (V.o)_{[W]} \right) = \begin{pmatrix} v_{xx} & v_{yx} & v_{zx} & o_x \\ v_{xy} & v_{yy} & v_{zy} & o_y \\ v_{xz} & v_{yz} & v_{zz} & o_z \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

OpenGL uses some tricks to compute the inverse, M^{-1}, efficiently. Normally, inverting a matrix would involve invoking a linear algebra procedure (e.g., based on Gauss elimination). However, because M is constructed from an orthonormal frame, there is a much easier way to construct the inverse. In particular, the upper 3×3 portion of the matrix can be inverted by taking its transpose. Let R be the linear part of matrix M, and let T be the negation of the translation part:

$$R = \begin{pmatrix} v_{xx} & v_{yx} & v_{zx} & 0 \\ v_{xy} & v_{yy} & v_{zy} & 0 \\ v_{xz} & v_{yz} & v_{zz} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \text{and} \quad T = \begin{pmatrix} 1 & 0 & 0 & -o_x \\ 0 & 1 & 0 & -o_y \\ 0 & 0 & 1 & -o_z \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

It can be shown that the final inverted matrix is given by the formula $M^{-1} = R^{\mathsf{T}} \cdot T$.

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

[2]The need for inverting matrices when doing change-of-coordinates transformations is always a bit mysterious. Here is an intuitive explanation of why we do it. Think of typical transformations as moving space around while the coordinate frame stays fixed. In contrast, when performing a change-of-coordinates transformation, we want to think of the objects in the space as staying fixed and the coordinate frame as moving. Thust, in order to achieve such a result, we figure out what transformation should be used to map the standard coordinate frame into the new frame, and then apply the inverse of this.

In spite of their superficial similarities, parallel and perspective projections behave quite differently with respect to geometry. Parallel projections are *affine transformations*, while perspective projections are not. (In particular, perspective projections do not preserve parallelism, as is evidenced by a perspective view of a pair of straight train tracks, which appear to converge at the horizon.) Because parallel projections are rarely used, we will skip them and consider perspective projections only.

Perspective Projection: Perspective transformations are the domain of an interesting area of mathematics called *projective geometry*. Let us assume that we are working in 3-dimensional space, and (through the use of the view transformation) we assume that objects are represented in camera coordinates. Projective transformations map lines to lines. However, projective transformations are not affine, since (except for the special case of parallel projection) do not preserve affine combinations and do not preserve parallelism. For example, consider the perspective projection T shown in Fig. 33. Let r be the midpoint of segment $\overline{pq}$. As seen in the figure, $T(r)$ is not necessarily the midpoint of $T(p)$ and $T(q)$.

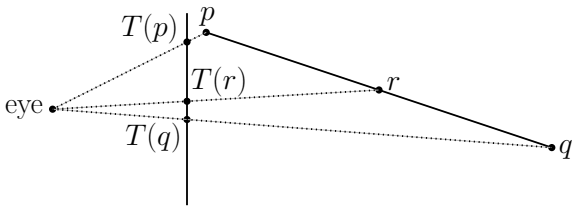


Fig. 33: Perspective transformations do not preserve affine combinations. The midpoint of $\overline{pq}$ does not map to the midpoint of $\overline{T(q)T(q)}$.

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

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

For example, in Fig. 34(a), the point p is a point at infinity. Since p is infinitely far away it does have a position (in the sense of affine space), but it can be specified by pointing to it, that is, by a direction. All lines that are parallel to one another along this direction intersect

at p. In the plane, the union of all the points at infinity forms a line, called the *line at infinity*. (In 3-space the corresponding entity is called the *plane at infinity*.) Note that every other line intersects the line at infinity exactly once. The regular affine plane together with the points and line at infinity define the *projective plane*. It is easy to generalize this to arbitrary dimensions as well.

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

Orientability and the Projective Space: Projective geometry seems to both generalize and simplify affine geometry, so why don't we just dispensed with affine geometry and use projective geometry instead? The reason is that, along with the good, come 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.)

One nice feature of the Euclidean planes is that each line partitions the plane into two halves, one above and one below (or left and right, if the line is vertical). This is not true for the projective plane (since each ideal point is both above and below and given line).

As another example of the strange things that occur in projective geometry, consider Fig. 34. Consider two ideal points p and q on the projective plane. We start with a standard clock. Imagine that we translate the clock through infinity, so that it passes between p and q, with the little hand pointing from p to q. When it wraps around, the little hand still points from p to q, but in order to achieve this (like a Möbius strip), the clock has flipped upside-down, thus changing its orientation. (If we were to follow the same course, it would flip back to its proper orientation on the second trip.) The implication is that there is no consistent way to define the concepts of clockwise and counterclockwise in projective geometry.

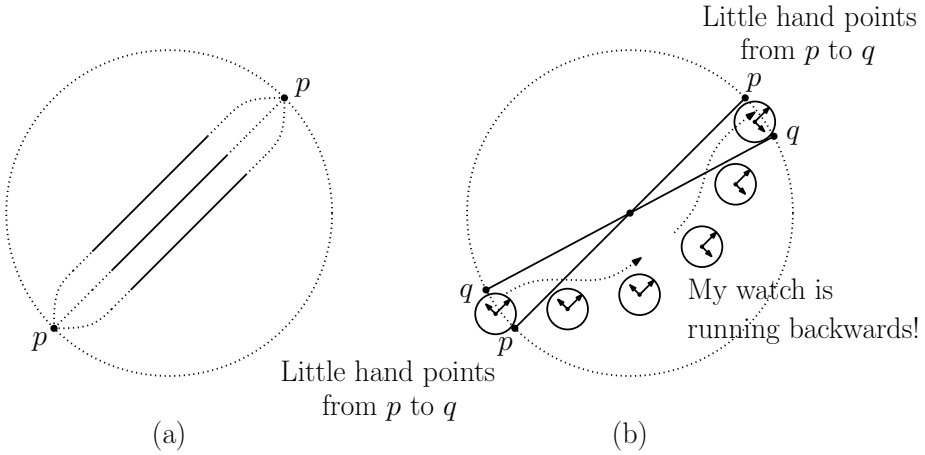


Fig. 34: The wacky world of projective geometry. The projective plane behaves much like Möbius strip. As you wrap around through infinity, there is a twist, which flips orientations.

In topological terms, we say that the projective plane is a *nonorientable manifold*. In contrast,

the Euclidean plane and the sphere are both orientable surfaces.

For these reasons, we choose not to use projective space as a domain in which to do most of our geometric computations. Instead, we will do almost all of our geometrical computations in the affine plane. We will briefly enter the domain of projective geometry to do our projective transformations. We will have to take care that when object map to 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 with the homogeneous coordinates that we introduced with affine geometry. First off, we will *not* deal with free vectors in projective space, just points. Consider a regular point p in the plane, with standard (nonhomogeneous) coordinates $(x, y)^\mathsf{T}$. There will not be a unique representation for this point in projective space. Rather, it will be represented by any coordinate vector of the form:

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

Thus, if $p = (4, 3)^\mathsf{T}$ are p's standard Cartesian coordinates, the homogeneous coordinates $(4, 3, 1)^\mathsf{T}$, $(8, 6, 2)^\mathsf{T}$, and $(-12, -9, -3)^\mathsf{T}$ are all legal representations of p in projective plane. Because of its familiarity, we will use the case $w = 1$ most often.

Given the homogeneous coordinates of a regular point $p = (x, y, w)^\mathsf{T}$, the *projective normalization* of p is the coordinate vector $(x/w, y/w, 1)^\mathsf{T}$. (This term is confusing, because it is quite different from the process of *length normalization*, which maps a vector to one of unit length. In computer graphics this operation is also referred as *perspective division* or *perspective normalization*.)

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

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

Clearly these are equivalent to the following

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

(This is illustrated in Fig. 35.) We can see that as x tends to infinity, the limiting point has the homogeneous coordinates $(1, 2, 0)^\mathsf{T}$. So, when $w = 0$, the point $(x, y, w)^\mathsf{T}$ is the point at infinity, that is pointed to by the vector $(x, y)^\mathsf{T}$ (and $(-x, -y)^\mathsf{T}$ as well by wraparound).

Important Note: In spite of the similarity of the names, homogeneous coordinates in projective geometry and homogeneous coordinates in affine are *entirely different* concepts, and should not be mixed. This is because the two geometric systems are entirely different.

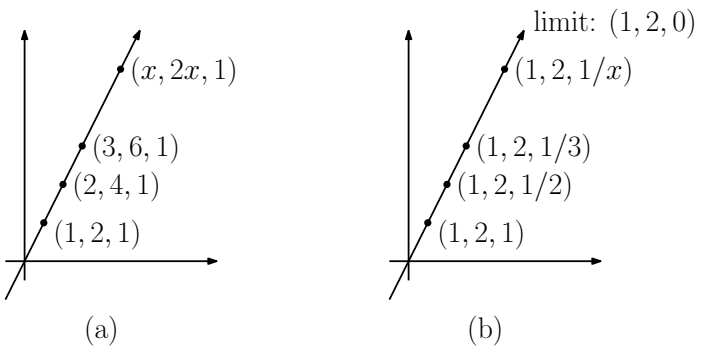


Fig. 35: Homogeneous coordinates for ideal points.

Lecture 10: More on 3-d Viewing and Projections

Perspective Projection Transformations: We shall see today that is possible to define a general perspective projection using a 4×4 matrix, just as we did with affine transformations. However, due to the differences in projective homogeneous coordinates, we will need treat projective transformations somewhat differently. We assume that we will be transforming points only, not vectors. (Typically we will be transforming the vertices of geometric objects, e.g., as presented by calls to glVertex.) 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. We assume that the center of projection is located at the origin of the coordinate frame. This is normally the camera coordinate frame, as generated by gluLookAt. The viewer is facing the $-z$ direction. (Recall that this needed so that the coordinate frame is right-handed.) The x-axis points to the viewer's right and the y-axis points upwards relative to the viewer (see Fig. 36(a)).

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.) Since it is hard to draw good perspective drawings in 3-space, we will take a side view and consider just the y and z axes for now (see Fig. 36(b)). Everything we do with y we will do symmetrically with x later.

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

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

implying that $y' = y/(-z/d)$ (see Fig. 36(b)). We also have $z = -d$. Generalizing this to 3-space, the point with coordinates $(x, y, z, 1)^\mathsf{T}$ is transformed to the point with homogeneous

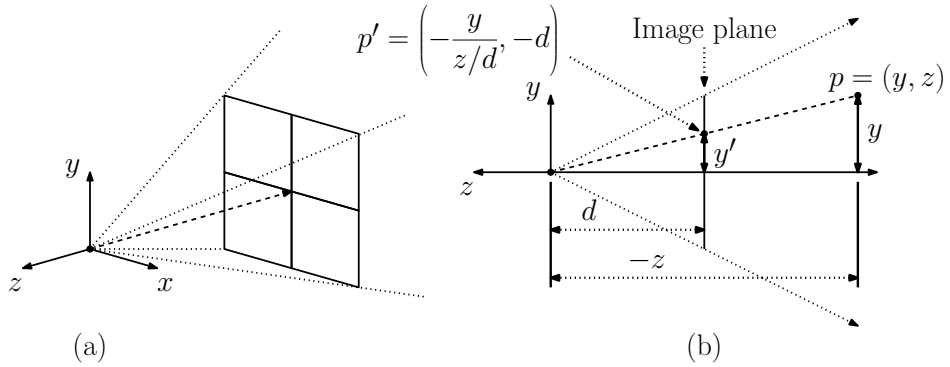


Fig. 36: Perspective transformation. (On the right, imagine that the x-axis is pointing towards you.)

coordinates

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

Unfortunately, there is no 4×4 matrix that can realize this result. (Generally a matrix will map a point $(x, y, z, 1)^\mathsf{T}$ to a point whose coordinates are of the form $ax + by + cz + d$. The problem here is that z is in the denominator.)

However, there is a 4×4 matrix that will generate an *equivalent* point, with respect to homogeneous coordinates. In particular, if we multiply the above vector by $(-z/d)$ we obtain:

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

The coordinates of this vector are all linear function of x, y, and z, and so we can write the perspective transformation in terms of the following matrix.

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

After we have the coordinates of a (affine) transformed point $p' = M \cdot p$, we then apply projective normalization (perspective division) to determine the corresponding point in Euclidean space.

$$Mp = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/d & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ -z/d \end{pmatrix} \equiv \begin{pmatrix} x/(-z/d) \\ y/(-z/d) \\ -d \\ 1 \end{pmatrix}.$$

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.

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

Consider the following viewing model. In front of his eye, the user holds rectangular window, centered on the view direction, onto which the image is to be projected. The viewer sees any object that lies within a rectangular pyramid, whose axis is the $-z$-axis, and whose apex is his eye. In order to indicate the height of this pyramid, the user specifies its angular height, called the *y field-of-view* and denoted *fovy* (see Fig. 37). It is given in degrees.

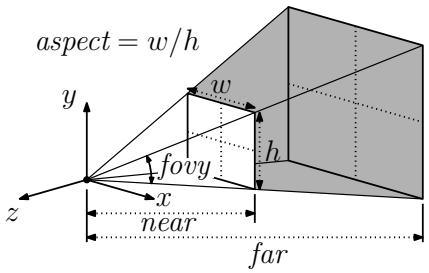


Fig. 37: OpenGL's perspective specification.

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

Finally, for technical reasons related to depth buffering, we need to specify a distance along the $-z$-axis to the *near clipping plane* and to the *far clipping plane*. Objects in front of the near plane and behind the far plane will be clipped away. We have a limited number of bits of depth-precision, and supporting a greater range of depth values will limit the accuracy with which we can represent depths. The resulting shape is called the *viewing frustum*. (A *frustum* is the geometric shape that arises from chopping off the top of a pyramid. An example appears on the back of the US one dollar bill.) These arguments form the basic elements of the main OpenGL command for perspective.

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

All arguments are positive and of type GLdouble. This command creates a matrix which performs the necessary depth perspective transformation, and multiplies it with the matrix on top of the current stack. This transformation should be applied to the projection matrix

stack. So this typically occurs in the following context of calls, usually as part of your initializations.

```
void myDisplay() {
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glLoadIdentity();
  gluLookAt( ... );                          // set up camera frame
  glMatrixMode(GL_PROJECTION);               // set up projection
  glLoadIdentity();
  gluPerspective(fovy, aspect, near, far);   // or glFrustum(...)
  glMatrixMode(GL_MODELVIEW);
  myWorld.draw();                            // draw everything
  glutSwapBuffers();
}
```

The function gluPerspective does not have to be called again unless the camera's projection properties are changed (e.g., increasing or decreasing zoom). For example, it does not need to be called if the camera is simply moved to a new location.

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

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

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

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

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

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

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

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

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

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

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

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

The canonical view volume (after perspective division) is just a 3-dimensional rectangle. It is defined by the following constraints (see Fig. 38(b)):

$$-1 \le x \le +1, \qquad -1 \le y \le +1, \qquad -1 \le z \le +1.$$

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

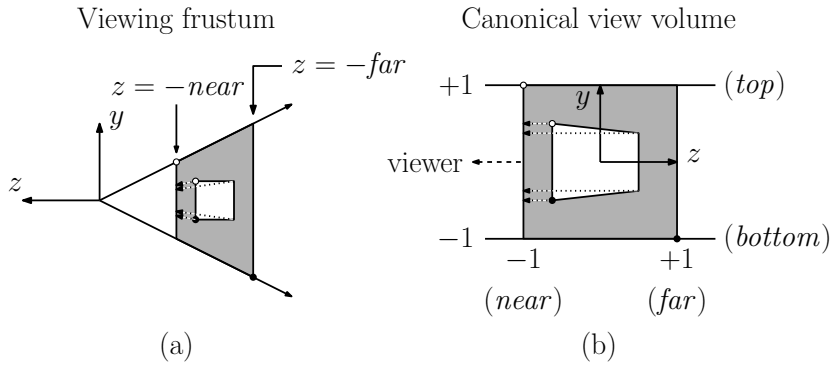


Fig. 38: Perspective with depth and the canonical view volume.

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

$$-1 \;=\; -\alpha - \frac{\beta}{-n} \qquad \text{and} \qquad +1 \;=\; -\alpha - \frac{\beta}{-f}.$$

Solving for α and β yields

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

Perspective Matrix (Optional): To see how OpenGL handles this process, recall the function gluPerspective. Let θ denote the y field of view (*fovy*) in radians. Let $c = \cot(\theta/2)$. We will take a side view as usual (imagine that the x-coordinate is directed out of the page). Let a denote the aspect ratio, let n denote the distance to the near clipping plane and let f denote the distance to the far clipping plane. (All quantities are positive.) Here is the matrix it constructs to perform the perspective transformation.

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

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

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

How did we come up with such a strange mapping? Notice that other than the scaling factors, this is very similar to the perspective-with-depth matrix given earlier (given our values α and

β plugged in). The diagonal entries c/a and c are present to scale the arbitrarily shaped window into the square (as we'll see below).

To see that this works, we will show that the corners of the viewing frustum are mapped to the corners of the canonical viewing volume (and we'll trust that everything in between behaves nicely). In Fig. 38 we show a side view, thus ignoring the x-coordinate. Because the window has the aspect ratio $a = w/h$, it follows that for points on the upper-right edge of the viewing frustum (relative to the viewer's perspective) we have $x/y = a$, and thus $x = ay$.

Consider a point that lies on the top side of the view frustum. We have $-z/y = \cot\theta/2 = c$, implying that $y = -z/c$. If we take the point to lie on the near clipping plane, then we have $z = -n$, and hence $y = n/c$. Further, if we assume that it lies on the upper right corner of the frustum (relative to the viewer's position) then $x = ay = an/c$. Thus the homogeneous coordinates of the upper corner on the near clipping plane (shown as a white dot in Fig. 38) are $(an/c, n/c, -n, 1)^\mathsf{T}$. If we apply the above transformation, this is mapped to

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

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

Similarly, consider a point that lies on the bottom side of the view frustum. We have $-z/(-y) = \cot\theta/2 = c$, implying that $y = z/c$. If we take the point to lie on the far clipping plane, then we have $z = -f$, and so $y = -f/c$. Further, if we assume that it lies on the lower left corner of the frustum (relative to the viewer's position) then $x = -af/c$. Thus the homogeneous coordinates of the lower corner on the far clipping plane (shown as a black dot in Fig. 38) are $(-af/c, -f/c, -f, 1)^\mathsf{T}$. If we apply the above transformation, this is mapped to

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

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

Lecture 11: Lighting and Shading

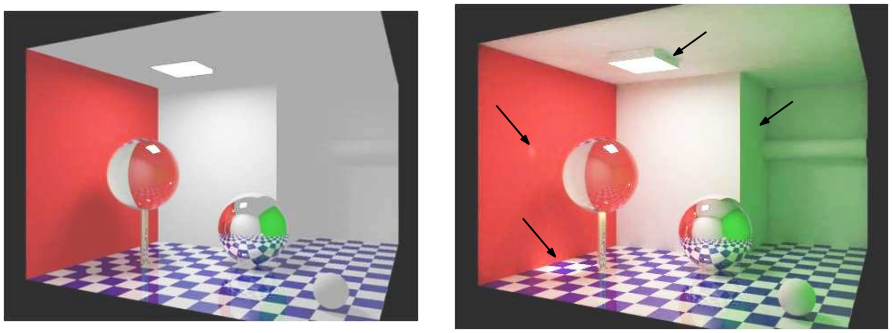
Lighting and Shading: We will now take a look at the next major element of graphics rendering: light and shading. This is one of the primary elements of generating realistic images. This topic is the beginning of an important shift in approach. Up until now, we have discussed graphics from are purely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

What we "see" is a function of the light that enters our eye. Light sources generate energy, which we may think of as being composed of extremely tiny packets of energy, called *photons*. The photons are reflected and transmitted in various ways throughout the environment. They bounce off various surfaces and may be scattered by smoke or dust in the air. Eventually, some of them enter our eye and strike our retina. We perceive the resulting amalgamation of photons of various energy levels in terms of *color*. The more accurately we can simulate this physical process, the more realistic lighting will be. Unfortunately, computers are not fast enough to produce a truly realistic simulation of indirect reflections in real time, and so we will have to settle for much simpler approximations.

OpenGL's Fixed-Function Lighting Model: Throughout this lecture we will discuss a lighting model that is part of OpenGL's older (and now deprecated) fixed-function pipeline. Because of the high complexity of computing highly realistic lighting, OpenGL supports a very simple lighting and shading model, and hence can achieve only limited realism. This was done primarily for the sake of efficiency and scalability. Modern GPUs achieve speed through parallel processing. For this to work, however, it should be that the illumination of every pixel can be computed based on a small amount of information that can be stored in a single processing element.

To make this feasible, (old) OpenGL assumes a *local illumination model*, which means that the shading of a point depends *only* on its relationship to a small number of light sources, without considering interactions with the many other objects of the scene. Note that modern versions of OpenGL have the programmer explicitly control the shading of elements through the use of *programmable shaders* (which we will discuss later in the semester). While this enhances the programmer's control over the shading process, it is still hard to achieve efficiency without violating the assumption of locality.

Global Illumination Model: This is in contrast to a *global illumination model*, in which light reflected or passing through one object might affects the illumination of other objects. Global illumination models deal with many affects, such as shadows, indirect illumination, color bleeding (colors from one object reflecting and altering the color of a nearby object), caustics (which result when light passes through a lens and is focused on another surface). An example of some of the differences between a local and global illumination model are shown in Fig. 39.



Local Illumination Model Global Illumination Model

Fig. 39: Local versus global illumination models.

For example, OpenGL's fixed-function lighting model does not model shadows, it does not handle indirect reflection from other objects (where light bounces off of one object and illuminates another), it does not handle objects that reflect or refract light (like metal spheres and glass balls). OpenGL's fixed-function light and shading model was designed to be very efficient, but not highly realistic. Nonetheless, there are many clever tricks that can be used to "fake" realism (which we will also discuss later in the semester).

Light: A detailed discussion of light and its properties would take us more deeply into physics than we care to go. For our purposes, we can imagine a simple model of light consisting of a large number of photons being emitted continuously from each light source. Each photon has an associated energy, which (when aggregated over millions of different reflected photons) we perceive as *color*. Although color is complex phenomenon, for our purposes it is sufficient to consider color to be a modeled as a triple of red, green, and blue components. (We will consider color later this semester.)

The strength or *intensity* of the light at any location can be modeled in terms of the *flux*, that is, the amount of illumination energy passing through a fixed area over a fixed amount of time. Assuming that the atmosphere is a vacuum (in particular there is no smoke or fog), a photon of light travels unimpeded until hitting a surface, after which one of three things can happen (see Fig. 40):

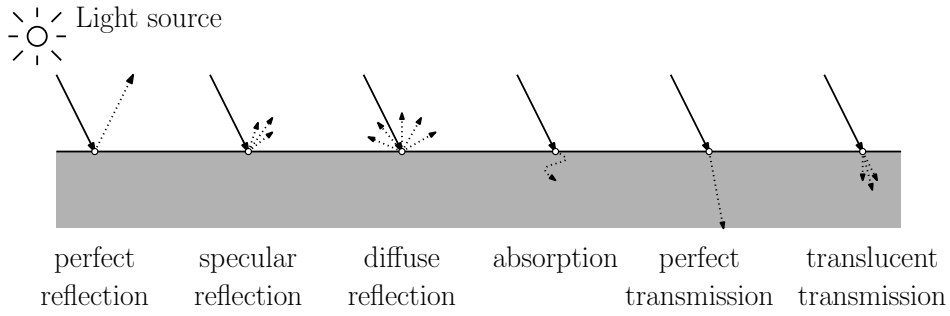


Fig. 40: The ways in which a photon of light can interact with a surface.

Reflection: The photon can be reflected or scattered back into the atmosphere (or whatever the transmission medium is). If the surface were perfectly smooth (like a mirror or highly polished metal) the refection would satisfy the rule "angle of incidence equals angle of reflection" and the result would be a mirror-like and very shiny in appearance. On the other hand, if the surface is rough at a microscopic level (like foam rubber, say) then the photons are scattered nearly uniformly in all directions. We can further distinguish different varieties of reflection:

Pure reflection: Perfect mirror-like reflectors

Specular reflection: Imperfect reflectors like brushed metal and shiny plastics.

Diffuse reflection: Uniformly scattering, and hence not shiny.

Absorption: The photon can be absorbed into the surface (and hence dissipates in the form of heat energy). We do not see this light. Thus, an object appears to be green, because it reflects photons in the green part of the spectrum and absorbs photons in the other regions of the visible spectrum.

Transmission: The photon can pass through the surface. This happens perfectly with *transparent* objects (like glass and polished gem stones) and with a significant amount of scattering with *translucent* objects (like human skin or a thin piece of tissue paper).

All of the above involve how incident light reacts with a surface. Another way that light may result from a surface is through emission, which will be discussed below.

Of course, real surfaces possess various combinations of these elements, and these element can interact in complex ways. For example, human skin and many plastics are characterized by a complex phenomenon called *subsurface scattering*, in which light is transmitted under the surface and then bounces around and is reflected at some other point.

Light Sources: Before talking about light reflection, we need to discuss where the light originates. In reality, light sources come in many sizes and shapes. They may emit light in varying intensities and wavelengths according to direction. The intensity of light energy is distributed across a continuous spectrum of wavelengths.

To simplify things, the OpenGL fixed-function model assumes that each light source is a *point*, and that the energy emitted can be modeled as an RGB triple, called a *luminance function*. This is described by a vector with three components $L = (L_r, L_g, L_b)$, which indicate the intensities of red, green, and blue light respectively. We will not concern ourselves with the exact units of measurement, since this is very simple model. Note that, although your display device will have an absolute upper limit on how much energy each color component of each pixel can generate (which is typically modeled as an 8-bit value in the range from 0 to 255), in theory there is no upper limit on the intensity of light. (If you need evidence of this, go outside and stare at the sun for a while!)

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

Ambient emission: Refers to light that does not come from any particular location. Like heat, it is assumed to be scattered uniformly in all locations and directions. A point is illuminated by ambient emission even if it is not visible from the light source.

Point emission: Refers to light that originates from a single point. In theory, point emission only affects points that are directly visible to the light source. That is, a point p is illuminate by light source q if and only if the open line segment $\overline{pq}$ does not intersect any of the objects of the scene.

Unfortunately, determining whether a point is visible to a light source in a complex scene with thousands of objects can be computationally quite expensive. So OpenGL simply tests whether the surface is facing towards the light or away from the light. Surfaces in OpenGL are polygons, but let us consider this in a more general setting. Suppose that have a point p lying on some surface. Let n denote the normal vector at p, directed *outwards* from the

object's interior, and let ℓ denote the directional vector from p to the light source ($\ell = q - p$), then p will be illuminated if and only if the angle between these vectors is acute. We can determine this by testing whether their dot produce is positive, that is, $n \cdot \ell > 0$.

For example, in the Fig. 41, the point p is illuminated. In spite of the obscuring triangle, point p' is also illuminated, because other objects in the scene are ignored by the local illumination model. The point p'' is clearly not illuminated, because its normal is directed away from the light.

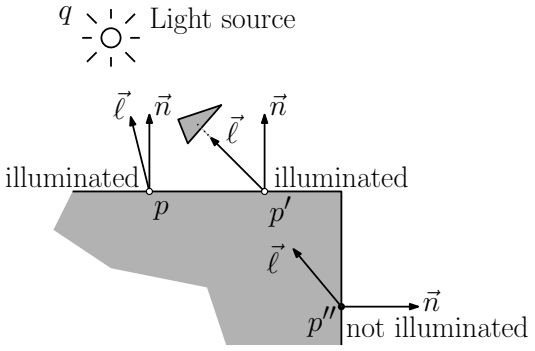


Fig. 41: Point light source visibility using a local illumination model. Note that p' is illuminated in spite of the obscuring triangle.

Attenuation: The light that is emitted from a point source is subject to *attenuation*, that is, the decrease in strength of illumination as the distance to the source increases. Physics tells us that the intensity of light from a point source falls off as the inverse square of the distance. This would imply that the intensity at some (unblocked) point p would be

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

where $\|p - q\|$ denotes the Euclidean distance from p to q. However, in OpenGL, our various simplifying assumptions (ignoring indirect reflections, for example) will cause point sources to appear unnaturally dim using such a pure model of attenuation. Consequently, OpenGL provides the user a more general attenuation function that allows the user to control three terms, one is a constant and the other two grow linearly and quadratically with the distance from the light source. In particular, the user specifies constants a, b and c. Let $d = \|p - q\|$ denote the distance to the point source. Then the attenuation function is

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

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

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

$$(0, 0, 1, 0)^{\mathsf{T}}.$$

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

Sometimes it is nice to have a directional component to the light sources. OpenGL also supports something called a *spotlight*, where the intensity is strongest along a given direction, and then drops off according to the angle from this direction (see Fig. 42). See the OpenGL function glLight() for further information.

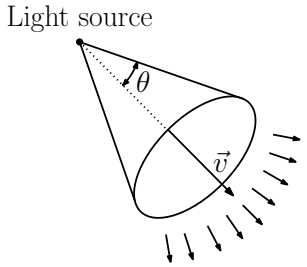


Fig. 42: Spotlight. The intensity decreases as the angle θ increases.

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

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

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

Diffuse reflection: The illumination produced by matte (i.e, dull or nonshiny) smooth objects, such as foam rubber.

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

Let $L = (L_r, L_g, L_b)$ denote the illumination intensity of the light source. OpenGL allows us to break this light's emitted intensity into three components: *ambient* L_a, *diffuse* L_d, and *specular* L_s. Each type of light component consists of the three color components, so, for example, $L_d = (L_{dr}, L_{dg}, L_{db})$, denotes the RGB vector (or more generally the RGBA

components) of the diffuse component of light. As we have seen, modeling the ambient component separately is merely a convenience for modeling indirect reflection. It is not as clear why someone would want to turn on or turn off a light source's ability to generate diffuse and specular reflection. (There is no physical justification to this that I know of. It is an object's surface properties, not the light's properties, which determine whether light reflects diffusely or specularly. But, again this is all just a model.) The diffuse and specular intensities of a light source are usually set equal to each other.

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

$$L \otimes C = (L_r C_r, L_g C_g, L_b C_b).$$

For example, if the light is white $L = (2, 1, 1)$ and the color is red $C = (0.25, 0, 0)$ then the reflection is $L \otimes C = (0.5, 0, 0)$ which is a dark shade of red. However if the light is blue $L = (0, 0, 1)$, then $L \otimes C = (0, 0, 0)$, and hence the object appears to be black.

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

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

Lecture 12: More on Lighting and Shading

Lighting and Shading: Last time we discussed the basic elements of the Phong lighting model. Let us consider how the various components (ambient, diffuse, and specular) are computed.

The Relevant Vectors: The shading of a point on a surface is a function of the relationship between the viewer, light sources, and surface. (Recall that because this is a local illumination model the other objects of the scene are ignored.) The following vectors are relevant to shading. We can think of them as being centered on the point whose shading we wish to compute. For the purposes of our equations below, it will be convenient to think of them all as being of unit length (see Fig. 43(a)).

Normal vector: A vector $\vec{n}$ that is perpendicular to the surface and directed outwards from the surface. There are a number of ways to compute normal vectors, depending on the

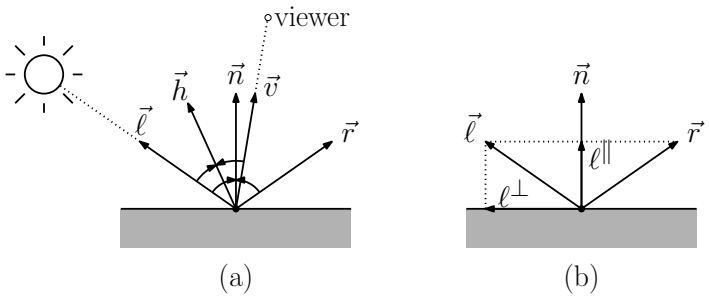


Fig. 43: Vectors used in Phong Shading.

representation of the underlying object. For our purposes, the following simple method is sufficient. Given any three noncollinear points, p_0, p_1, and p_2, on a polygon, we can compute a normal to the surface of the polygon as a cross product of two of the associated vectors.

$$\vec{n} = \mathit{normalize}((p_1 - p_0) \times (p_2 - p_0)).$$

The vector will be directed outwards if the triple $\langle p_0, p_1, p_2 \rangle$ has a counterclockwise orientation when seen from outside.

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

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

Reflection vector: A vector $\vec{r}$ that indicates the direction of pure reflection of the light vector. (Based on the law that the angle of incidence with respect to the surface normal equals the angle of reflection.) The reflection vector computation reduces to an easy exercise in vector arithmetic. First, let us decompose $\vec{\ell}$ into two parts, one parallel to $\vec{n}$ and one orthogonal to $\vec{n}$. Since $\vec{n}$ is of unit length (and recalling the properties of the dot product) we have

$$\vec{\ell} \;=\; \ell^{\parallel} + \ell^{\perp} \qquad \text{where} \qquad \ell^{\parallel} \;=\; \frac{(\vec{n} \cdot \vec{\ell})}{(\vec{n} \cdot \vec{n})}\vec{n} \;=\; (\vec{n} \cdot \vec{\ell})\vec{n} \quad \text{and} \quad \ell^{\perp} = \vec{\ell} - \ell^{\parallel}.$$

To get $\vec{r}$ observe that we need add two copies of $-\ell^{\perp}$ to $\vec{\ell}$. Thus we have

$$\vec{r} \;=\; \vec{\ell} - 2\ell^{\perp} \;=\; \vec{\ell} - 2(\vec{\ell} - \ell^{\parallel}) \;=\; 2(\vec{n} \cdot \vec{\ell})\vec{n} - \vec{\ell}.$$

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

$$\vec{h} \;=\; \mathit{normalize}\left(\frac{\vec{\ell} + \vec{v}}{2}\right) \;=\; \mathit{normalize}(\vec{\ell} + \vec{v}).$$

Phong Lighting Equations: There almost no objects that are pure diffuse reflectors or pure specular reflectors. The Phong reflection model is based on the simple modeling assumption

that we can model any (nontextured) object's surface to a reasonable extent as some mixture of purely diffuse and purely specular components of reflection along with emission and ambient reflection. Let us ignore emission for now, since it is the rarest of the group, and will be easy to add in at the end of the process.

The surface material properties of each object will be specified by a number of parameters, indicating the intrinsic color of the object and its ambient, diffuse, and specular reflectance. Let C denote the RGB factors of the object's base color. For consistency with OpenGL's convention, we assume that the light's energy is given by three RGB vectors: its ambient intensity L_a, its diffuse intensity L_d, and its specular intensity, L_s. (In any realistic physical model, these three are all equal to each other).

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

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

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

$$I_a = \rho_a L_a C$$

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

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

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

It is easy to see from the Fig. 44 that as the angle θ between the surface normal $\vec{n}$ and the vector to the light source $\vec{\ell}$ increases (up to a maximum of 90 degrees) then amount of light intensity hitting a small differential area of the surface dA is proportional to the area of the perpendicular cross-section of the light beam, $dA \cos\theta$. The is called *Lambert's Cosine Law*.

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

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

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

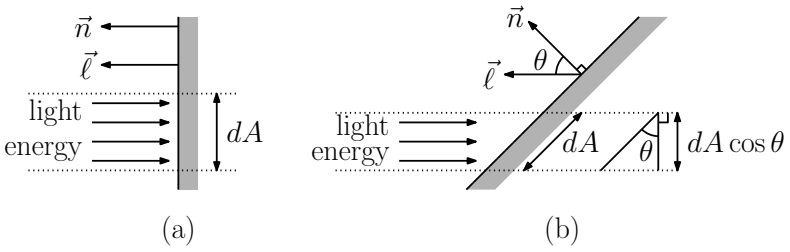


Fig. 44: Lambert's Cosine Law.

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

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

The parameters of surface finish that control specular reflection are ρ_s, the surface's *coefficient of specular reflection*, and *shininess*, denoted α (see Fig. 45). As α increases, the specular reflection drops off more quickly, and hence the size of the resulting shiny spot on the surface appears smaller as well. Shininess values range from 1 for low specular reflection up to, say, 1000, for highly specular reflection. The formula for the specular component is

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

As with diffuse, this is subject to attenuation.

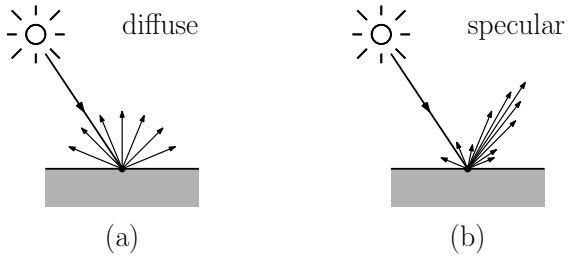


Fig. 45: Diffuse and specular reflection.

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

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

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

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

Lighting/Shading model: There are a number of global lighting parameters and options that can be set through the command glLightModel*(). It has two forms, one for scalar-valued parameters and one for vector-valued parameters.

```
glLightModelf(GLenum pname, GLfloat param);
glLightModelfv(GLenum pname, const GLfloat* params);
```

Perhaps the most important parameter is the global intensity of ambient light (independent of any light sources). Its pname is GL_LIGHT_MODEL_AMBIENT and params is a pointer to an RGBA vector.

One important issue is whether polygons are to be drawn using *flat shading*, in which every point in the polygon has the same shading, or *smooth shading*, in which shading varies across the surface by interpolating the vertex shading. This is set by the following command, whose argument is either GL_SMOOTH (the default) or GL_FLAT.

```
glShadeModel(GL_SMOOTH);     --OR--     glShadeModel(GL_FLAT);
```

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

The commands glLightModel and glShadeModel are usually invoked in your initializations.

Create/Enable lights: To use lighting in OpenGL, first you must enable lighting, through a call to glEnable(GL_LIGHTING). OpenGL allows the user to create up to 8 light sources, named

GL_LIGHT0 through GL_LIGHT7. Each light source may either be enabled (turned on) or disabled (turned off). By default they are all disabled. Again, this is done using glEnable() (and glDisable()). The properties of each light source is set by the command glLight*(). This command takes three arguments, the name of the light, the property of the light to set, and the value of this property.

Let us consider a light source 0, whose position is $(2, 4, 5, 1)^\mathsf{T}$ in homogeneous coordinates, and which has a red ambient intensity, given as the RGB triple $(0.9, 0, 0)$, and white diffuse and specular intensities, given as the RGB triple $(1.2, 1.2, 1.2)$. (Normally all the intensities will be of the same color, albeit of different strengths. We have made them different just to emphasize that it is possible.) There are no real units of measurement involved here. Usually the values are adjusted manually by a designer until the image "looks good."

Light intensities are actually expressed in OpenGL as RGBA, rather than just RGB triples. The 'A' component can be used for various special effects, but for now, let us just assume the default situation by setting 'A' to 1. Here is an example of how to set up such a light in OpenGL. The procedure glLight*() can also be used for setting other light properties, such as attenuation.

Setting up a simple lighting situation

```
glClearColor(0.0, 1.0, 0.0, 1.0);           // intentionally background
glEnable(GL_NORMALIZE);                     // normalize normal vectors
glShadeModel(GL_SMOOTH);                    // do smooth shading
glEnable(GL_LIGHTING);                      // enable lighting
                                            // ambient light (red)
GLfloat ambientIntensity[4] = {0.9, 0.0, 0.0, 1.0};
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambientIntensity);

                                            // set up light 0 properties
GLfloat lt0Intensity[4] = {1.5, 1.5, 1.5, 1.0};  // white
glLightfv(GL_LIGHT0, GL_DIFFUSE, lt0Intensity);
glLightfv(GL_LIGHT0, GL_SPECULAR, lt0Intensity);

GLfloat lt0Position[4] = {2.0, 4.0, 5.0, 1.0};   // location
glLightfv(GL_LIGHT0, GL_POSITION, lt0Position);
                                           // attenuation params (a,b,c)
glLightf (GL_LIGHT0, GL_CONSTANT_ATTENUATION,  0.0);
glLightf (GL_LIGHT0, GL_LINEAR_ATTENUATION,    0.0);
glLightf (GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.1);
glEnable(GL_LIGHT0);
```

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

Every object in OpenGL is a polygon, and in general every face can be colored in two different

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

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

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

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

Typical drawing with lighting

```
GLfloat color[]  = {0.0, 0.0, 1.0, 1.0};    // blue
GLfloat white[]  = {1.0, 1.0, 1.0, 1.0};    // white
                                            // set object colors
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, color);
glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, white);
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, 100);

glPushMatrix();
    glTranslatef(...);                      // your transformations
    glRotatef(...);
    glBegin(GL_POLYGON);                    // draw your shape
        glNormal3f(...); glVertex(...);     // remember to add normals
        glNormal3f(...); glVertex(...);
        glNormal3f(...); glVertex(...);
    glEnd();
glPopMatrix();
```

Recall from the Phong model that each surface is associated with a single color and various coefficients are provided to determine the strength of each type of reflection: emission, ambient, diffuse, and specular. In OpenGL, these two elements are combined into a single vector given as an RGB or RGBA value. For example, in the traditional Phong model, a red object might have a RGB color of $(1, 0, 0)$ and a diffuse coefficient of 0.5. In OpenGL, you would just set the diffuse material to $(0.5, 0, 0)$. This allows objects to reflect different colors of ambient and diffuse light (although I know of no physical situation in which this arises).

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

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

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

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

GL_NORMALIZE: Recall that normal vectors are used in shading computations. You supply these normal to OpenGL. These are assumed to be normalized to unit length in the Phong model. Enabling this option causes all normal vectors to be normalized to unit length automatically. If you know that your normal vectors are of unit length, then you will not need this. It is provided as a convenience, to save you from having to do this extra work.

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

Normals by Cross Product: Given three (nocollinear) points on a polygonal surface, p_0, p_1, and p_2, we can compute a normal vector by forming the two vectors and taking their cross product.

$$\vec{u}_1 = p_1 - p_0 \qquad \vec{u}_2 = p_2 - p_0 \qquad \vec{n} = \text{normalize}(\vec{u}_1 \times \vec{u}_2).$$

This will be directed to the side from which the points appear in counterclockwise order.

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 more robust method is to consider all the points on the polygon. Suppose we are given a planar polygonal patch, defined by a sequence of m points $p_0, p_1, \ldots, p_{m-1}$. We assume that these points define the vertices of a polygonal patch.

Here is a nice method for determining the plane equation,

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

Given the plane equation, the normal vector has the coordinates $\vec{n} = (a, b, c)^{\mathsf{T}}$, which can be normalized to unit length.

This leaves the question of to compute a, b, and c? An interesting 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. So how do we compute the projected area of a polygon? Let us consider the xy-projection for concreteness. The formula is:

$$c = \frac{1}{2}\sum_{i=1}^{m}(y_i + y_{i+1})(x_i - x_{i+1}).$$

But where did this formula come from? The idea is to break the polygon's area into the sum of signed trapezoid areas (see Fig. 46).

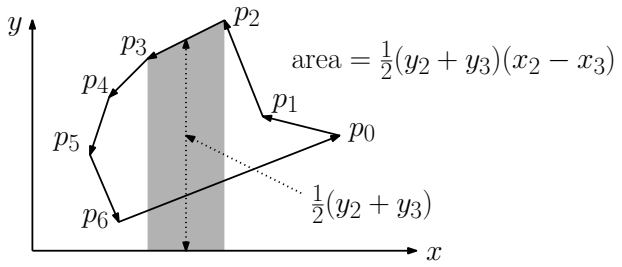


Fig. 46: Area of polygon.

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. (Recall that this is the product of the length of the base times the average height.) 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.

The final computation of the projected areas is, therefore:

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

Normals for Implicit 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*, denoted ∇. This is a vector whose components are the partial derivatives of the function at this point

$$\vec{n} = \text{normalize}(\nabla) \qquad \nabla = \begin{pmatrix} \partial f/\partial x \\ \partial f/\partial y \\ \partial f/\partial z \end{pmatrix}.$$

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.)

For example, consider a bowl shaped *paraboloid*, defined by the equal $x^2 + y + 2 = z$. The corresponding implicit equation is $f(x, y, z) = x^2 + y^2 - z = 0$. The gradient vector is

$$\nabla(x, y, z) = \begin{pmatrix} 2x \\ 2y \\ -1 \end{pmatrix}.$$

Consider a point $(1, 2, 5)^\mathsf{T}$ on the surface of the paraboloid. The normal vector at this point is $\nabla(1, 2, 5) = (2, 4, -1)^\mathsf{T}$.

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 $\vec{u}$ and $\vec{v}$:

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

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 $(\phi_x(\vec{u}, \vec{v}), \phi_y(\vec{u}, \vec{v}), \phi_z(\vec{u}, \vec{v}))$ on the surface.

To compute a normal vector, first compute the gradients with respect to $\vec{u}$ and $\vec{v}$,

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

and then return their cross product

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

Lecture 13: Texture Mapping

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

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

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

Rather than thinking of the image as being stored in an array, it will be a little more elegant to think of the image as function that maps a point (s, t) in 2-dimensional *texture space* to an RGB value. That is, given any pair (s, t), $0 \le s, t < 1$, the texture image defines the value of $T(s, t)$ is an RGB value. Note that the interval $[0, 1)$ does not depend on the size of the images. This has the advantage an image of a different size can be substituted without the need of modifying the wrapping process.

For example, if we assume that our image array $I[n][n]$ is indexed by row and column from 0 to $n-1$ with (as is common with images) the origin in the upper left corner. Our texture space $T(s, t)$ is coordinatized with axes s (horizontal) and t (vertical) where (following OpenGL's conventions) the origin is in the lower left corner. We could then apply the following function to round a point in image space to the corresponding array element:

$$T(s,t) = I\big[\,\lfloor (1-t)n \rfloor\,\big]\big[\,\lfloor sn \rfloor\,\big], \qquad \text{for } 0 \le s, t < 1$$

(see Fig. 47).

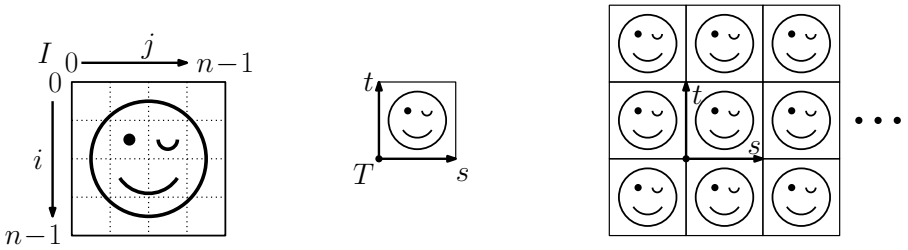


Fig. 47: Texture space.

In many cases, it is convenient to think of the texture is an infinite function. We do this by imagining that the texture image is repeated cyclically throughout the plane. (This handy when applying a small texture, such as a patch of grass, to a very large surface, like the

surface of a soccer field.) This is sometimes called a *repeated texture*. In this case we can modify the above function to be

$$T(s,t) = I\big[\,\lfloor (1-t)n \rfloor \bmod n\big]\big[\,\lfloor sn \rfloor \bmod n\big], \qquad \text{for any } s,t.$$

Inverse Wrapping Function and Parameterizations: Suppose that we wish to "wrap" a 2-dimensional texture image onto the surface of a 3-dimensional ball of unit radius, that is, a unit sphere. We need to define a wrapping function that achieves this. The surface resides in 3-dimensional space, so the wrapping function would need to map a point (s,t) in texture space to the corresponding point (x,y,z) in 3-space. That is, the wrapping function can be thought of as a function $W(s,t)$ that maps a point in 2-dimensional texture space to a point (x,y,z) in three dimensional space.

Later we will see that it is not the wrapping function that we need to compute, but rather its inverse. So, let us instead consider the problem of computing a function W^{-1} that maps a point (x,y,z) on the sphere to a point (s,t) in parameter space. This is called the *inverse wrapping function*.

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

Our approach to solving the inverse wrapping problem will be to map a point (x,y,z) to the corresponding parameter pair (u,v), and then map this parameter pair to the desired point (s,t) in texture space.

Example—Parameterizing a Sphere: Let's make this more concrete with an example. Our shape is a unit sphere centered at the origin. We want to find the inverse wraping function W^{-1} that maps any point (x,y,z) on the of the sphere to a point (s,t) in texture space.

We first need to come up with a surface parameterization for the sphere. We can represent any point on the sphere with two angles, representing the point's latitude and longitude. We will use a slightly different approach. Any point on the sphere can be expressed by two angles, φ and θ, which are sometimes called *spherical coordinates*. (These will take the roles of the parameters u and v mentioned above.)

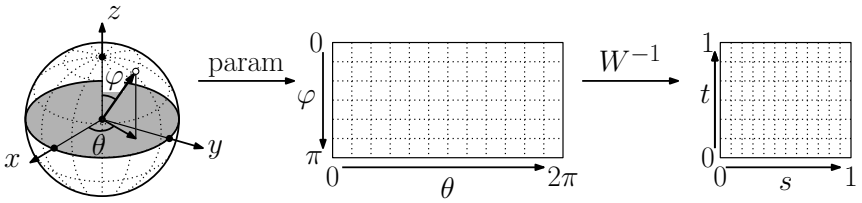


Fig. 48: Parameterization of a sphere.

Consider a vector from the origin to the desired point on the sphere. Let φ denote the angle in radians between this vector and the z-axis (north pole). So φ is related to, but not equal to,

the latitude. We have $0 \leq \varphi \leq \pi$. Let θ denote the counterclockwise angle of the projection of this vector onto the xy-plane. Thus $0 \leq \theta < 2\pi$ (see Fig. 48).

Our next task is to determine how to convert a point (x, y, z) on the sphere to a pair (θ, φ). It will be a bit easier to approach this problem in the reverse direction, by determining the (x, y, z) value that corresponds to a given parameter pair (θ, φ).

The z-coordinate is just $\cos\varphi$, and clearly this ranges from 1 to -1 as φ increases from 0 to π. To determine the value of θ, let us consider the projection of this vector onto the x, y-plane. Since the vertical component is of length $\cos\varphi$, and the overall length is 1 (since it's a unit sphere), by the Pythagorean theorem the horizontal length is $\ell = \sqrt{1 - \cos^2\varphi} = \sin\varphi$. The lengths of the projections onto the x and y coordinate axes are $x = \ell\cos\theta$ and $y = \ell\sin\theta$. Putting this all together, it follows that the (x, y, z) coordinates corresponding to the spherical coordinates (θ, φ) are

$$z(\varphi, \theta) \;=\; \cos\varphi, \qquad x(\varphi, \theta) \;=\; \sin\varphi \cdot \cos\theta, \qquad y(\varphi, \theta) \;=\; \sin\varphi \cdot \sin\theta.$$

But what we wanted to know was how to map (x, y, z) to (θ, φ). To do this, observe first that $\varphi = \arccos z$. It appears at first that θ will be much messier, but there is an easy way to get its value. Observe that $y/x = \sin\theta/\cos\theta = \tan\theta$. Therefore, $\theta = \arctan(y/x)$. In summary:

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

(Remember that this can be computed accurately as $\mathsf{atan2}(y, x)$.)

The final step is to map the parameter pair (θ, φ) to a point in (s, t) space. To get the s coordinate, we just scale θ from the range $[0, 2\pi]$ to $[0, 1]$. Thus, $s = \theta/(2\pi)$.

The value of t is trickier. The value of φ increases from 0 at the north pole π at the south pole, but the value of t decreases from 1 at the north pole to 0 at the south pole. After a bit of playing around with the scale factors, we find that $t = 1 - (\varphi/\pi)$. Thus, as φ goes from 0 to π, this function goes from 1 down to 0, which is just what we want. In summary, the desired inverse wrapping function is $W^{-1}(x, y, z) = (s, t)$, where:

$$\begin{aligned} s &= \frac{\theta}{2\pi}, \qquad \text{where} \quad \theta = \arctan\frac{y}{x}, \\ t &= 1 - \frac{\varphi}{\pi}, \qquad \text{where} \quad \varphi = \arccos z. \end{aligned}$$

Note that at the north and south poles there is a singularity in the sense that we cannot derive a unique value for θ. This is phenomenon is well known to cartographers. (What is the longitude of the north or south pole?)

To summarize, the *inverse wrapping* function $W^{-1}(x, y, z)$ maps a point on the surface to a point (s, t) in texture space. This is often done through a two step process, first determining the parameter values (u, v) associated with this point, and then mapping (u, v) to texture-space coordinates (s, t).This "unwrapping" function, maps the surface back to the texture. For this simple example, let's just set this function to the identity, that is, $W^{-1}(u, v) = (u, v)$. In general, we may want to stretch, translate, or rotate the texture to achieve the exact placement we desire.

The Texture Mapping Process: Suppose that the inverse wrapping function W^{-1}, and a parameterization of the surface are given. Here is an overview of an idealized version of the texture mapping process (see Fig. 49). (The actual implementation in common graphics systems differs for the sake of efficiency.)

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

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

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

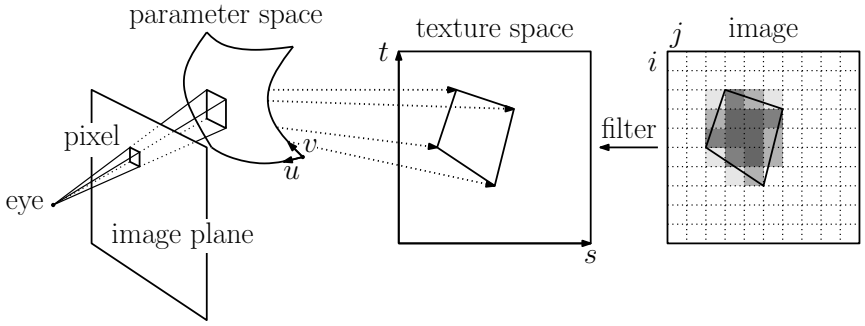


Fig. 49: Texture mapping overview.

We have covered the basic mathematical elements of texture mapping. In the next lecture, we will consider how to make this happen in OpenGL.

Texture Mapping in OpenGL: Recall that all objects in OpenGL are rendered as polygons or generally meshes of polygons. This simplifies the texture mapping process because it means that we need only provide the inverse wrapping function for the vertices, and we can rely on simple interpolation to fill in the polygon's interior. For example, suppose that a triangle is being drawn. When the vertices of the polygon are given, the user also specifies the corresponding (s, t) coordinates of these points in texture space. These are called the vertices' *texture coordinates*. This implicitly defines the inverse wrapping function from the surface of the polygon to a point in texture space.

As with surface normals (which were used for lighting computations) texture vertices are specified *before* each vertex is drawn. For example, a texture-mapped object in 3-space with

shading might be drawn using the following general form, where $\vec{n} = (n_x, n_y, n_z)$ is the surface normal, (s, t) are the texture coordinates, and $p = (p_x, p_y, p_z)$ is the vertex position.

```
glBegin(GL_POLYGON);
  glNormal3f(nx, ny, nz); glTexCoord2f(s, t); glVertex3f(px, py, pz);
  // ...
glEnd();
```

Interpolating Texture Coordinates: Given the texture coordinates, the next question is how to interplate the texture coordinates for the points in the interior of the polygon. An obvious approach is to first project the vertices of the triangle onto the viewport. This gives us three points p_0, p_1, and p_2 for the vertices of the triangle in 2-space. Let q_0, q_1 and q_2 denote the three texture coordinates, corresponding to these points. Now, for any pixel in the triangle, let p be its center. We can represent p uniquely as an affine combination

$$p = \alpha_0 p_0 + \alpha_1 p_1 + \alpha_2 p_2 \qquad \text{for } \alpha_0 + \alpha_1 + \alpha_2 = 1.$$

(Computing the α values for an arbitrary point of the polygon generally involves solving a system of linear equations, but there are simple and efficient methods which are discussed under the topic of polygon rasterization.) Once we have computed the α_i's the corresponding point in texture space is just

$$q = \alpha_0 q_0 + \alpha_1 q_1 + \alpha_2 q_2.$$

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

What is wrong with this direct approach? The first problem has to do with perspective. The direct approach makes the incorrect assumption that affine combinations are preserved under perspective projection. This is not true (see Fig. 50(c)).

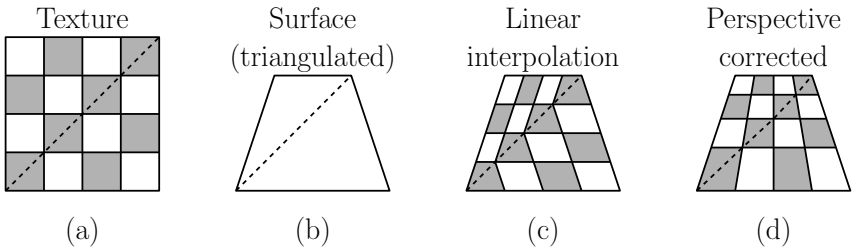


Fig. 50: Perspective correction.

There are a number of ways to fix this problem. One approach is to use a more complex formula for interpolation, which corrects for perspective distortion. (See the text for details. An example of the result is shown in Fig. 50(d)). This can be activated using the following OpenGL command:

```
glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
```

(The other possible choice is GL_FASTEST, which does simple linear interpolation.)

The other method involves slicing the polygon up into sufficiently small polygonal pieces, such that within each piece the amount of distortion due to perspective is small. Recall that with Gouraud shading, it was often necessary to subdivide large polygons into small pieces for accurate lighting computation. If you have already done this, then the perspective distortion due to texture mapping may not be a significant issue for you.

The second problem has to do with something called *aliasing*. Remember that we said that after determining the fragment of texture space onto which the pixel projects, we should average the colors of the texels in this fragment. The above procedure just considers a single point in texture space, and does no averaging. In situations where the pixel corresponds to a point in the distance and hence covers a large region in texture space, this may produce very strange looking results, because the color of the entire pixel is determined entirely by a single point in texture space that happens to correspond (say) to the pixel's center coordinates (see Fig. 51(a)).

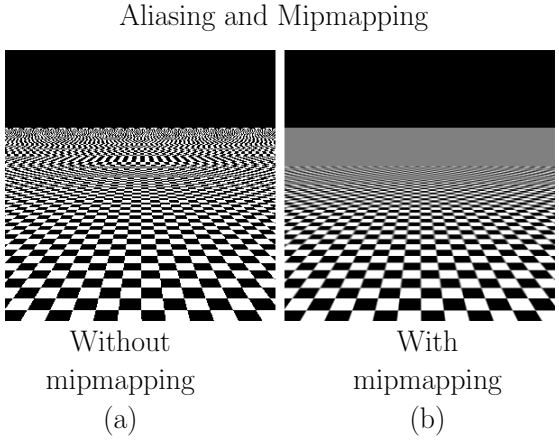


Fig. 51: Aliasing and mipmapping.

Dealing with aliasing in general is a deep topic, which is studied in the field of signal processing. OpenGL applies a simple method for dealing with the aliasing of this sort. The method is called *mipmapping*. (The acronym "mip" comes from the Latin phrase *multum in parvo*, meaning "much in little.")

The idea behind mipmapping is to generate a series of texture images at decreasing levels of resolution. For example, if you originally started with a 128×128 image, a mipmap would consist of this image along with a 64×64 image, a 32×32 image, a 16×16 image, etc. All of these are scaled copies of the original image. Each pixel of the 64×64 image represents the average of a 2×2 block of the original. Each pixel of the 32×32 image represents the average of a 4×4 block of the original, and so on (see Fig. 52).

Now, when OpenGL needs to apply texture mapping to a screen pixel that overlaps many texture pixels (that is, when "minimizing" the texture), it determines the mipmap in the hiearchy that is at the closest level of resolution, and uses the corresponding averaged pixel value from that mipmapped image. This results in more nicely blended images (see Fig. 51(b)).

If you wish to use mipmapping in OpenGL (and it is a good idea), it is good to be aware of

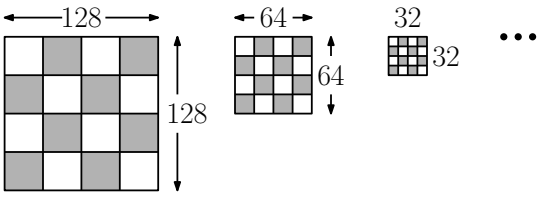


Fig. 52: Mipmapping.

the command gluBuild2DMipmaps, which can be used to automatically generate them.

Texture mapping in OpenGL: OpenGL supports a fairly general mechanism for texture mapping. The process involves a bewildering number of different options. You are referred to the OpenGL documentation for more detailed information. By default, objects are not texture mapped. If you want your objects to be colored using texture mapping, you need to enable texture mapping before you draw them. This is done with the following command.

```
glEnable(GL_TEXTURE_2D);
```

After drawing textured objects, you can disable texture mapping.

If you plan to use more than one texture, then you will need to request that OpenGL generate texture objects. This is done with the following command:

```
glGenTextures(GLsizei n, GLuint* textureIDs);
```

This requests that n new texture objects be created. The n new texture id's are stored as unsigned integers in the array *textureIDs*. Each texture ID is an integer greater than 0. (Typically, these are just integers 1 through n, but OpenGL does not require this.) If you want to generate just one new texture, set $n = 1$ and pass it the address of the unsigned int that will hold the texture id.

By default, most of the texture commands apply to the "active" texture. How do we specify which texture object is the active one? This is done by a process called *binding*, and is defined by the following OpenGL command:

```
glBindTexture(GLenum target, GLuint textureID);
```

where *target* is one of GL_TEXTURE_1D, GL_TEXTURE_2D, or GL_TEXTURE_3D, and *textureID* is one of the texture IDs returned by glGenTextures. The *target* will be GL_TEXTURE_2D for the sorts of 2-dimensional image textures we have been discussing so far. The *textureID* parameter indicates which of the texture IDs will become the active texture. If this texture is being bound for the first time, a new texture object is created and assigned the given texture ID. If *textureID* has been used before, the texture object with this ID becomes the active texture.

Presenting your Texture to OpenGL: The next thing that you need to do is to input your texture and present it to OpenGL in a format that it can access efficiently. It would be nice if you could just point OpenGL to an image file and have it convert it into its own internal

format, but OpenGL does not provide this capability. You need to input your image file into an array of RGB (or possibly RGBA) values, one byte per color component (e.g. three bytes per pixel), stored row by row, from upper left to lower right. By the way, OpenGL requires images whose height and widths are powers of two.

Once the image array has been input, you need to present the texture array to OpenGL, so it can be converted to its internal format. This is done by the following procedure. There are many different options, which are partially explained in below.

```
glTexImage2d(GL_TEXTURE_2D, level, internalFormat, width, height,
  border, format, type, image);
```

The procedure has an incredible array of options. Here is a simple example to present OpenGL an RGB image stored in the array myPixelArray. The image is to be stored with an internal format involving three components (RGB) per pixel. It is of width *nCols* and height *nRows*. It has no border (*border* = 0), and we are storing the highest level resolution. (Other levels of resolution are used to implement the averaging process, through a method called *mipmaps*.) Typically, the level parameter will be 0 (*level* = 0). The format of the data that we will provide is RGB (GL_RGB) and the type of each element is an unsigned byte (GL_UNSIGNED_BYE). So the final call might look like the following:

```
glTexImage2d(GL_TEXTURE_2D, 0, GL_RGB, nCols, nRows, 0, GL_RGB,
  GL_UNSIGNED_BYTE, myPixelArray);
```

In this instance, your array *myPixelArray* is an array of size $256 \times 512 \times 3 = 393,216$ whose elements are the RGB values, expressed as unsigned bytes, for the 256×512 texture array. An example of a typical texture initialization is shown in the code block below. (The call to gluBuild2DMipmpaps is needed only if mipmapping is to be used.)

Initialization for a Single Texture

```
GLuint textureID;                     // the ID of this texture
glGenTextures(1, &textureID);         // assign texture ID
glBindTexture(GL_TEXTURE_2D, textureID); // make this the active texture
//
// ... input image nRows x nCols into RGB array myPixelArray
//
glTexImage2d(GL_TEXTURE_2D, 0, GL_RGB, nCols, nRows, 0, GL_RGB,
  GL_UNSIGNED_BYTE, myPixelArray);
                                      // generate mipmaps (see below)
gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB, nCols, nRows, GL_RGB,
  GL_UNSIGNED_BYTE, myPixelArray);
```

Texturing Options: Once the image has been input and presented to OpenGL, we need to tell OpenGL how it is to be mapped onto the surface. Again, OpenGL provides a large number of different methods to map a surface. These parameters are set using the following function:

```
glTexParameteri(target, param_name, param_value);
glTexParameterf(target, param_name, param_value);
```

The first form is for integer parameter values and the second is for float values. For most options, the argument *target* will be set to GL_TEXTURE_2D. There are two common parameters to set. First, in order to specify that a texture should be repeated or not (clamped), the following options are useful.

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
```

These options determine what happens of the s parameter of the texture coordinate is less than 0 or greater than 1. (If this never happens, then you don't need to worry about this option.) The first causes the texture to be displayed only once. In particular, values of s that are negative are treated as if they are 0, and values of s exceeding 1 are treated as if they are 1. The second causes the texture to be wrapped-around repeatedly, by taking the value of s modulo 1. Thus, $s = 5.234$ amd $s = 76.234$ are both equivalent to $s = 0.234$. This can independently be set for he t parameter of the texture coordinate, by setting GL_TEXTURE_WRAP_T.

Filtering and Mipmapping: Another useful parameter determines how rounding is performed during *magnification* (when a screen pixel is smaller than the corresponding texture pixel) and "*minification*" (when a screen pixel is larger than the corresponding texture pixel). The simplest, but not the best looking, option in each case is to just use the *nearest pixel* in the texture:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

A better approach is to use linear filtering when magnifying and mipmaps when minifying. An example is given below.

Combining Texture with Lighting: How are texture colors combined with object colors? The two most common options are GL_REPLACE, which simply makes the color of the pixel equal to the color of the texture, and GL_MODULATE (the default), which makes the colors of the pixel the product of the color of the pixel (without texture mapping) times the color of the texture. The former is for painting textures that are already *prelit*, meaning that lighting has already been applied. Examples include skyboxes and precomputed lighting for the ceiling and walls of a room. The latter is used when texturing objects to which lighting is to be applied, such as the clothing of a moving character.

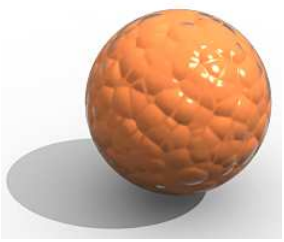
```
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
```

Drawing a Texture Mapped Object: Once the initializations are complete, you are ready to start drawing. First, bind the desired texture (that is, make it the active texture), set the texture parameters, and enable texturing. Then start drawing your textured objects. For each vertex drawn, be sure to specify the texture coordinates associated with this vertex, prior to issuing the glVertex command. If lighting is enabled, you should also provide the surface normal. A generic example is shown in the code block below.

```
glEnable(GL_TEXTURE_2D);              // enable texturing
glBindTexture(GL_TEXTURE_2D, textureID); // select the active texture
                                      // (use GL_REPLACE below for skyboxes)
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
                                      // repeat texture
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
                                      // reasonable filter choices
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
glBegin(GL_POLYGON);                  // draw the object(s)
  glNormal3f( ... );                  // set the surface normal
  glTexCoord2f( ... );                // set texture coords
  glVertex3f( ... );                  // draw vertex
  // ... (repeat for other vertices)
glEnd()
glDisable(GL_TEXTURE_2D);             // disable texturing
```

Lecture 14: Bump Mapping

Bump mapping: Texture mapping is good for changing the surface color of an object, but we often want to do more. For example, if we take a picture of an orange, and map it onto a sphere, we find that the resulting object does not look realistic. The reason is that there is an interplay between the bumpiness of the orange's peel and the light source. As we move our viewpoint from side to side, the specular reflections from the bumps should move as well. However, texture mapping alone cannot model this sort of effect. Rather than just mapping colors, we should consider mapping whatever properties affect local illumination. One such example is that of mapping surface normals, and this is what *bump mapping* is all about (see Fig. 53).



A bump-mapped object

A bump-map

Fig. 53: Bump mapping.

What is the underlying reason for this effect? The bumps are too small to be noticed through perspective depth. It is the subtle variations in *surface normals* that causes this effect. At first it seems that just displacing the surface normals would produce a rather artificial effect. But in fact, bump mapping produces remarkably realistic bumpiness effects. (For example, in Fig. 53, the object appears to have a bumpy exterior, but an inspection of its shadow shows

that it is in fact modeled as a perfect geometric sphere. It just "looks" bumpy.)

How it's done: As with texture mapping we are presented with an image that encodes the bumpiness. Think of this as a monochrome (gray-scale) image, where a large (white) value is the top of a bump and a small (black) value is a valley between bumps. (An alternative, and more direct way of representing bumps would be to give a *normal map* in which each pixel stores the (x, y, z) coordinates of a normal vector. One reason for using gray-valued bump maps is that they are often easier to compute and involve less storage space.) As with texture mapping, it will be more elegant to think of this discrete image as an encoding of a continuous 2-dimensional *bump space*, with coordinates s and t. The gray-scale values encode a function called the *bump displacement function* $b(s, t)$, which maps a point (s, t) in bump space to its (scalar-valued) height. As with texture mapping, there is an *inverse wrapping function* W^{-1}, which maps a point (u, v) in the object's surface parameter space to (s, t) in bump space.

Parametric Surfaces, Partial Derivatives, and Tangents: Before getting into bump mapping, let's take a short digression to talk a bit about parameterized surfaces and surface derivatives. Since surfaces are two-dimensional, a surface can be presented as a parametric function in two parameters u and v. That is, each point $p(u, v)$ on the surface is given by three coordinate functions $x(u, v)$, $y(u, v)$, and $z(u, v)$. That is,

$$p(u, v) = \begin{pmatrix} x(u, v) \\ y(u, v) \\ z(u, v) \end{pmatrix}.$$

As an example, let us consider a cone of height 1 whose apex is at the origin, whose axis coincides with the z-axis, and whose interior angle with the axis is 45° (see Fig. 54(a)). We can express any point on this cone by thinking of v as indicating the height of the point above the (x, y)-plane, and identifying u with a rotational axis about the z-axis. Clearly, $z(u, v) = v$. Since the angle of the cone about the central axis is 45°, a point at height v projects to a point at distance v from the origin. Therefore, we have $x(u, v) = v \cos(2\pi \cdot u)$ and $y(u, v) = v \sin(2\pi \cdot u)$. Thus, we have

$$p(u, v) = \begin{pmatrix} v \cos(2\pi \cdot u) \\ v \sin(2\pi \cdot u) \\ v \end{pmatrix}, \quad \text{where } 0 \le v \le 1, \quad 0 \le u \le 1.$$

To get a specific point, we plug in any two valid values of u and v. For example, when $u = 0$ and $v = 1$, we have the point

$$p(0, 1) = (1, 0, 1)^{\mathsf{T}},$$

which is a point on the cone immediately above the tip of the x-axis.

Returning to the general case, let us assume for now that we know these three functions of u and v. In order to compute a surface normal at this point, we would like to produce two vectors that are tangent to the surface, and then take their cross product.

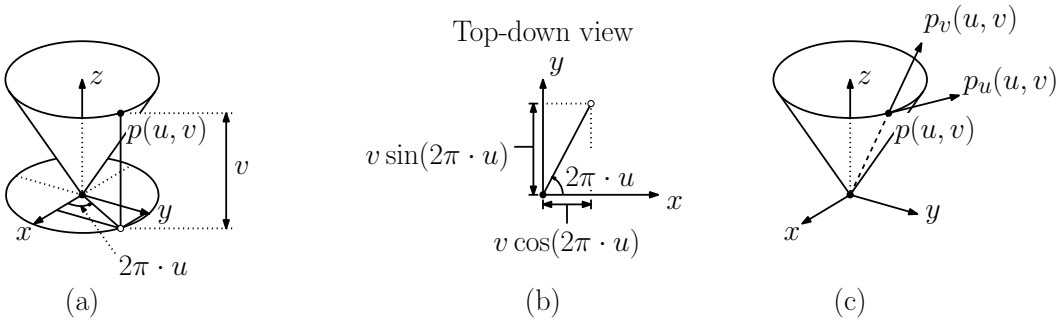


Fig. 54: Parameterization of a cone.

To do this, consider the partial derivative of $p(u, v)$ with respect to u, which we will denote by $\vec{p}_u$.

$$\vec{p}_u(u, v) = \begin{pmatrix} \partial x(u, v)/\partial u \\ \partial y(u, v)/\partial u \\ \partial z(u, v)/\partial u \end{pmatrix}.$$

(Note that, although $p(u, v)$ is a point, its partial derivative should be interpreted as a vector in 3-dimensional space. As u and v vary, the direction in which this vector points changes.) Recall from differential calculus that this means computing the derivative of each of these functions, but treating the variable v as if it were a constant.

We can assign a geometric interpretation to this vector as follows. As u varies over time, the point $p(u, v)$ traces out a curve along the surface. The partial derivative vector $\vec{p}_u(u, v)$ can be thought of as the instantaneous velocity of a point on this curve. That is, it is a tangent vector on the surface which points in the direction along which the surface changes most rapidly as a function of u.

Analogously, we can define the partial derivative $\vec{p}_v$ to be

$$\vec{p}_v(u, v) = \begin{pmatrix} \partial x(u, v)/\partial v \\ \partial y(u, v)/\partial v \\ \partial z(u, v)/\partial v \end{pmatrix}.$$

When evaluated at any point $p(u, v)$ this is a tangent vector pointing in the direction of most rapid change for v. Thus, the cross product $\vec{p}_u(u, v) \times \vec{p}_v(u, v)$ gives us a normal vector to the surface. Depending on the orientations of the parameterization, we may need to reverse the two vectors to get the normal to point in the desired direction.

For example, in the case of our cone, we have

$$\vec{p}_u(u, v) = (-2\pi \cdot v \cdot \sin(2\pi \cdot u),\ 2\pi \cdot v \cdot \cos(2\pi \cdot u)),\ 0)^{\mathsf{T}}.$$

If you think about it a bit, this can be seen to be a vector that is tangent to the horizontal circle passing through $p(u, v)$ (see Fig. 54(c)). Also, we have

$$\vec{p}_v(u, v) = (\cos(2\pi \cdot u),\ \sin(2\pi \cdot u),\ 1)^{\mathsf{T}}.$$

This can be seen to be a vector that is directed along a slanted line passing through $p(u, v)$ that is aligned with the cone's boundary (see Fig. 54(c)). Thus, these two partial derivatives give us (for any valid choice of u and v) two surface tangents.

To obtain the surface normal, we compute the cross product of these vectors. This gives us (trust me)

$$\vec{n}(u, v) \;=\; \vec{p}_u(u, v) \times \vec{p}_v(u, v) \;=\; (2\pi \cdot v)\,(\cos(2\pi \cdot u),\, \sin(2\pi \cdot u),\, -1)^{\mathsf{T}}\,.$$

If we use the same values $(u, v) = (0, 1)$, we obtain the tangent vectors

$$\vec{p}_u(0, 1) \;=\; 2\pi\,(0, 1, 0)^{\mathsf{T}} \qquad \text{and} \qquad \vec{p}_v(0, 1) \;=\; (1, 0, 1)^{\mathsf{T}}\,.$$

If you draw the picture carefully, you can see that these two vectors are tangent to the cone at the point lying immediately above the tip of the x-axis. Finally, if we compute their cross product, we have

$$\vec{n}(0, 1) \;=\; 2\pi\,(1, 0, -1)^{\mathsf{T}}\,.$$

Again, if you think about it carefully, this is a normal vector to the cone at this point.

Perturbing normal vectors: Now, let us return to the original question of how to compute the perturbed normal vector. Consider a point $p(u, v)$ on the surface of the object (which we will just call p). Let $\vec{n}$ denote the surface normal vector at this point. Let $(s, t) = W^{-1}(u, v)$, so that $b(s, t)$ is the corresponding bump value (see Fig. 55(a)). The question is, what is the *perturbed normal* $\vec{n}'$ for the point p according to the influence of the bump map? Once we know this normal, we just use it in place of the true normal in our Phong illumination computations.

Here is a method for computing the perturbed normal vector. The idea is to imagine that the bumpy surface has been wrapped around the object. The question is how do these bumps affect the surface normals? This is illustrated in Fig. 55(a).

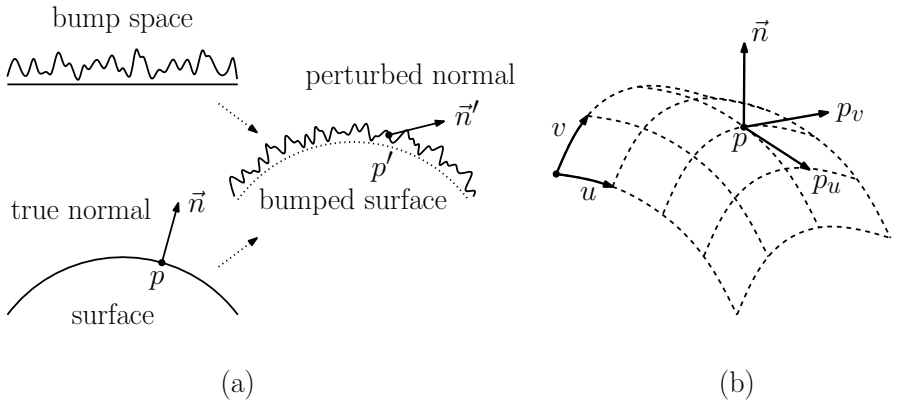


Fig. 55: The bump-mapping process.

All the geometric entities we will be considering here and below (e.g., p, $\vec{p}_u$, $\vec{p}_v$, b, $\vec{n}$, $\vec{n}'$) are functions of u and v. Henceforth, to simplify notation, we will omit the (u, v) arguments.

Let $\vec{p}_u$ denote the partial derivative of p with respect to u and define $\vec{p}_v$ similarly with respect to v. That is,

$$\vec{p}_u = \begin{pmatrix} \partial x/\partial u \\ \partial y/\partial u \\ \partial z/\partial u \end{pmatrix} \qquad \vec{p}_v = \begin{pmatrix} \partial x/\partial v \\ \partial y/\partial v \\ \partial z/\partial v \end{pmatrix}.$$

As mentioned earlier, we can think of u and v values as defining a sort of graph-paper grid that is wrapped over our surface (see Fig. 55(b)), and $\vec{p}_u$ and $\vec{p}_v$ can be viewed as tangent vectors passing through point p, where $\vec{p}_u$ is parallel to the u-line of the graph paper passing through p and $\vec{p}_v$ is parallel to the v-line of the graph paper passing through p. The normal vector $\vec{n}$ is (up to a scale factor) given by their cross product, that is,

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

This is illustrated in Fig. 55(b). Since $\vec{n}$ may not generally be of unit length, we define $\widehat{n} = \vec{n}/\|\vec{n}\|$ to be the normalized normal vector.

If we apply our bump at point p, it will be elevated by a distance $b = b(u, v)$ in the direction of the normal (thus, b is a scalar). So we have

$$p' = p + b\widehat{n},$$

is the elevated point. (Ultimately, we do not need to compute p', since we are not actually displacing the surface. We are using p' as a means to determine the perturbed normal vector.) Determining the perturbed normal at p' requires that we know the partial derivatives of $p'(u, v)$ with respect to u and v. Letting $\vec{n}'$ denote this perturbed normal we have

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

where $\vec{p}'_u$ and $\vec{p}'_v$ are the partial derivatives of p' with respect to u and v, respectively. Thus we have

$$\vec{p}'_u \;=\; \frac{\partial}{\partial u}(p + b\widehat{n}) \;=\; \vec{p}_u + b_u\widehat{n} + b\widehat{n}_u,$$

where b_u and $\widehat{n}_u$ denote the respective partial derivatives of b and $\widehat{n}$ with respect to u. An analogous formula applies for $\vec{p}'_v$. Assuming that the height of the bump b is small but its rate of change b_u and b_v may be high, we can neglect the last term, and write these as

$$\vec{p}'_u \;\approx\; \vec{p}_u + b_u\widehat{n} \qquad \vec{p}'_v \;\approx\; \vec{p}_v + b_v\widehat{n}.$$

Taking the cross product (and recalling that cross product distributes over vector addition) we have

$$\begin{aligned} \vec{n}' &\approx (\vec{p}_u + b_u\widehat{n}) \times (\vec{p}_v + b_v\widehat{n}) \\ &\approx (\vec{p}_u \times \vec{p}_v) + b_v(\vec{p}_u \times \widehat{n}) + b_u(\widehat{n} \times \vec{p}_v) + b_u b_v(\widehat{n} \times \widehat{n}). \end{aligned}$$

By basic properties of the cross product, we know that $\widehat{n} \times \widehat{n} = 0$ and $(\vec{p}_u \times \widehat{n}) = -(\widehat{n} \times \vec{p}_u)$. Thus, we have

$$\vec{n}' \approx \vec{n} + b_u(\widehat{n} \times \vec{p}_v) - b_v(\widehat{n} \times \vec{p}_u).$$

The partial derivatives b_u and b_v depend on the particular parameterization of the object's surface. It will greatly simplify matters to assume that the object's parameterization has been constructed in common alignment with the image. (If not, then the formulas become much messier.) With this assumption, we have the following formula for the perturbed surface normal:

$$\vec{n}' \approx \vec{n} + b_s(\widehat{n} \times \vec{p}_v) - b_t(\widehat{n} \times \vec{p}_u).$$

This is the final answer that we desire. Note that for each point on the surface we need to know the following quantities:

- The true surface normal $\vec{n}$ and its normalization $\widehat{n}$.
- The partial derivative vectors $\vec{p}_u$ and $\vec{p}_v$. We can compute these if we have an algebraic representation of the surface, e.g., as we did for the earlier cone example.
- The partial derivatives of the bump function b with respect to the texture parameters s and t, denoted b_s and b_t. (Typically, we do not have an algebraic representation of the bump function, since it is given as a gray-scale image. However, we can estimate these derivatives from the bump image through the use of finite differences.)

In summary, for each point p on the surface with (smooth) surface normal $\vec{n}$ we apply the above formula to compute the perturbed normal $\vec{n}'$. Now we proceed as we would in any normal lighting computation, but instead of using $\vec{n}$ as our normal vector, we use $\vec{n}'$ instead. OpenGl does not provide direct support bump mapping, but there are some extensions of OpenGl that provide for an alternate form of bump mapping, called normal maps. We will discuss this next time.

Lecture 15: Normal and Environment Mapping

Normal Maps: Bump mapping uses a single monochromatic (gray-scale) and some differential geometry to generate perturbed normals at each point of a surface. You might ask, why derive the surface normals, couldn't you just store them in the map instead? This is exactly what normal mapping does. In particular, a *normal map* is an RGB color image, in which each (R, G, B) triple is interpreted as the (x, y, z) coordinates of a normal vector. As in bump mapping, this normal vector is not used directly, but rather is treated as a *perturbation* to an existing normal vector.

The obvious disadvantage of normal mapping is that it requires more space than bump mapping, since a bump map requires only gray-scale information and a normal map requires three color components per pixel.[3] One advantage of normal mapping is that it is faster, because we do not need to perform the cross product computations needed by bump mapping. Also, it accords some greater degree of flexibility, since bump maps can only store normal displacements that are consistent with a bump height function.

Encoding Normals as Colors: How are (perturbed) normals encoded in a normal map? There are a number of conventions, but the most common is based on identifying each possible

[3]Actually, two components would have been sufficient. You only need a direction to encode a normal vector, and a direction in 3-dimensional space can be encoded with two angles, elevation and azimuth. But, since images naturally come in RGB triplets, normal maps follow this 3-dimensional convention.

(R, G, B) value with a vector from the origin to the top face of a unit cube. Let us assume that the color components range from 0 to 1 (see Fig. 56(a)). (They typically are stored as a single byte, and so range from 0 to 255. Thus, we divide each by 256, yielding a value in the interval $[0, 1)$.)

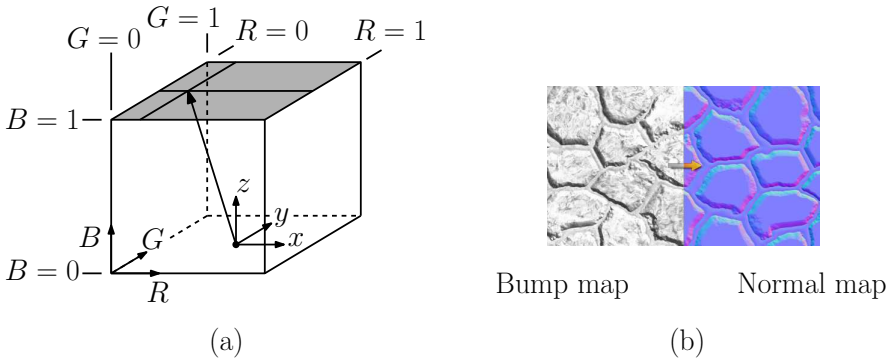


Fig. 56: Encoding normals as colors.

We will assume that the B value is used to encode the z-coordinate of the normal vector, which we will assume to be 1. Thus, $B = z = 1$. (This is an obvious source of inefficiency.) In order to map an R component to an x coordinate, we set $x = 2 \cdot R - 1$. Thus, as R ranges from 0 to 1, x ranges from -1 to $+1$. Similarly, we set $y = 2 \cdot G - 1$. In summary, each for each RGB triple of our image, we assume that $B = 1$ and $0 \leq R, G \leq 1$. Thus, a given (R, G, B) triple is mapped to the normal vector:

$$(x, y, z) \;=\; (2R - 1, 2G - 1, B) \;=\; 2(R, G, B) - 1.$$

Above, we have used the fact that $B = 1$, so $2B - 1 = 1 = B$. (An example of a bump map, that is, a height function, and the equivalent normal map is shown in Fig 56(b).)

The Normal Mapping Process: The rest of the process is conceptually the same as it is for bump mapping. Consider a pixel p that we wish to determine the lighting for, and let $\vec{p}_u$ and $\vec{p}_v$ denote two (ideally orthogonal) tangent vectors on the object's surface at the current point. Let us assume that they have both been normalized to unit length. Let $\vec{n} = \vec{p}_u \times \vec{p}_v$ denote the standard normal vector for the current surface point (see Fig. 57(a)). We proceed as follows:

Unwrap and look-up: Based on the inverse wrapping function (which is given by the user), we do a look-up in the normal map to obtain the appropriate (R, G, B) triple. (As with normal texture mapping, we may perform some smoothing to avoid aliasing effects.)

Convert: We perform the aforementioned conversion to obtain the (x, y) pair associated with this triple (see Fig. 57(b)).

Compute final normal: We compute the perturbed normal by scaling the tangents by x and y, respectively, and adding them to the standard normal (see Fig. 57(c)). Thus, we have

$$\vec{n}' \;=\; \vec{n} + x \cdot \vec{p}_u + y \cdot \vec{p}_v.$$

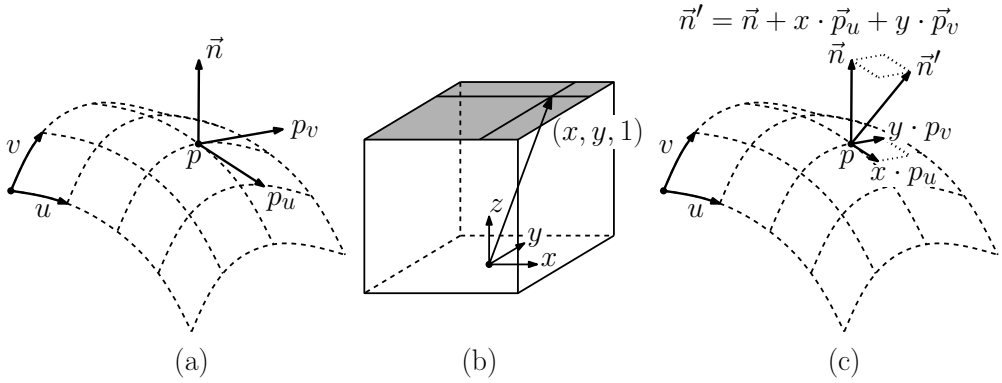


Fig. 57: Computing the perturbed normal.

Lighting: Given the perturbed normal, $\vec{n}'$, we normalize it to unit length. We then apply the standard Phong lighting model, but rather than using the geometric surface normal, $\vec{n}$, we use the perturbed normal, $\vec{n'}$, instead.

Environment Mapping: Next, we consider another method of applying surface detail to model reflective objects. Suppose that you are looking at a shiny waxed floor, or a metallic sphere. We have seen that we can model the shininess by setting a high coefficient of specular reflection in the Phong model, but this will mean that the only light sources will be reflected (as bright spots). Suppose that we want the surfaces to actually reflect the surrounding environment. This sort of reflection of the environment is often used in commercial computer graphics. The shiny reflective lettering and logos that you see on television, the reflection of light off of water, the shiny reflective look of a automobile's body, are all examples (see Fig. 58).

An environment mapped teapot

Fig. 58: Environment mapping.

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

What distinguishes reflection from texture? When you use texture mapping to "paint" a texture onto a surface, the texture stays put. For example, if you fix your eye on a single point of the surface, the color stays the same, even if you change your viewing position. In

contrast, reflective objects have the property that, as you move your head and look at the same point on the surface, the reflected color changes. This is because reflection is a function of the position of the viewer, while normal surface colors are not.

Computing Reflections: How can we encode such a complex reflective relationship? The basic question that we need to answer is, given a point on the reflective surface, and given the location of the viewer, determine what the viewer sees in the reflection. Before seeing how this is done in environment mapping, let's see how this is done in the more accurate method called *ray tracing*. In ray tracing we track the path of a light photon backwards from the eye to determine the color of the object that it originated from. When a photon strikes a reflective surface, it bounces off. If v is the (normalized) view vector and $\vec{n}$ is the (normalized) surface normal vector, we can compute the *view reflection vector*, denoted r_v, as follows (see Fig. 59):

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

To compute the "true" reflection, we should trace the path of this ray back from the point on the surface along r_v. Whatever color this ray hits, will be the color that the viewer observes as reflected from this surface.

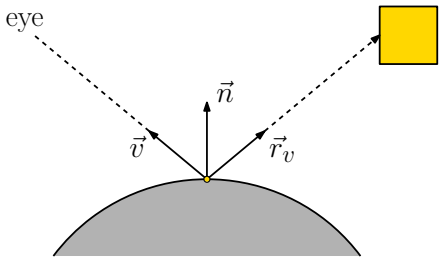


Fig. 59: Reflection vector.

Unfortunately, it is expensive to shoot rays through 3-dimensional environments to determine what they hit. (This is exactly what ray-tracing does.) Instead, we will precompute the reflections, store them in an image file, and look them up as needed. But, storing reflections accurately is very complicated. (An accurate representation of the reflection would be like a hologram, since it would have to store all the light energy arriving from all angles to all points on the surface.) To make the process tractable, we will make an important simplifying assumption:

Distant Environment Assumption: (For environment mapping) The reflective surface is small in comparison with the distances to the objects being reflected in it.

For example, the reflection of a room surrounding a silver teapot would satisfy this requirement. However, if the teapot is sitting on a table, then the table would be too close (resulting in a distorted reflection).

This assumption implies that the most important parameter in determining what the reflection ray hits is the *direction* of the reflection vector, and not the actual point of this vector on the surface from which the ray starts. Happily, the space of directions is a 2-dimensional

space. (Recall, that a direction can be stored as an azimuth, that is, a compass direction, and an elevation, that is, the angle above the horizon.) This implies that we can precompute the (approximate) reflection information and store it in a 2-dimensional image array.

The environment mapping process: Here is a sketch of how environment mapping can be implemented. The first thing you need to do is to compute the environment map. To do this, remove the reflective object from your environment. Place a small cube about the center of the object. (Spheres are also commonly used, and in fact, OpenGL assumes spherical environment maps.) The cube should be small enough that it does not intersect any of the surrounding objects.

Next, project the entire environment onto the six faces of the cube, using the center of the cube as the center of projection. That is, take six separate pictures which together form a complete panoramic picture of the surrounding environment, and store the results in six image files. (OpenGL provides the ability to generate an image, and rather sending it to the display buffer, it can be saved in memory.) By the way, an accurate representation of the environment is not always necessary in order to generate the illusion of reflectivity.

Now suppose that we want to compute the color reflected from some point p on the object. As in the Phong model we compute the usual vectors: normal vector $\vec{n}$, view vector v, etc. We compute the view reflection vector r_v from these two. (This is not the same as the light reflection vector, r, which we discussed in the Phong model, but it is the counterpart where the reflection is taken with respect to the viewer rather than the light source.)

To determine the reflected color, we imagine that the view reflection vector r_v is shot from the *center of the cube* and determine the point on the cube which is hit by this ray. We use the color of this point to color the corresponding point on the surface (see Fig. 60). (We will leave as an exercise the problem of mapping a vector to a point on the surface of the cube.)

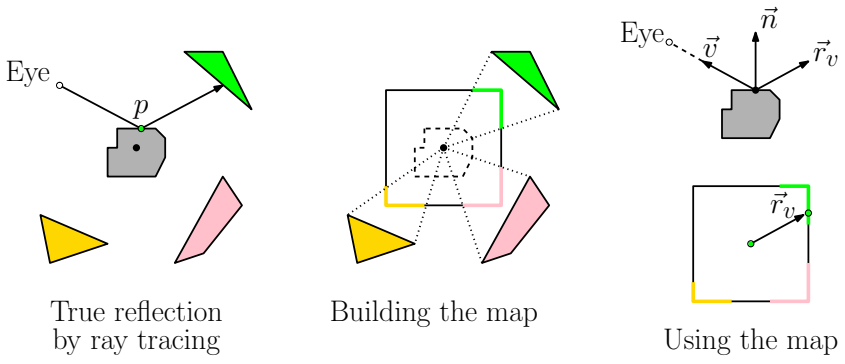


Fig. 60: Environment mapping.

Note that the final color returned by the environment map is a function of the contents of the environment image and r_v (and hence of v and $\vec{n}$). In particular, it is *not* a function of the location of the point on the surface. Wouldn't taking this location into account produce more accurate results? Perhaps, but by our assumption that objects in the environment are

far away, the directional vector is the most important parameter in determining the result. (If you really want accuracy, then use ray tracing instead.)

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

What makes the approach visually convincing is that, when the viewer shifts positions, the texture coordinates of the vertices change as well. In standard texture mapping, these coordinates would be fixed, independent of the viewer's position.

Lecture 16: Ray Tracing and Picking

Ray Tracing: Ray tracing is among the conceptually simplest methods for synthesizing highly realistic images. Unlike the simple polygon rendering methods used by OpenGL, ray tracing can easily produce shadows, and it can model reflective and transparent objects. (Fig. 61 shows an image generated by Andre Kirschner.)

Fig. 61: A ray traced image.

Because it is relatively slow, ray tracing is typically used for generating highly realistic images offline (as opposed to interactively). For example, it is used when interactivity is not needed, as in producing high quality animated films. It is also useful for generating realistic texture maps and environment maps that could later be used in interactive rendering. Ray tracing

also forms the basis of many approaches to more producing highly realistic complex types of shading and lighting.

Ray tracing can also be used in the context of interactive computer graphics for a process called *picking*. In picking, we imagine that we have rendered a 3-dimensional scene, and the user has placed the cursor over one of the objects that has been drawn, and we want to know which object the user is referring to. OpenGL provides a number of methods to perform picking. Raytracing can be used for picking. If you imagine a ray shot from the viewer's eye through the screen pixel associated with the mouse coordinates, the first object this ray hits in the scene is the visible object that the user is referring to.

In spite of its conceptual simplicity, ray tracing can be computationally quite intensive (particularly when applied to the generation of complex high resolution images). We will discuss the basic elements of ray tracing, revisit the Phong lighting model in this context, and discuss some of the details of generating rays and handling ray intersections.

Ray Tracing for Image Synthesis: Imagine that the viewing window is replaced with a fine mesh of horizontal and vertical grid lines, so that each grid square corresponds to a pixel in the final image. We shoot rays out from the eye through the center of each grid square in an attempt to trace the path of light backwards toward the light sources. Consider the first object that such a ray hits. (In order to avoid problems with jagged lines, called *aliasing*, it is more common to shoot a number of rays per pixel and average their results.) We want to know the intensity of reflected light at this surface point. This depends on a number of things, principally the reflective and color properties of the surface, and the amount of light reaching this point from the various light sources. The amount of light reaching this surface point is the hard to compute accurately. This is because light from the various light sources might be blocked by other objects in the environment and it may be reflected off of others.

A purely local approach to this question would be to use the model we discussed in the Phong model, namely that a point is illuminated if the angle between the normal vector and light vector is acute. In ray tracing it is common to use a somewhat more global approximation. We will assume that the light sources are points. For each light source L_i, we shoot a ray R_{L_i} from the surface point to each of the light sources (see Fig. 62(a)). For each of these rays that succeeds in reaching a light source before being blocked another object, we infer that this point is illuminated by this source (as for L_1 in Fig. 62(a)), and otherwise we assume that it is not illuminated, and hence we are in the shadow of the blocking object (as for L_2 in Fig. 62(a)). (Can you imagine a situation in which this model will fail to correctly determine whether a point is illuminated?)

Given the direction to the light source and the direction to the viewer, and the surface normal (which we can compute because we know the object that the ray struck), we have all the information that we need to compute the reflected intensity of the light at this point, say, by using the Phong model and information about the ambient, diffuse, and specular reflection properties of the object. We use this model to assign a color to the pixel. We simply repeat this operation on all the pixels in the grid, and we have our final image.

Even this simple ray tracing model is already better than what OpenGL supports, because, for example, OpenGL's local lighting model does not compute shadows. The ray tracing model can easily be extended to deal with reflective objects (such as mirrors and shiny spheres) and

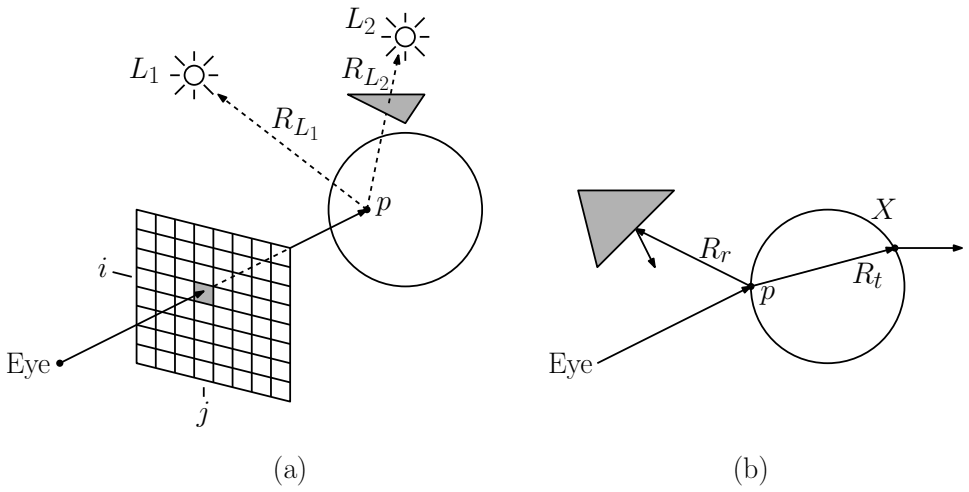


Fig. 62: Ray Tracing.

transparent objects (glass balls and rain drops). For example, when the ray hits a reflective object, we compute the reflection ray and shoot it into the environment. We invoke the ray tracing algorithm *recursively* (see Fig 62(b)). When we get the associated color, we blend it with the local surface color and return the result. The generic algorithm is outlined below.

rayTrace() : Given the camera setup and the image size, generate a ray R_{ij} from the eye passing through the center of each pixel (i, j) of your image window (See Fig. 62.) Call trace(R) and assign the color returned to this pixel.

trace(R) : Shoot R into the scene and let X be the first object hit and p be the point of contact with this object.

(a) If X is reflective, then compute the reflection ray R_r of R at p. Let $C_r \leftarrow \text{trace}(R_r)$.

(b) If X is transparent, then compute the transmission (refraction) ray R_t of R at p. Let $C_t \leftarrow \text{trace}(R_t)$.

(c) For each light source L,

(i) Shoot a ray R_L from p to L.

(ii) If R_L does not hit any object until reaching L, then apply the lighting model to determine the shading at this point.

(d) Combine the colors C_r and C_t due to reflection and transmission (if any) along with the combined shading from (c) to determine the final color C. Return C.

Ray Tracing for Picking: Another application of ray tracing is for the process of identifying the object associated with a particular pixel of the image. Suppose that the user places the cursor over the window at row i and column j. In order to determine the object of the scene that lies under this pixel, we shoot a ray through this pixel, and determine the first object that is hit by the ray.

Ray Representation: Let us consider how rays are represented, generated, and how intersections are determined. First off, how is a ray represented? An obvious method is to represent it by

its origin point p and a directional vector $\vec{u}$. Points on the ray can be described *parametrically* using a scalar t:

$$R = \{p + t\vec{u} \mid t > 0\}.$$

Notice that our ray is *open*, in the sense that it does not include its endpoint. This is done because in many instances (e.g., reflection) we are shooting a ray from the surface of some object. We do not want to consider the surface itself as an intersection. (As a practical matter, it is good to require that t is larger than some very small value, e.g. $t \geq 10^{-3}$. This is done because of floating point errors.)

In implementing a ray tracer, it is also common to store some additional information as part of a *ray object*. For example, you might want to store the value t_0 at which the ray hits its first object (initially, $t_0 = \infty$) and perhaps a pointer to the object that it hits.

Ray Generation: Let us consider the question of how to generate rays. Let us assume that we are given essentially the same information that we use in gluLookAt and gluPerspective. In particular, let *eye* denote the eye point, *at* denote the center point at which the camera is looking, and let $\overrightarrow{up}$ denote the "up vector" for gluLookAt. Let $\theta_y = \pi \cdot \text{fovy}/180$ denote the y-field of view in radians. Let n_{rows} and n_{cols} denote the number of rows and columns in the final image, and let $\alpha = n_{cols}/n_{rows}$ denote the window's aspect ratio.

In gluPerspective we also specified the distance to the near and far clipping planes. This was necessary for setting up the depth buffer. Since there is no depth buffer in ray tracing, these values are not needed, so to make our life simple, let us assume that the window is exactly one unit in front of the eye. (The distance is not important, since the aspect ratio and the field-of-view really determine everything up to a scale factor.)

The height and width of view window relative to its center point are

$$h = 2\tan\frac{\theta_y}{2} \qquad w = h \cdot \alpha.$$

So, the window extends from $-h/2$ to $+h/2$ in height and $-w/2$ to $+w/2$ in width (see Fig. 63.)

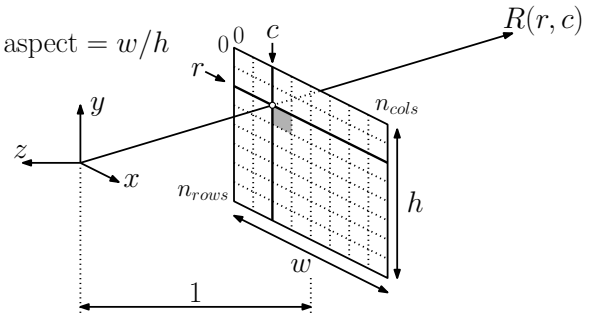


Fig. 63: Ray generation.

Next, we proceed to compute the viewing coordinate frame, very much as we did in our earlier lecture on 3-dimensional viewing. The origin of the camera frame is *eye*, the location of the

eye. The view vector is directed from the eye to the point we are looking at, that is,

$$\overrightarrow{view} \;=\; \text{normalize}(at - eye)$$

Recall from the earlier lecture that the unit vectors for the camera frame are:

$$V.\vec{v}_z \;=\; -\overrightarrow{view}, \qquad V.\vec{v}_x \;=\; \text{normalize}(\overrightarrow{view} \times \overrightarrow{up}), \qquad V.\vec{v}_y \;=\; (V.\vec{v}_z \times V.\vec{v}_x).$$

Next, we need to determine the point on the image plane corresponding to row r and column c. Since we are interested in using this for picking, we will follow the standard convention (used, e.g., by GLUT) that rows are indexed from top to bottom, and columns are indexed from left to right. Every point on the view window has $V.\vec{v}_z$ coordinate of -1. Now, suppose that we want to shoot a ray for row r and column c, where $0 \le r < n_{rows}$ and $0 \le c < n_{cols}$. Observe that r/n_{rows} is in the range from 0 to 1. Multiplying by $-h$ maps us linearly to the (inverted) interval $[0, -h]$ and then adding $h/2$ yields the final desired interval $[h/2, -h/2]$. This means that the vertical offset of row r is:

$$a_y \;=\; -h\frac{r}{n_{rows}} + \frac{h}{2} \;=\; -h\left(\frac{r}{n_{rows}} - \frac{1}{2}\right).$$

By an analogous reasoning, the horizontal offset of column c is

$$a_x \;=\; w\left(\frac{c}{n_{cols}} - \frac{1}{2}\right).$$

Therefore, the location of the point corresponding to row r and column c on the image plane is

$$p[r,c] \;=\; eye + a_x \cdot V.\vec{v}_x + a_y \cdot V.\vec{v}_y - V.\vec{v}_z.$$

(Since *eye* is a point and the other three components involve the product of a scalar and a vector, it follows that $p[r,c]$ is a valid affine equation for a point in space.) The direction vector for the ray emanates from the eye and passes through this point, thus, it is

$$\vec{u}[r,c] \;=\; \text{normalize}(p[r,c] - eye).$$

Thus, the desired ray $R[r,c]$ (see Fig. 63) has the origin *eye* and the directional vector $\vec{u}[r,c]$. In conclusion, the desired ray is:

$$R[r,c] \;=\; eye + t \cdot \vec{u}[r,c], \qquad (\text{for } t > 0).$$

Now that we have the desired ray, we next consider how to determine what it hits and where it hits it.

Rays and Intersections: Given an object in the scene, a *ray intersection procedure* determines whether the ray intersects and object, and if so, returns the value $t' > 0$ at which the intersection occurs. (This is a natural use of object-oriented programming, since the intersection procedure can be made a member function of the object.) Otherwise, if t' is smaller than the current t_0 value, then t_0 is set to t'. Otherwise the trimmed ray does not intersect the object. (For practical purposes, it is useful for the intersection procedure to determine two other quantities. First, it should return the normal vector at the point of intersection and second, it should indicate whether the intersection occurs on the inside or outside of the object. The latter information is useful if refraction is involved.)

Ray-Plane Intersection: Any plane in 3-dimensional space can be expressed as the set of points $p = (x, y, z)$ that satisfy a linear equation, that is

$$H : ax + by + cz + d = 0,$$

for some real-valued coefficients a, b, c, and d. For example, the plane containing the x, y-axes is the plane $z = 0$, which corresponds to the vector $(a, b, c, d) = (0, 0, 1, 0)$.

The question we wish to consider is, given a plane H, and given a ray $R : p + t\vec{u}$, does the ray hit the plane, and if so, where?

To determine the value of t where the ray intersect the plane, we could plug the ray's parametric representation into this equation and simply solve for t. If the ray is represented by $p + t\vec{u}$, then we have the equation

$$a(p_x + tu_x) + b(p_y + tu_y) + c(p_z + tu_z) + d = 0.$$

Soving for t we obtain

$$t_0 = -\frac{ap_x + bp_y + cp_z}{au_x + bu_y + cu_z}.$$

Note that the denominator is zero if the ray is parallel to the plane. We may simply assume that the ray does not intersect the polygon in this case (ignoring the highly unlikely case where the ray hits the polygon along its edge). Once the intersection value t_0 is known, the actual point of intersection is just computed as $p + t_0\vec{u}$. If the value of t_0 is negative at the intersection point, then the intersection lies behind the viewer's eye, and it may be ignored. Otherwise, we report that $p + t_0\vec{u}$ is the intersection point.

Normal Vector: For the sake of computing lighting, it is necessary to have a surface normal vector. In the case of a plane, this is very easy to compute. In particular, if the plane's equation is $ax + by + cz + d = 0$, then the vector (a, b, c) is orthogonal to the plane. (For example, in the case of the plane $z = 0$, the equation is given by the coefficient vector $(a, b, c, d) = (0, 0, 1, 0)$, and hence the vector $(0, 0, 1)$ is normal to the plane's surface, as expected.

Lecture 17: More on Ray Tracing: Reflection and Refraction

Ray Tracing: We continue our discussion of ray tracing. Recall that this is a powerful method for rendering highly realistic images. Unlike OpenGL, it implements a global model for image generation, based on tracing the rays of light, mostly working backwards from the eye to the light sources.

Last time we discussed how rays are represented, how to generate rays based on the camera setup, and how this can be used in the context of object picking in interactive computer graphics. We also discussed how to compute the intersection of a ray with a plane. Today we consider how to intersect a ray with a sphere, and other issues such as how to handle reflection and refraction.

Ray-Sphere Intersection: Let us consider how to solve the intersection of a ray with a sphere. Suppose that the sphere is represented by giving its center point c and its radius $r > 0$ (see Fig. 64(a)). Recall that the ray $R : (p, \vec{u})$ is represented by the origin point p of the ray and the unit directional vector $\vec{u}$. Any point on the ray can be described as $p + t\vec{u}$ for some $t > 0$. The intersection problem involves determining whether the ray hits the object and if so, what is the value t of the intersection point q (see Fig. 64(c)). For the sake of lighting, we also would like to have the surface normal vector $\vec{n}$ at the contact point.

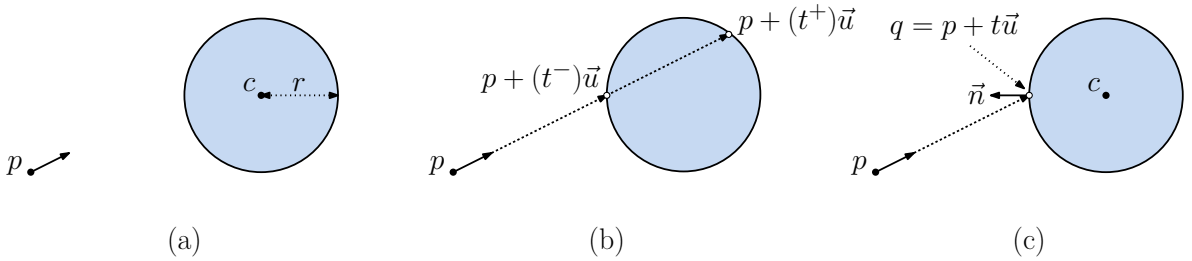


Fig. 64: Ray-sphere intersection.

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

$$\|(p + t\vec{u}) - c\| = r.$$

Notice that the quantity inside the $\|.\|$ above is a vector. Let $\vec{w} = c - p$. This gives us

$$\|t\vec{u} - \vec{w}\| = r.$$

We know $\vec{u}$, $\vec{w}$, and r and we want to find t. By the definition of length using dot products we have

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

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:

$$t^2(\vec{u} \cdot \vec{u}) - 2t(\vec{u} \cdot \vec{w}) + (\vec{w} \cdot \vec{w}) - r^2 = 0$$

This is a quadratic equation $at^2 + bt + c = 0$, where

$$\begin{aligned} a &= (\vec{u} \cdot \vec{u}) = 1 \qquad \text{(since } \vec{u} \text{ is normalized)}, \\ b &= -2(\vec{u} \cdot \vec{w}), \\ c &= (\vec{w} \cdot \vec{w}) - r^2 \end{aligned}$$

We can solve this using the quadratic formula to produce two roots. Let $\Delta = b^2 - 4ac$ denote the *determinant*. If $\Delta < 0$, then there is no root. Otherwise, using the fact that $a = 1$ we have:

$$\begin{aligned} t^- &= \frac{-b - \sqrt{b^2 - 4ac}}{2a} = \frac{-b - \sqrt{\Delta}}{2}, \\ t^+ &= \frac{-b + \sqrt{b^2 - 4ac}}{2a} = \frac{-b + \sqrt{\Delta}}{2}. \end{aligned}$$

These two values reflect the fact that the ray may hit the sphere twice (see Fig. 64(b)). The general rule in ray tracing is to *use the smaller of the two positive roots*. Thus, if $t^- > 0$ we use t^- to define the intersection point (as indicated in the figure). Otherwise, if $t^+ > 0$ we use t^+. (What does this imply about the origin of the ray?) If both are nonpositive, then there is no intersection. (Note that there are two ways that there may be no solution, either this or because the determinant is negative. These have two different geometric interpretations. What are they?)

Note that it is a not a good idea to compare floating point numbers against zero, since floating point errors are always possible. A good rule of thumb is to do all of these 3-d computations using doubles (not floats) and perform comparisons against some small value instead, e.g. "`const double TINY = 1E-3`". The proper choice of this parameter is a bit of "magic". It is usually adjusted until the final image looks okay.

(Note that this is not the most numerically accurate method for computing the intersection point. However, for most applications of ray tracing, the approximation is close enough that any errors are not noticeable to the human eye. A book on numerical analysis will discuss more accurate methods for solving the quadratic equation.)

Normal Vector: In the case of the sphere, that the normal vector is directed from the center of the sphere to point of contact. Thus, if t is the parameter value at the point of contact, the normal vector is just

$$\vec{n} = \text{normalize}(p + t\vec{u} - c).$$

Note that this vector is directed outwards. If t^- was used to define the intersection, then we are hitting the object from the outside, and so $\vec{n}$ is the desired normal. However, if t^+ was used to define the intersection, then we are hitting the object from the inside, and $-\vec{n}$ should be used instead.

Reflection: Recall the basic recursive structure to ray tracing. Shoot a ray and determine the color of the first object hit. Next, shoot a ray to the light sources to determine which of these sources illuminate this point. (If you hit another object before arriving at the light source then you are in the shadow cast by this object.) Based on the surface normal vector, apply some lighting model (e.g., the Phong model) to compute the shading at this point.

If the object hit is reflective or refractive, we next shoot additional rays (recursively) and determine their colors. (Note that lighting will be determined for these points at their points of contact, so these colors are not applied to lighting on the surface of the reflective object.) Finally, we blend the various colors together to obtain the final surface color, which is returned.

How is this blending done? Recall that in the Phong reflection model each object is associated with a color, and its coefficients of ambient, diffuse, and specular reflection, denoted ρ_a, ρ_d and ρ_s. To model the reflective component, each object will be associated with an additional parameter called the *coefficient of reflection*, denoted ρ_r. As with the other coefficients this is typically a number in the interval $[0, 1]$. Let us assume that this coefficient is nonzero. We compute the view reflection ray (which equalizes the angle between the surface normal and the view vector). Let $\vec{v}$ denote the normalized *view vector*, which points backwards along the viewing ray (see Fig. 65). Thus, if the ray is $p + t\vec{u}$, then $\vec{v} = -\text{normalize}(\vec{u})$. (This is essentially the same as the view vector used in the Phong model, but it may not

point directly back to the eye because of intermediate reflections.) Let $\vec{n}$ denote the outward pointing surface *normal vector*, which we assume is also normalized. The normalized *view reflection vector* (see Fig. 65), denoted $\vec{r}_v$, is derived as follows

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

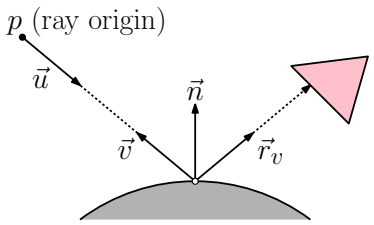


Fig. 65: Reflection.

Since the surface is reflective, we shoot the ray emanating from the surface contact point along this direction and apply the above ray-tracing algorithm recursively. Eventually, when the ray hits a nonreflective object, the resulting color is returned. This color is then factored into the Phong model, as will be described below. Note that it is possible for this process to go into an infinite loop, if say you have two mirrors facing each other. To avoid such looping, it is common to have a maximum recursion depth, after which some default color is returned, irrespective of whether the object is reflective.

Transparent objects and refraction: To model *refraction*, also called *transmission*, we maintain a coefficient of transmission, denoted ρ_t. We also need to associate each surface with two additional parameters, the *indices of refraction*[4] 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:

Material	Index of Refraction
Air (vacuum)	1.0
Water	1.333
Glass	1.5
Diamond	2.47

Snell's law says that if a ray is incident with angle θ_i (relative to the surface normal) then it will transmitted with angle θ_t (relative to the opposite normal) such that

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

[4]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, which is called *chromatic dispersion*. 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) it is not easy to model this phenomenon. For simplicity we assume that all wavelengths have the same index of refraction.

Let us work out the direction of the transmitted ray from this. As before let $\vec{v}$ denote the normalized view vector, directed back along the incident ray. Let $\vec{t}$ denote the unit vector along the transmitted direction, which we wish to compute (see Fig. 66(a)).

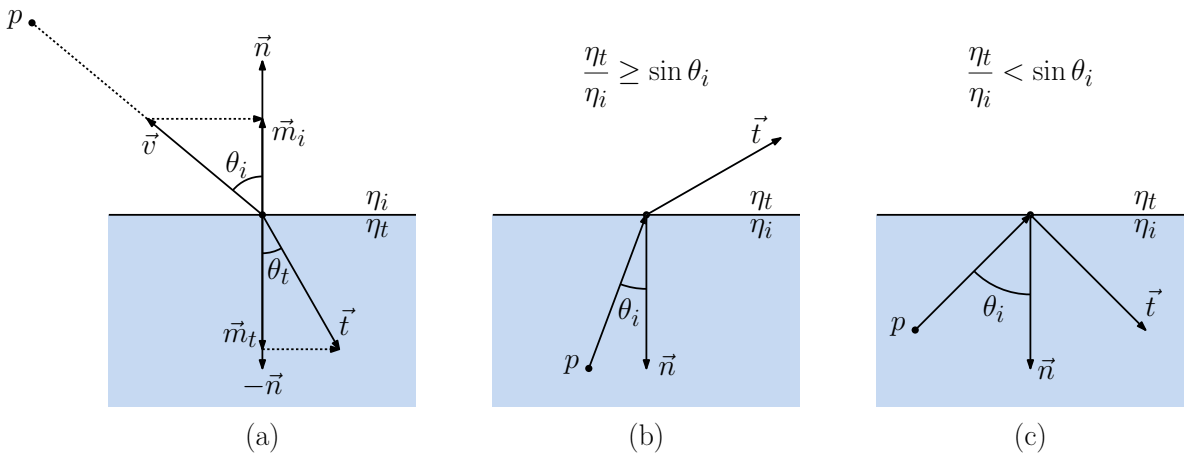


Fig. 66: Refraction. ((b) and (c) show the conditions under which total internal reflection might occur.)

The orthogonal projection of $\vec{v}$ onto the normalized normal vector $\vec{n}$ is

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

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

$$\overrightarrow{w}_i = \overrightarrow{m}_i - \vec{v}.$$

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

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

Since $\overrightarrow{w}_i$ and $\overrightarrow{w}_t$ are parallel we have

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

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

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

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

trigonometry:

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

What if the term in the square root is negative? This is possible if $(\eta_t/\eta_i) < \sin\theta_i$. In particular, this can only happen if $\eta_i > \eta_t$, 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. In this situation, *total internal reflection* takes place (see Fig. 66(c)). That is, the light source is not refracted at all, but is reflected back onto the incident side. (By the way, this phenomenon, combined with chromatic dispersion, is one of the reasons for the existence of rainbows.) When this happens, the refraction reduces to reflection and so we set $\vec{t} = \vec{r}_v$, the view reflection vector.

In summary, the transmission process is solved as follows.

(1) Compute the point where the ray intersects the surface. Let $\vec{v}$ be the normalized view vector, let $\vec{n}$ be the normalized surface normal at this point, and let η_i and η_t be the indices of refraction on the incoming and outgoing sides, respectively.

(2) Compute the angle of refraction:

$$\theta_t = \arccos\sqrt{1-\left(\frac{\eta_i}{\eta_t}\right)^2(1-(\vec{v}\cdot\vec{n})^2)}.$$

If the quantity under the square root symbol is negative, process this as internal reflection, rather than transmission.

(4) If the quantity under the square root symbol is nonnegative, compute the transmission vector

$$\vec{t} = \left(\frac{\eta_i}{\eta_t}\cos\theta_i - \cos\theta_t\right)\vec{n} - \frac{\eta_i}{\eta_t}\vec{v}.$$

The transmission ray is emitted from the contact point along this direction.

Lecture 18: Shadows and Stenciling

Shadows: Shadows give an image a much greater sense of realism. The manner in which objects cast shadows onto the ground and other surrounding surfaces provides us with important visual cues on the spatial relationships between these objects. As an example of this, imagine that you are looking down (say, at a 45° angle) at a ball sitting on a smooth table top. Suppose that (1) the ball is moved vertically straight up a short distance, or (2) the ball is moved horizontally directly away from you by a short distance. In either case, the impression in the visual frame is essentially the same. That is, the ball moves upwards in the picture's frame (see Fig. 67(a)).

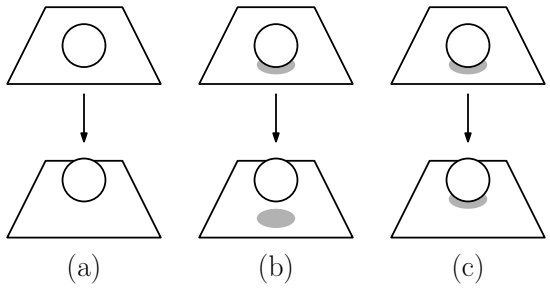


Fig. 67: The role of shadows in ascertaining spatial relations.

If the ball's shadow were drawn, however, the difference would be quite noticeable. In case (1) (vertical motion), the shadow remains in a fixed position on the table as the ball moves away. In case (2) (horizontal motion), the ball and its shadow both move together (see Fig. 67(b)).

We will consider various ways to handle shadows in interactive computer graphics: shadow painting, shadow mapping, and shadow volumes. Note that OpenGL does not support shadows directly (because shadows are global effects, while OpenGL uses a local lighting model).

Shadow Painting: Perhaps the simplest and most sneaky way in which to render shadows is to simply "paint" them onto the surfaces where shadows are cast. For example, suppose that a shadow is being cast on flat table top by some occluding object P. First, we compute the shape of P's shadow P' on the table top, and then we render a polygon in the shape of P' directly onto the table.

If P is a polygon and the shadow is being cast onto a flat surface, then the shadow shape P' will also be a polygon. Therefore, we need only determine the transformation that maps each vertex v of P to the corresponding shadow vertex v' on the table top. This process is illustrated in Fig. 68.

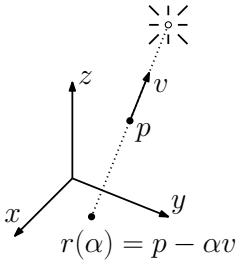


Fig. 68: The shadow projection transformation.

Let us consider a simple instance of this. Suppose that we model space using a coordinate system where the z-axis points up and the x, y-plane is the ground surface, onto which the shadow will be cast. Let us suppose that the light source is a point at infinity, given in (projective) homogeneous coordinates as $(v_x, v_y, v_z, 0)^\mathsf{T}$. This means that the direction to the light source is given by the vector $v = (v_x, v_y, v_z)^\mathsf{T}$.

To specify the ground, we observe that, generally, a plane in 3-dimensional space can be

specified by a equation of the form

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

In our case, the ground satisfies the equation $z = 0$ (that is, $a = b = d = 0$ and $c = 1$). (As an exercise, consider repeating the following derivation for an arbitrary plane.)

Given a vertex p of P, we want to imagine projecting a ray from the light source through p until it hits the ground. Such a ray has the directional vector $-v$ (since it is directed away from the light) and passes through p, and so an arbitrary point on this ray can be represented as

$$r(\alpha) = p - \alpha v,$$

where $\alpha \geq 0$ is any nonnegative scalar. This is a vector equation, and so we have

$$r(\alpha)_x \;=\; p_x - \alpha v_x \qquad r(\alpha)_y \;=\; p_y - \alpha v_y \qquad r(\alpha)_z \;=\; p_z - \alpha v_z.$$

We know that the shadow hits the ground at $z = 0$, and thus we have

$$0 \;=\; r(\alpha)_z \;=\; p_z - \alpha v_z,$$

from which we infer that $\alpha^* = p_z/v_z$. We can derive the x- and y-coordinates of the shadow point as

$$p'_x \;=\; r(\alpha^*)_x \;=\; p_x - \frac{p_z}{v_z} v_x \qquad \text{and} \qquad p'_y \;=\; r(\alpha^*)_y \;=\; p_y - \frac{p_z}{v_z} v_y.$$

Thus, the desired shadow point can be expressed as

$$p' \;=\; \left(p_x - \frac{v_x}{v_z} p_z,\; p_y - \frac{v_y}{v_z} p_z,\; 0 \right)^{\mathsf{T}}.$$

The shadow-projection matrix: It is interesting to observe that this transformation is an affine transformation of p. In particular, we can express this in matrix form, called the *shadow-projection matrix*, as

$$\begin{pmatrix} p'_x \\ p'_y \\ p'_z \\ 1 \end{pmatrix} \;=\; \begin{pmatrix} 1 & 0 & -v_x/v_z & 0 \\ 0 & 1 & -v_y/v_z & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix}.$$

This is nice, because it provides a particularly elegant mechanism for rendering the shadow polygon. The process is described in the following code block. The first drawing draws a projection of P on the ground in the shadow color, and the second one actually draws P itself. Note that this assumes that P is a constructed from polygons, but this assumption holds for for all OpenGL drawing.

The above shadow matrix works for light sources at infinity. You might wonder whether this is possible for light sources that are not at infinity. The answer is yes, but the problem is that the projection transformation is no longer an affine transformation. (In particular, the light

```
GLfloat shadowProjection[16];        // shadow projection matrix (4x4)
drawGround();                        // draw the ground
glPushMatrix();
    ... // disable lighting and set color to shadow color
    ... // enable transparency, so ground texture shows through shadow
    glMultMatrix(shadowProjection);
    drawObject();                    // draw the object's shadow
glPopMatrix();
... // restore lighting and set object's natural color
drawObject();                        // draw the object
```

rays are not parallel to each other, they converge at the light source.) If you think about this a moment, you will realize that this is exactly the issue that we faced with perspective projections. Indeed, the shadow projection is just an example of a projective transformation, where the light source is the camera and the ground is the image plane! (We will leave the solution as an exercise.)

Recall that in order to present your shadow projection matrix to OpenGL, it is represented by an array of 16 floats stored in column major order. Thus, the shadow matrix would be given to OpenGL in column-by-column order:

```
GLfloat shadowProjection[] = {    // shadow projection matrix
    1,      0,      0,      0,    // first column
    0,      1,      0,      0,    // second column
    -vx/vz, -vy/vz, 0,      0,    // third column
    0,      0,      0,      1     // fourth column
};
```

Depth conflicts with shadows: There are a couple of aspects of this process that need to be taken with some care. The first is the problem of hidden surface removal. If the shadow is drawn at exactly $z = 0$, then there will be competition in the depth buffer between the ground and the shadow. A quick-and-dirty fix for this is to nudge the shadow slightly off the ground. For example, store a small positive constant $\delta > 0$ in the third row, last column of the above matrix. This will force the z-coordinate of p' to be δ, which will perturb it to lie just above the ground. The choice of δ is a bit delicate, and depends on the general scale of your scene and the distance to the camera.

OpenGL provides a function that will achieve the same sort of perturbation to the depth values in order to avoid conflicts in the depth buffer. I will not discuss this, but if you are interested, check out the OpenGL command glPolygonOffset. If you decide to use this, you will also need to enable GL_POLYGON_OFFSET_FILL, and disable it when you are done.

Stenciling shadows: A second issue is that this will draw shadows on the the plane $z = 0$. But if we come to the edge of the ground surface, it will continue to draw the shadow, even if it falls off the edge of the ground surface (see Fig. 69(a)).

In order to avoid this problem, we can make use of a technique called *stenciling*. The idea is to first draw the ground top into the stencil buffer (see Fig. 69(b)). This buffer forms a sort

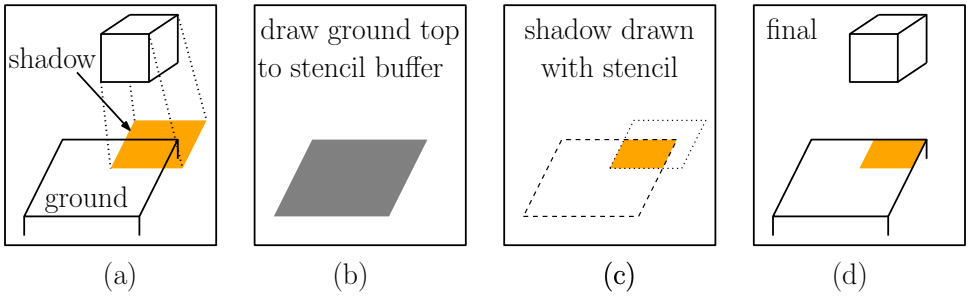


Fig. 69: Using the stencil buffer to limit shadow rendering.

of "mask" which can then be applied to limit future drawing. We then draw the shadow, but we enable *stenciling*, which means that any pixels that lie outside the stencil region are no drawn (see Fig. 69(c)). Finally, the rest of the scene is drawn. (Because the shadow is drawn a tiny bit higher than the ground top, it will survive in the depth buffer.)

The OpenGL Stencil Buffer: To begin, we need to understand something more about the *stencil buffer*, which (along with the color buffer and depth buffer) is one of the important buffers that OpenGL maintains. When stenciling is enabled in OpenGL, every pixel that is drawn is subjected to a *stencil test*. If the pixel fails the test it is discarded. On the other hand, if it passes the test, then it is sent along to the depth test. If it passes both of these tests, it will then be colored and rendered to the color buffer.

As with all things in OpenGL, there are a number of options for the user to play with. As we have described it above, it may seem that the stencil buffer is a boolean buffer. This is not quite true. Generally, the stencil buffer usually has a small number of bits (8 bits is typical). Whenever a pixel passes the stencil test, a stencil operation is performed. This operation may alter the contents of the stencil buffer. The effect that occurs is determined by something called the *stencil operation*. The stencil test and the stencil operation are items that you can control in order to achieve various effects. When using stenciling, the following steps are usually applied:

- Specify the desired stencil test (using glStencilFunc)
- Specify the desired stencil operation (using glStencilOp)
- Enable stenciling (using glEnable(GL_STENCIL_TEST))
- Draw your objects
- Disable stenciling (using glDisable(GL_STENCIL_TEST))

The first OpenGL function controls the stencil test:

```
glStencilFunc(func, ref, mask)
```

The values *ref* and *mask* are integer values, which are used to control the manner in which the test is performed. The *mask* is used for extracting a portion of the bits of interest to which to apply the stencil test. For example, if you have an 8-bit stencil buffer, you could adjust the map to perform as many as 8 different single-bit tests. Alternatively, you could have two 3-bit tests one 2-bit test, and so on. The value *ref* is called the reference value, and

it is what the comparison is based on. The *func* is one of the following: (Let $\wedge$ denote the boolean-and operator, that is, "&" in C++.)

- GL_ALWAYS: The pixel always passes the test
- GL_NEVER: The pixel always fails the test
- GL_LESS: The pixel passes if $(ref \wedge mask) < (stencil \wedge mask)$
- GL_LEQUAL: The pixel passes if $(ref \wedge mask) \leq (stencil \wedge mask)$
- GL_GREATER: Same, but for $>$
- GL_GEQUAL: Same, but for $\geq$
- GL_EQUAL: Same, but for $=$
- GL_NOTEQUAL: Same, but for $\neq$

The second OpenGL function controls the stencil operation:

glStencilOp(sfail, dfail, dpass)

where *sfail* indicates the action to be applied to the stencil buffer when the stencil test fails, *dfail* indicates the action to be applied to the stencil buffer when the stencil test passes but the depth test fails, and *dpass* indicates the action to be applied to the stencil buffer when both the stencil and depth tests pass. The possible values are:

- GL_KEEP: Keeps the current value in the stencil buffer
- GL_ZERO: Sets the stencil buffer value to 0
- GL_REPLACE: Sets the stencil buffer value to *ref*, as specified by glStencilFunc
- GL_INCR: Increments the current stencil buffer value. Clamps to the maximum representable unsigned value
- GL_DECR: Decrements the current stencil buffer value. Clamps to 0
- GL_INVERT: Bitwise inverts the current stencil buffer value

On first inspection, the number of possible combinations of tests and operations is truly bewildering. Indeed, there are very few combinations that are widely used in practice. The designers of OpenGL wanted this to be as flexible as possible, and did not give much consideration to intuitiveness.

Let's return to the question of how to apply stenciling in shadow painting. In the next code block, we show how to set up the stencil buffer to apply a mask that allows drawing only over the ground. We will make use of a handy function, glColorMask, which essentially disables writing to the color buffer. This guarantees that our drawing goes only to the stencil buffer.

Let's see in greater detail what each of the steps does in setting up the stencil buffer.

(a) glClear(...): Clears the color, depth, and stencil buffers (before drawing).

(b) glDisable(GL_DEPTH_TEST) and glColorMask$(0, 0, 0, 0)$: no drawing to the color or depth buffers, only to the stencil buffer.

(c) glStencilFunc(...) and glStencilOp(...): Every fragment (pixel) in the subsequent drawing (namely the ground) that passes the depth test will be stored as the value 1 in the stencil buffer. Portions of the ground that do not appear in the final image (because they fail the depth test) do not change the stencil buffer. Thus, every pixel that shows up as ground in the final image has a 1 stored in the stencil buffer, and all other pixels are 0.

Setting up the Stencil Buffer for Shadow Painting

```
glClear(GL_STENCIL_BUFFER_BIT);   // clear the stencil buffer contents
// ...
glDisable(GL_DEPTH_TEST);         // disable drawing to depth buffer
glColorMask(0, 0, 0, 0);          // disable drawing to color buffer
glStencilFunc(GL_ALWAYS, 1, 1);   // set reference value to 1
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE); // if visible, write 1 to stencil
glEnable(GL_STENCIL_TEST);        // enable stenciling
  drawGround();                   // draw ground to stencil buffer
glDisable(GL_STENCIL_TEST);       // disable stenciling
glEnable(GL_DEPTH_TEST);          // restore regular drawing
glColorMask(1, 1, 1, 1);
```

(d) glEnable(GL_STENCIL_TEST) and drawGround(): Draw ground with stenciling enabled. When done, every pixel that corresponds to a point on the ground will be associated with a 1 in the stencil buffer, and all others will be associated with the value 0.

(e) glDisable(GL_STENCIL_TEST), glEnable(GL_DEPTH_TEST), glColorMask$(1, 1, 1, 1)$: Turn off stenciling and turn back on regular (color and depth) drawing.

Now that we have set up the stencil buffer as we want it, we are ready to draw the shadows. We want the shadows to appear only where the stencil buffer contains the value 1. This is shown in the code fragment below. Whenever we paint a shadow pixel we zero-out the stencil buffer at this position (using the option GL_ZERO whenever a pixel is written). As we shall see below, this is done because shadows are transparent, and if we were to write the same pixel more than once (which can happen) the resulting pixels would appear darker than pixels that are written only once.

Paint the Shadow using the Stencil Buffer

```
glStencilFunc(GL_EQUAL, 1, 1);           // draw only if stencil value = 1
glStencilOp(GL_KEEP, GL_KEEP, GL_ZERO);  // zero out stencil after writing
glEnable(GL_STENCIL_TEST);               // enable stenciling
  drawObjects();                         // draw objects that cast shadows
glDisable(GL_STENCIL_TEST);              // disable stenciling
```

Let's see in greater detail what each of the above steps does in the shadow drawing code fragment. We assume that the ground surface has already been drawn.

(a) glStencilFunc(...) and glStencilOp(...): For every pixel of the subsequent drawing (the shadow) we check whether the stencil value equals the references value of 1. If it is equal, we draw it. No matter what happens, we will keep the value in the stencil buffer unchanged.

(b) glEnable(GL_STENCIL_TEST) and drawShadow(): Draw the shadow on the ground with the stencil enabled. Only shadow pixels that lie on the ground will be drawn.

(c) glDisable(GL_STENCIL_TEST): Turn off stenciling and return to normal drawing.

Blending Shadow and Ground Colors: Shadow painting takes a rather backwards approach to the generation of shadows. You first render the scene with lighting, and then apply the shadow on top. (A more natural approach would be to first render the scene without lighting and then apply the lighted object color on top of this.) As a consequence of this backwards approach, when drawing the shadow, you want to blend it with the color underneath.

In OpenGL, this is done using *blending*. Normally, when drawing the fragment that is closest to the viewer provides the entire color for the corresponding pixel. If the depth test is disabled, then the last pixel to be drawn covers all previous pixel colors. This is like a painter laying down layers of color, where the last layer is the only one that shows. When blending is enabled, you can imagine that the painter is working with translucent paints, and each coat of paint he applies is blended with the existing color, so that the final pixel color is some combination of the two colors.

Let us suppose that we have either disabled the depth test, or (as described above), we have offset the next layer so that it appears slightly above the existing layer. Suppose that we are painting one layer (e.g., the shadow) on top of an existing layer (e.g., the lit ground). In OpenGL terminology, the new fragment being drawn is called the *source*, and the fragment that is already in the color buffer is called the *destination*. Let C_s denote the new source color and let C_d denote the existing destination color. In order to specify the degree of blending, we recall that OpenGL represents colors using RGBA, where each component lies in the interval $[0, 1]$. We will use the A-component (or α-component) to express how transparent the new color. When $\alpha = 0$, the new color is completely transparent and has no effect, and when $\alpha = 1$, the new color is completely opaque and overwrites the existing color. When $\alpha = 1/2$, we will get a 50-50 blend of the two colors. That is, the final color C_f can be expressed as:

$$C_f \;=\; \alpha \cdot C_s + (1 - \alpha) \cdot C_d.$$

(Note that the α value is that of the source. The α value of the destination is ignored.)

OpenGL provides many different ways to combine colors (based on the α values of the source and destination fragments), but we will just consider this one example. This is specified by the command:

```
glBlendFunc(GLenum source_factor, GLenum dest_factor)
```

The source_factor indicates how the source fragment affects the color and the dest_factor indicates how the destination fragment affects the color. In order to perform this sort of blending, we will multiply the source color by the source alpha, and we will multiply the destination color time 1 minus the source alpha. To do this, we will use the following command:

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
```

Finally, when we specify our shadow color, we will use a dark shade of gray, but with a fractional α value. For example, RGBA $= (0.1, 0.1, 0.1, 0.3)$. The final blending is shown in the following code block.

Putting it all together: We now have all the pieces that we need to perform shadow painting. The two functions that you need to provide are:

```
// ... draw the ground using the lit color
glEnable(GL_BLEND);                 // enable color blending
                                    // standard transparency blending
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
                                    // color for shadows
const GLfloat SHADOW_COLOR[] = {0.1f, 0.1f, 0.1f, 0.3f};
drawMyShadows(SHADOW_COLOR);        // draw shadows
glDisable(GL_BLEND);                // disable blending
```

- drawGroundTop(): Draws just the top surface of the ground surface (the region of the stencil buffer where shadows may be drawn).
- drawObects(): Draws the objects that might cast shadows onto the ground.

This code assumes that the ground itself and the objects have already been drawn. It only draws the shadows. This is given in the following code block.

Lecture 19: More on Shadows: Shadow Maps and Shadow Volumes

Recap: Last time we introduced shadows as a method for enhancing realism. We discussed how to compute the shadow projection matrix and a simple mechanism for painting shadows on to surfaces. While this method is capable of casting shadows of very complex objects, it has the limitation that it can cast them onto a single planar surface (because the shadow projection transformation works for only a single plane). In this lecture we will consider other methods for enhanced lighting and shadowing.

Shadow Maps: Shadow mapping and its variants is perhaps the most popular method of generating shadows for interactive graphics systems. This technique involves a creative combination of texture mapping and the depth buffer. Indeed, it is based on creating a texture map that stores depth values, and then using the depth test to determine which pixels are in the light and which are in the shadow. The method's principal advantages are that it can handle both complex surfaces that cast shadows and complex surfaces onto which the shadows are cast. Its principal disadvantage is that employs image-based methods for representing shadows, and so the boundary between the shadow and lit areas will exhibit a staircase effect. (In the other two methods we discuss, shadow painting and shadow volumes, shadows are represented as accurately as are the object geometries.) It is possible to apply a bit of blurring to minimize these jagged artifacts.

To understand the technique, recall the shadow mechanism that is used in ray tracing. Recall that in ray tracing, in order to determine whether a surface point is lit or in shadow, we shoot a ray from this point towards the light source. (We will assume there is a single point light source, but the method can be generalized to work with multiple light sources, but each light source requires an additional redraw cycle.) If the ray shot to the light source reaches the light before hitting any other object, then the object is illuminated, and otherwise it is in the shadow. Unfortunately, ray tracing involves gloabl reasoning about the scene, so it is unclear how to apply this idea in the context of OpenGL.

```
                                    // color for shadows
const GLfloat SHADOW_COLOR[] = {0.1f, 0.1f, 0.1f, 0.3f};
GLfloat shadowProjection[16];       // shadow projection matrix
// ... compute shadow projection matrix
// ... remember that OpenGL stores matrices in column-major order

glClear(GL_STENCIL_BUFFER_BIT);   // clear the stencil buffer contents
//
// Set up the stencil buffer
//
glDisable(GL_DEPTH_TEST);           // suspend drawing
glColorMask(0, 0, 0, 0);
glStencilFunc(GL_ALWAYS, 1, 1);     // set reference value to 1
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE); // store 1 for each pixel
glEnable(GL_STENCIL_TEST);          // enable stenciling
  drawGroundTop();                  // draw just the top of ground
glDisable(GL_STENCIL_TEST);         // disable stenciling
glEnable(GL_DEPTH_TEST);            // restore drawing
glColorMask(1, 1, 1, 1);
//
// Paint the shadows using blending and the stencil buffer
//
glStencilFunc(GL_EQUAL, 1, 1);      // draw if stencil value = 1
glStencilOp(GL_KEEP, GL_KEEP, GL_ZERO); // zero stencil after write
glDisable(GL_LIGHTING);             // disable lighting
glColor4fv(SHADOW_COLOR);           // set shadow color
glPushMatrix();
glMultMatrixf(shadowProjection);    // shadow projection matrix
glEnable(GL_BLEND);                 // enable color blending
                                    // standard transparency blending
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glEnable(GL_STENCIL_TEST);          // enable stenciling
  drawObjects();                    // draw objects that cast shadows
glDisable(GL_STENCIL_TEST);         // disable stenciling
glDisable(GL_BLEND);                // disable blending
glPopMatrix();
glEnable(GL_LIGHTING);              // restore lighting
```

Here is where the depth buffer comes to the rescue. Suppose that we were to draw the entire scene from the perspective of a camera located at the light source, and store the result in a depth buffer D (see Fig. 70(a)). All the surfaces represented in the depth buffer are to be lit, and all other surfaces are in shadow (with respect to this light source). In particular, consider an arbitrary point p on some surface that is visible to the viewer (see Fig. 70(b)). Let us convert p's coordinates so that, rather than being represented in the world (i.e., modelview) coordinate system, it is instead represented in the light source's coordinate system. Now, project this point onto the view plane associated with the camera associated with the light source. Suppose that its projection falls on the pixel at row r and column c of the light source's viewing window. Let δ_p denote p's distance from the light source relative to the light source's frame. If $\delta_p > D[r, c]$, then p is not the closest object to the light source, and therefore p is in the shadow. (This is the case for point p' in Fig. 70(b).) On the other hand, if $\delta_p = D[r, c]$, then p is illuminated by the light source (This is the case for point p in Fig. 70(b).).

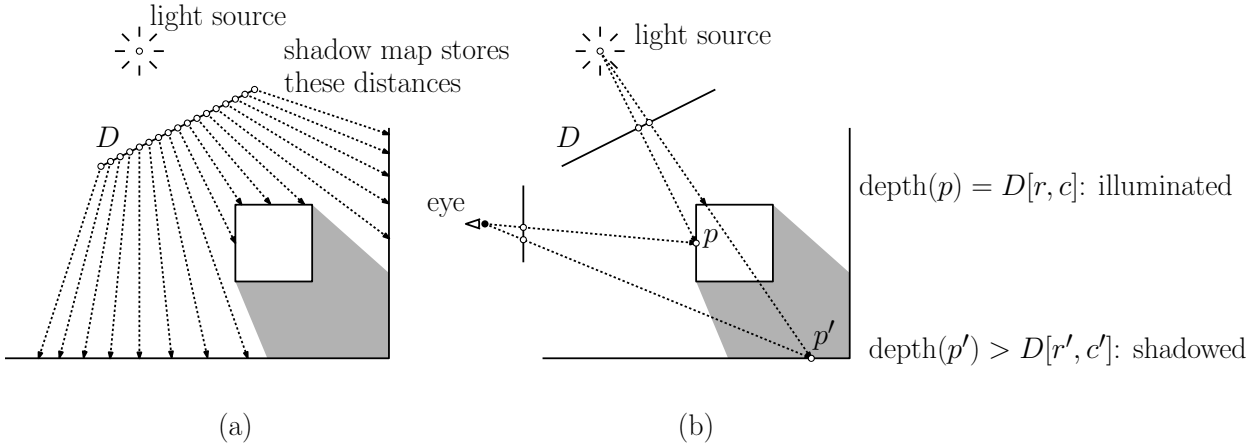


Fig. 70: Building and using shadow maps.

The process is performed in three steps:

(1) Render the entire scene by making the light source the "eye" for the view frame. Copy the contents of the depth buffer to an auxiliary buffer for use in phase (3). Call this buffer D.

(2) Render the scene from the viewer's perspective, but using only the ambient light. This draws the world as if it were all in shadows.

(3) Set the light to the intensity of the actual light source. Now, render the scene again from the viewers perspective, but with one important change. Each time we attempt to draw a point p, convert this point from its representation in the world coordinate system to its representation relative to the light source. Project this point onto the view plane for the light source (which corresponds to the view plane where D is stored). If it fails the depth test (that is, if its depth with respect to the light source is greater than the depth of the corresponding pixel stored in the depth buffer), then discard this point. Otherwise, draw the point p using the brighter intensity of the light source.

Getting this to work in OpenGL involves a amazing degree of cleverness in the way that the various buffers are processed. It would be too complex to go into how this is done here. If you would like to see a concrete example, you might check out the tutorial found on "Paul's Projects."

http://www.paulsprojects.net/tutorials/smt/smt.html

Shadow Volumes: Together with shadow maps, this is among the most popular methods for real-time rendering of shadows. The technique is based on a clever use of the stencil buffer and front/back-face culling, in order to identify areas of the scene (in three-dimensional space) that are visible to the light sources. The areas that are not visible to the light source are called *shadow volumes* (see Fig. 71(a)). Once the regions that lie outside the shadow volumes have been identified, these areas are then rendered with full lighting. The method makes quite sophisticated use of the stencil buffer to selectively draw the surfaces outside the shadow volumes.

Unlike shadow painting, this method can cast shadows onto to complex (possibly curved) surfaces. Unlike shadow maps, this method produces highly accurate shadows. Its main shortcoming is that its computational complexity grows with the number of edges in the objects that cast shadows. Therefore, it is only good for casting shadows of simple polygonal objects. (Thus, this method is somewhat dual to shadow painting. While shadow painting can cast complex shadows onto simple surfaces, shadow volumes can cast simple shadows onto complex surfaces.)

Let us begin by considering what a shadow volume is. Given a point light source L and an occluding object O, a *shadow volume* is the region of 3-D space occluded from the light source by O (see Fig. 71(a)). The shadow rendering problem therefore reduces to determining, for each surface S of our scene, which portions of S lie inside the shadow volume (and hence are not lit) and which lie outside the shadow volume (and hence are lit) (see Fig. 71(b)).

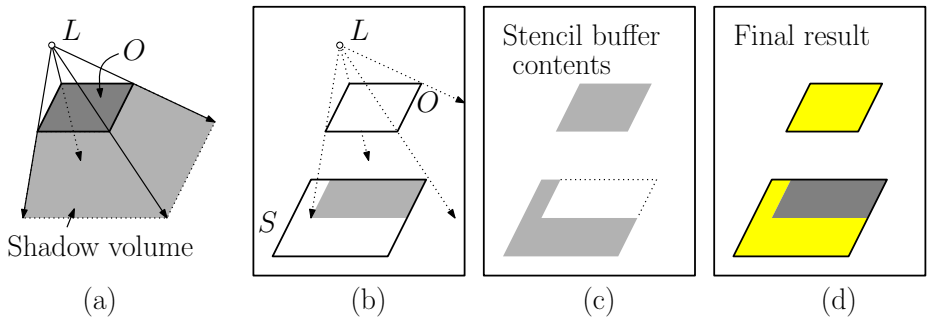


Fig. 71: Shadow volumes.

The basic shadow-volume method works as follows. Don't worry now if you do not see how to implement these two steps. We will discuss this below. First, render the scene as if it were completely in shadow (e.g., using only ambient light). Then, for each light source:

- Using the depth information for the scene, construct a mask in the stencil buffer that draws only the regions corresponding to points of the surface that are *not* in the shadow (see Fig. 71(c)).

- Render the scene again as if it were completely lit, but use the stencil buffer to mask away the shadowed areas (see Fig. 71(d)). (You can use additive blending along with RGBA color to successively add additional layers of colors to the scene.)

The question, then, is how to set up the stencil buffer so that it contains the desired region corresponding to where the light hits the object? Here are more details regarding how the stencil buffer is used to create shadow maps.

- Render the faces of the shadow volumes to the stencil buffer. (They do not appear in the color or depth buffer.)
- Each pixel of the stencil buffer maintains a counter (see Fig. 72).
  - Whenever we are moving from light into shadow, we increment the counter.
  - Whenever we are moving from shadow into light, we decrement the counter.
  - If the final counter value is 0, then this pixel is in the light.
- We draw the lit scene, but filter drawing so that pixels only appear if the corresponding stencil-buffer value is 0.

Shadow Volumes in OpenGL: The particularly nice aspect of the above strategy is that it can be implemented effieicntly in OpenGL (assuming that the shadow volumes are not too complex). It makes use of OpenGL's facilities for stenciling and culling of both back and front faces. Before explaining how the process is implemented, let us give a definition. Given an edge e of O, we define the corresponding *wall* of the shadow volume to be the truncated pyramid polygonal region defined by shooting rays from the light source L through any one edge of the occluding object O. (For example, in Fig. 71(a) above, there are four walls, one for each edge of O.) Think of these as the walls that bound the sides of the shadow volume.

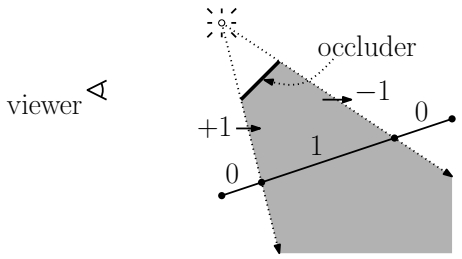


Fig. 72: Using stencil-buffer counters to implement shadow volumes.

Here is the process, broken down into steps that OpenGL can deal with.

(0) Draw your scene with only ambient light, and clear the stencil buffer to value 0.

(1) Disable writes to the depth and color buffers. (We will modify only the stencil buffer.)

(2) Enable back-face culling: glCullFace(GL_BACK); glEnable(GL_CULL_FACE)

(3) Set the stencil operation to *increment* the stencil buffer value whenever the depth-test passes. (This means that, for each front shadow wall that falls in front some object, we will increment the stencil buffer value.): glStencilOp(GL_KEEP, GL_KEEP, GL_INCR)

(4) Render the walls of the shadow volumes. (Due to culling, only front faces are rendered.)

(5) Enable front-face culling: glCullFace(GL_FRONT)

(6) Set the stencil operation to *decrement* the stencil buffer value whenever the depth-test passes. (This means that, for each back shadow wall that falls in front some object, we will increment the stencil buffer value.): glStencilOp(GL_KEEP, GL_KEEP, GL_DECR)

(7) Render the shadow volumes again. (Due to culling, only back faces are rendered.)

(8) At this point, a stencil value of 0 means that a point of the scene is illuminated. Reenable the depth and color buffers, and draw your scene (filtered to pixels where the stencil value is 0).

Wow! That is a definitely a pretty sophisticated use of OpenGL's buffering and rendering capabilities. To see how it works, consider the example shown in Fig. 73 of a single light source, a single occluder, and the rendering of a surface S.

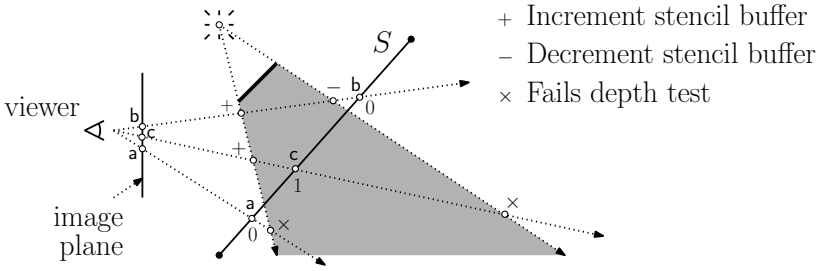


Fig. 73: Shadow volume example.

- In the case of pixel a, both of the shadow volume walls are farther away from the surface S to eye, and thus the stencil buffer value is not changed. Since it is initially 0, a is illuminated.
- In the case of pixel b, both of the shadow walls are closer than the surface S, and hence they are both considered. The front wall is rendered in step (4) and increments the stencil buffer. The back wall is rendered in step (5) and so the stencil value is decremented. The final stencil value is 0, and so b is illuminated.
- In the case of pixel c, the front shadow wall is closer but the back shadow wall fails the depth test. Thus, the stencil buffer value is incremented in step (4). Since it's value is greater than 1, this point will not be drawn in step (8), and so it will not be illuminated.

Lecture 20: Curved Models and Bézier Curves

Geometric Modeling: Geometric modeling is a key element of computer graphics. Up until now, we have considered only very simple 3-dimensional models, such as triangles and other polygons, 3-dimensional rectangles, planar surfaces, and simple implicit surfaces such as spheres. Generating more interesting shapes motivates a study of the methods used for representing objects with complex geometries.

Geometric modeling is fundamentally about representing 3-d objects efficiently, in a manner that makes them easy to design, visualize, and modify. In computer graphics, there is almost no limit to the things that people would like to model. Including:

Natural objects: Trees, flowers, rocks, water, fire, smoke, clouds

Human and animals: Skeletal structure, skin, hair, facial expressions

Architecture: Walls, doors, windows, furniture, pipes, railing

Manufacturing: Automobiles, appliances, weaponry, fabric and clothing

Non-geometric entities: Lighting, textures, surface materials

Because the things that we would like to model are so diverse, many different shape representation methods have emerged over the years. These include the following:

Boundary: Represent objects by their 2-dimensional surfaces. Methods tend to fall into one of two categories, *implicit* (blobs and metaballs) or *parametric* (Bézier surfaces, B-Splines, NURBS).

Volumetric: Represent complex objects through boolean operations on simple 3-dimension objects. For example, take a rectangular block and subtract cylindrical hole. This is popular in computer-aided design with machined objects. It is called *constructive solid geometry* (CSG).

Procedural: Objects are represented by invoking a procedure that generates the objects. This is used for very complex natural objects, that are not easily described by mathematical formulas, like mountains, trees, or cloudes. Examples include particle systems, fractals, and physically-based models.

Today, we will talk about boundary representations.

Boundary Representations: The most common way to represent a 3-dimensional object is to describe its boundary, that is, the 2-dimensional "skin" surrounding the object. These are called *boundary representations*, or *B-reps* for short. Boundary models can be formed of either smooth surfaces or flat (polygonal) surfaces. Polygonal surfaces most suitable for representing geometric objects with flat side, such as a cube. However, even smooth objects can be approximated by "gluing" together a large number of small polygonal objects into a *polygonal mesh*. This is the approach that OpenGL assumes. Through the use of polygonal mesh models and smooth shading, it is possible to produce the illusion of a smooth surface. Even when algebraic surfaces are used as the underlying representation, in order to render them, it is often necessary to first convert them into a polygonal mesh.

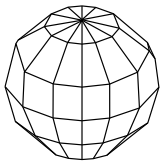


Fig. 74: Polygonal mesh used to represent a curved surface.

There are a number of reasons why polygonal meshes are easier to work with than curved surfaces. For example, the intersection of two planar surfaces is (ignoring special cases) a line. Generally when intersecting curved surfaces the curve along which they intersect may be quite hard to represent.

Curved Models: Smooth surface models can be broken down into many different forms, depending on the nature of the defining functions. The most well understood functions are *algebraic functions*. These are polynomials of their arguments (as opposed, say to trigonometric functions). The *degree* of an algebraic function is the highest sum of exponents. For example $f(x, y) = x^2 + 2x^2y - y$ is an algebraic function of degree 3. The ratio of two polynomial functions is called a *rational function*. These are important, since perspective projective transformations (because of perspective normalization), map rational functions to rational functions.

OpenGL only supports flat (i.e., polygonal) objects (or equivalently, algebraic functions of degree one). Although polygonal models are fundamental to OpenGL, they are not at all an easy method with which to design and manipulate smooth solid models. Most modeling systems allow the user to define objects at a higher-level (e.g., as a curved surface), and then procedures will be provided to break them down into polygonal meshes, for processing by OpenGL.

Implicit representation: In this representation a curve in 2-d and a surface in 3-d is represented as the zeros of a formula $f(x, y, z) = 0$. For example, the representation of a sphere of radius r about center point c is:

$$f(x, y, z) = (x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 - r^2 = 0.$$

It is common to place some restrictions on the possible classes of functions, for example, they must be algebraic or rational functions.

Implicit representations are nice for determining whether a point is inside, outside, or on the surface (just evaluate f). However, other tasks (e.g., generating a mesh) may not be as easy.

Implicit functions of constant degree are fine for very simple models (e.g., spheres, cylinders, cones), but they cannot represent very complex objects. There are methods (which we will not discuss) for creating complex objects from many simple (low-degree) implicit functions. These have colorful names like *blobs* and *metaballs*.

Parametric representation: In this representation the (x, y)-coordinates of a curve in 2-d is given as three functions of one parameter $(x(u), y(u))$. Similarly, a two-dimensional surface in 3-d is given as function of two parameters $(x(u, v), y(u, v), z(u, v))$. An example is the parametric representation of a sphere, which we have seen earlier in our discussion of texture mapping. For example, a sphere of radius r at center point c could be expressed parametrically as:

$$\begin{aligned} x(\theta, \phi) &= c_x + r \sin\phi \cos\theta \\ y(\theta, \phi) &= c_y + r \sin\phi \sin\theta \\ z(\theta, \phi) &= c_z + r \cos\phi, \end{aligned}$$

for $0 \le \theta \le 2\pi$ and $0 \le \phi \le \pi$. Here ϕ roughly corresponds to latitude and θ to longitude. Notice that this is *not* an algebraic representation, since it involves the use of trigonometric functions.

Note that parametric representations can be used for both curves and surfaces in 3-space (depending on whether 1 or 2 parameters are used).

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 because they are easy to subdivide into small patches for rendering, and hence they are popular in graphics. Sometimes (in fact, quite rarely) it is possible to convert from one representation to another. We will concentrate on parametric representations in this lecture.

Continuity: Consider a parametric curve $P(u) = (x(u), y(u), z(u))^{\mathsf{T}}$. An important condition that we would like our curves (and surfaces) to satisfy is that they should be as smooth as possible. This is particularly important when two or more curves or surfaces are joined together. We can formalize this mathematically as follows. We would like the curves themselves to be continuous (that is not making sudden jumps in value). If the first k derivatives (as function of u) exist and are continuous, we say that the curve has *kth order parametric continuity*, denoted C^k *continuity*. Thus, 0th order continuity just means that the curve is continuous, 1st order continuity means that tangent vectors vary continuously, and so on. This is shown in Fig. 75

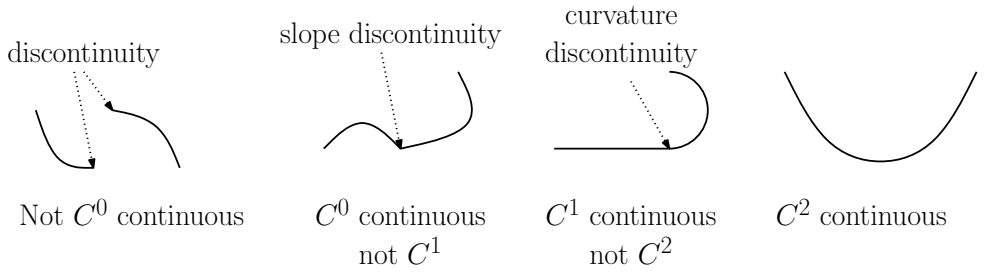


Fig. 75: Degrees of continuity.

Note that this definition is dependent on the particular parametric representation used. Since the same curve may be parameterized in different, some people suggest that a more appropriate definition in some circumstances is *geometric continuity*, denoted G^k, which depends solely on the shape of the curve, and not on the parameterization used.

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

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

It might seem most natural to have the curve pass through the control points, that is to *interpolate* between these points. There exists such an interpolating polygon, called the *Lagrangian interpolating polynomial*. However there are a number of difficulties with this

approach. For example, suppose that the designer wants to interpolate a nearly linear set of points. To do so he selects a sequence of points that are very close to lying on a line. However, polynomials tend to "wiggle", and as a result rather than getting a line, we get a wavy curve passing through these points (see Fig. 76).

Fig. 76: Curves based on interpolation versus approximation.

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

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

Observe that this is a weighted average of the points, and for any value of u, the two weighting or *blending functions* u and $(1-u)$ are nonnegative and sum to 1. That is, for any value of u the point $\mathbf{p}(u)$ is a convex combination of the control points.

Three control points: Linear interpolation is a concept that we have seen many times. The question is how to go from an interpolation process that involves two points to one that involves three, or four, or more control points. Certainly, we could linear interpolate from $\mathbf{p}_0$ to $\mathbf{p}_1$, and then from $\mathbf{p}_1$ to $\mathbf{p}_2$, and so on, but this would just give us a polygonal curve—not very smooth.

A mathematician named Paul de Casteljau, who was working for a French automobile company, came up with a very simple and ingenious method for generalizing linear interpolations to an arbitrary number of control points through a process of *repeated linear interpolation*.

To understand de Castlejau's idea, let us consider the case of three points. We want a smooth curve approximating them. Consider the line segments $\overline{\mathbf{p}_0\mathbf{p}_1}$ and $\overline{\mathbf{p}_1\mathbf{p}_2}$. From linear interpolation we know how to interpolate a point on each, say:

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

(See the middle figure in Fig. 77).

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

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

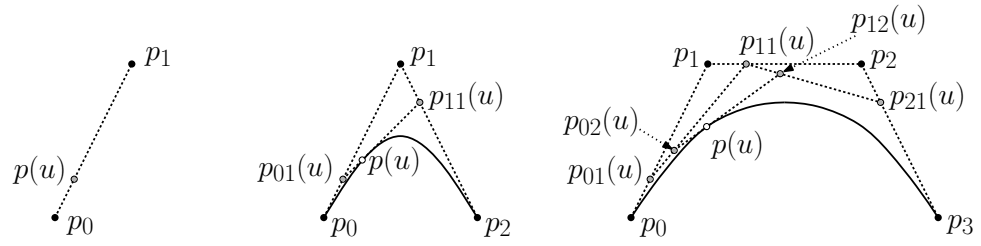


Fig. 77: Curves via repeated interpolation.

This is a algebraic parametric curve of degree two. This idea was popularized by an engineer, who was working for a rival French automobile company, named Pierre Bézier. In particular, the resulting curve is called the *Bézier curve* of degree two.

Observe that the function involves a weighted sum of the control points using the following *blending functions*:

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

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

An example of the resulting curve is shown in Fig. 78 on the left.

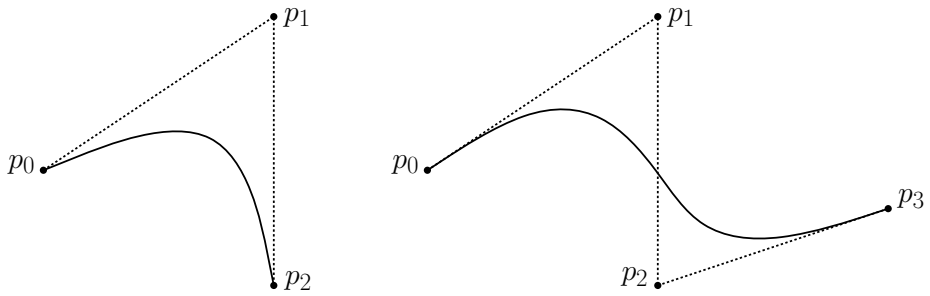


Fig. 78: Bézier curves for three and four control points.

Four control points: Let's carry this one step further. Consider four control points $\mathbf{p}_0$, $\mathbf{p}_1$, $\mathbf{p}_2$, and $\mathbf{p}_3$. First use linear interpolation between each pair yielding the points $\mathbf{p}_{01}(u)$ and $\mathbf{p}_{11}(u)$ and $\mathbf{p}_{21}(u)$ as given above. (The notation is rather messy. If you look at the right part of Fig. 77, you can see the pattern much more clearly. Simply draw line segments between each consecutive pair of control points and interpolate along each one.)

Now, we have gone from the initial four control points to three interplated control points. What now? Just repeat the three-point process! By linearly between these three points we have

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

Now we are down to just two points. Finally interpolate these $(1-u)\mathbf{p}_{02}(u) + u\mathbf{p}_{12}(u)$. This gives the final point on the curve for this value of u.

Expanding everything yields (trust me):

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

This is a polynomial of degree three, called the Bézier curve of degree three. Observe that the formula has the same form as the one above. It involves a blending of the four control points. The blending functions are:

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

It is easy to verify that, for any value of u, these blending functions are all nonnegative and sum to 1. That is, they form a convex combination of the control points. In this case, the blending functions are

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

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

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

$$b_{ik}(u) = \binom{k}{i}(1-u)^{k-i}u^i, \qquad \text{where} \qquad \binom{k}{i} = \frac{k!}{i!(k-i)!}.$$

These polynomial functions are important in mathematics, and are called the *Bernstein polynomials*, and are shown in Fig. 79 over the range $u \in [0,1]$.

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

Recall that computing the derivative of parametric curve gives you a tangent vector. (You may recall that we used this approach for computing normal vectors.) If you compute the

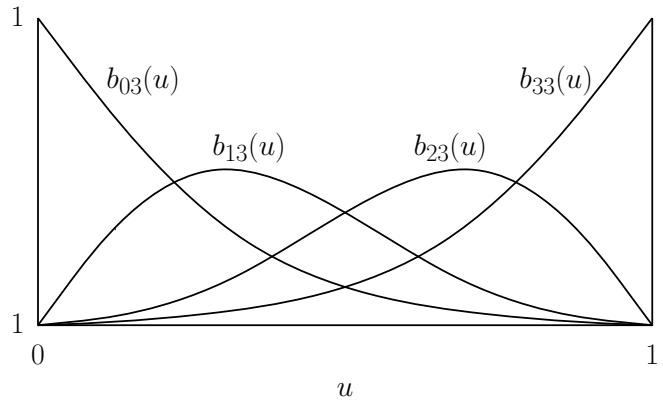


Fig. 79: Bézier blending functions (Bernstein polynomials) of degree 3. (Not drawn to scale.)

derivative of the curve with respect to u, you will discover to your amazement that the result is itself a Bézier curve. (That is, if you treat each tangent vector as if it were a point, these points would trace out a Bézier curve. Thus, the parameterized tangent vector of a Bézier curve is a Bézier curve.

Finally the Bézier curve has the following *variation diminishing property*. Consider the polyline connecting the control points. Given any line ℓ, the line intersects the Bézier curve no more times than it intersects this polyline. Hence the sort of "wiggling" that we saw with interpolating polynomials does not occur with Bézier curves.

Lecture 21: Bézier Surfaces and B-splines

Subdividing Bézier curves: Last time we introduced the mathematically elegant Bézier curves. Before going on to discuss surfaces, we need to consider one more issue. In order to render curves or surfaces using a system like OpenGL, which only supports rendering of flat objects, we need to approximate the curve by a number of small linear segments. Typically this is done by computing a sufficiently dense set of points along the curve or surface, and then approximating the curve or surface by a collection of line segments or polygonal patches, respectively.

Bézier curves (and surfaces) lend themselves to a very elegant means of recursively subdividing them into smaller pieces. This is nice, because if we want to render a curve at varying resolutions, we can perform either a high number or low number of subdivisions. Furthermore, if part of the surface is more important than another (e.g., it is closer to the viewer), we can adaptively subdivide the more important regions of the surface and leave less important regions less refined. This is called *adaptive refinement*.

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

Bézier curve. Amazingly, the concatenation of these two Bézier curves is equal to the original curve. (We will leave the proof of this as an exercise.)

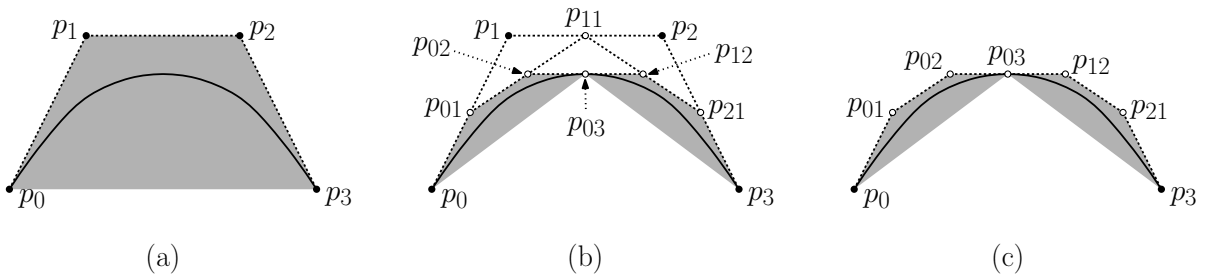


Fig. 80: Bézier subdivision.

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

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

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

This is sometimes called a *bilinear interpolation*, because it is linear in each of the two parameters individually. Note that the shape is not flat, however. It is called a *hyperboloid* (see Fig. 81(a).)

Recalling that $(1-u)$ and u are the first-degree Bézier blending functions $b_{0,1}(u)$ and $b_{1,1}(u)$, we see that this can be written as

$$\begin{aligned}\mathbf{p}(u,v) &= b_{01}(v)b_{01}(u)\mathbf{p}_{00} + b_{01}(v)b_{11}(u)\mathbf{p}_{01} + b_{11}(v)b_{01}(u)\mathbf{p}_{10} + b_{11}(v)b_{11}(u)\mathbf{p}_{11} \\ &= \sum_{i=0}^{1}\sum_{j=0}^{1} b_{i,1}(v)b_{j,1}(u)\mathbf{p}_{i,j}.\end{aligned}$$

Generalizing this to the next higher degree, say quadratic Bézier surfaces, we have we have a 3×3 array of control points, $\mathbf{p}_{ij}$, $0 \le i, j \le 2$, and the resulting parametric formula is

$$\mathbf{p}(u,v) = \sum_{i=0}^{2}\sum_{j=0}^{2} b_{i,2}(v)b_{j,2}(u)\mathbf{p}_{i,j}.$$

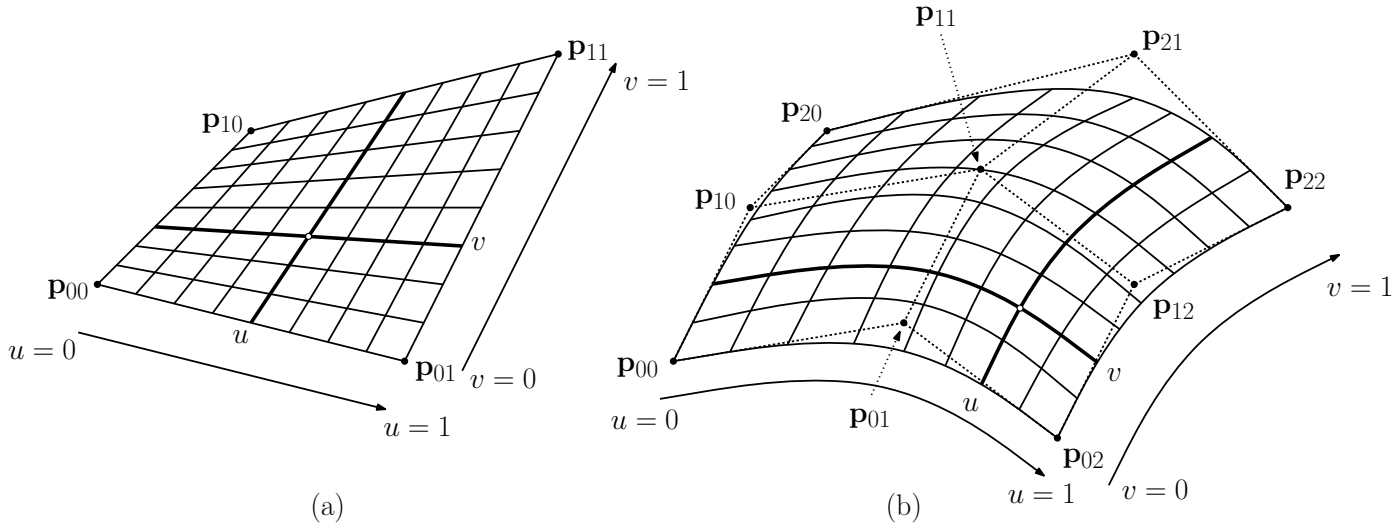


Fig. 81: Bézier surfaces of degree-1 (left) and degree-2 (right).

(See Fig. 81(b).) This is called a *tensor product* construction.

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

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

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

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

B-splines were developed to address these shortcomings. The idea is that we will still use smooth blending functions multiplied times the control points, but these functions will have

the property that these blending functions are nonzero only over a small amount of the parameter range. Thus these functions have only *local support*. Over the nonzero range, they will consist of the concatenation of smooth polynomials. As before each point on the curve will be given by blending the control points

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

where $B_i(u)$ denotes the ith blending function. Fig. 82(a) gives a crude rendering of B-splines blending functions of order-2. Note that once we know how to construct curves, we can apply the same tensor-product construction to form B-spline surfaces.

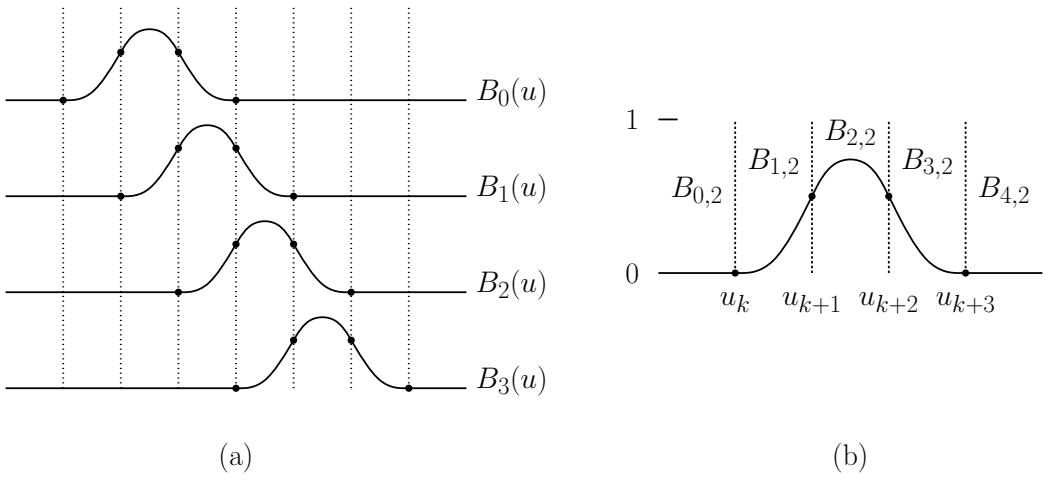


Fig. 82: B-spline basis functions. (Not drawn to scale.)

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

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

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

These parameter values are called *knot points*. (Note that the term "point" does not refer to a point in space, as with control points. These are just scalar values.) Each of the blending functions will consist of the concatenation of polynomial functions, with one polynomial over each *knot interval*, $[u_{i-1}, u_i]$. For simplicity you might think of these as being intervals of unit

length for the time being, but we will see later that there are advantages to making intervals of different sizes. There will be a relationship between the number of intervals n and the number of points m, which we will consider later.

How do we define the B-spline blending functions? There are two ways to do this. The first is to write down the requirements that the blending functions must be C^2 continuous at the joint points, and that they satisfy the convex hull property. Together these constraints completely define B-splines. (We leave this as an exercise.)

Instead, as with the Bézier blending functions, we will do this by recursively applying linear interpolation to the blending functions of the next lower degree. An elegant recursive expression of the blending function (but somewhat difficult to understand) is given by the *Cox-deBoor recursion*. Let $B_{i,d}(u)$ denote the ith blending function for a B-spline of degree d.

$$\begin{aligned} B_{k,0}(u) &= \begin{cases} 1 & \text{if } u_k \le u < u_{k+1}, \\ 0 & \text{otherwise.} \end{cases} \\ B_{k,d}(u) &= \frac{u - u_k}{u_{k+d} - u_k} B_{k,d-1}(u) + \frac{u_{k+d+1} - u}{u_{k+d+1} - u_{k+1}} B_{k+1,d-1}(u). \end{aligned}$$

This is quite hard to comprehend at first sight. However observe that as with Bézier curves, the curve at each degree is expressed as the weighted average of two curves and the next lower degree. It can be proved (by induction) that irrespective of the knot spacing the blending functions sum to 1.

If you grind through the definitions, then you will see that $B_{k,0}(u)$ is a step function that is 1 in the interval $[u_k, u_{k+1})$. $B_{k,1}(u)$ spans two intervals and is a piecewise linear function that goes from 0 to 1 and then back to 0. $B_{k,2}(u)$ spans three intervals and is a piecewise quadratic that grows from 0 to $\frac{1}{4}$, then up to $\frac{3}{4}$ in the middle of the second interval, back to $\frac{1}{4}$, and back to 0. Finally $B_{k,3}(u)$ is a cubic that spans four intervals growing from 0 to $\frac{1}{6}$ to $\frac{2}{3}$, then back to $\frac{1}{6}$ and to 0. Thus, successively higher degrees are successively smoother.

For example, the blending functions for the B-spline of order-2 are (see Fig. 82(b)):

$$B_{k,2} = \begin{cases} 0 & u < u_k \\ \frac{1}{2}(u - u_k)^2 & u_k \le u < u_{k+1} \\ -(u - u_{k+1})^2 + (u - u_{k+1}) + \frac{1}{2} & u_{k+1} \le u < u_{k+2} \\ \frac{1}{2}(1 - (u - u_{k+2}))^2 & u_{k+2} \le u < u_{k+3} \\ 0 & u_{k+3} \le u. \end{cases}$$

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.

Your eyes may strain to understand this formula, but if you work things out piece by piece, you'll see that the equations are actually fairly reasonable. For example, observe that in $B_{k,2}$ there are three intervals in which the function is nonzero, running from u_k to u_{k+3}. In the first interval, we have a quadratic that grows from 0 to $\frac{1}{2}$ (assuming that each interval $[u_i, u_{i+1}]$ is of size 1). In the second interval, we have a inverted parabola that starts at $\frac{1}{2}$ (when $u = u_{k+1}$) grows to $-\frac{1}{4} + \frac{1}{2} + \frac{1}{2} = \frac{3}{4}$ (when u is midway between u_{k+1} and u_{k+2}) and

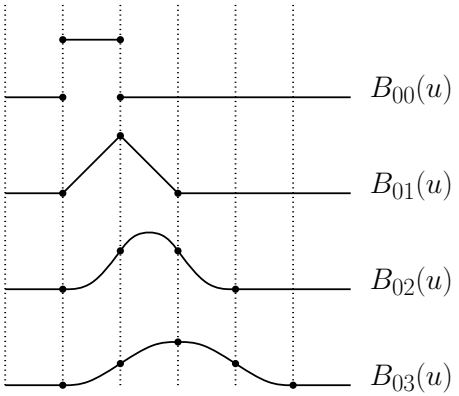


Fig. 83: B-spline blending functions of various orders. (Not drawn to scale.)

then shrinks down to $\frac{1}{2}$ (when $u = u_{k+2}$). In the third interval, we have a quadratic that decreases from $\frac{1}{2}$ down to 0. (See the function labeled $B_{0,2}$ in Fig. 83.)

There is one other important issue that we have not addressed, namely, how to assign knot points to contol points. Due to time limitations, we will skip this issue, but please refer to any standard source on B-splines for further information.

Lecture 22: 3-d Rotation and Quaternions

Rotation and Orientation in 3-Space: One of the trickier problems 3-d geometry is that of parameterizing rotations and the orientation of frames. We have introduced the notion of orientation before (e.g., clockwise or counterclockwise). Here we mean the term in a somewhat different sense, as a directional position in space. Describing and managing rotations in 3-space is a somewhat more difficult task (at least conceptually), compared with the relative simplicity of rotations in the plane.

Why do we care about rotations? Suppose that you are an animation programmer for a computer graphics studio. The object that you are animating is to be moved smoothly from one location to another. If the object is in the same directional orientation before and after, we can just translate from one location to the other. If not, we need to find a way of interpolating between its two orientations. This usually involves rotations in 3-space. But how should these rotations be performed so that the animation looks natural? Another example is one in which the world is stationary, but the camera is moving from one location and viewing situation to another. Again, how can we move smoothly and naturally from one to the other?

Since smoothly interpolating positions by translation is pretty easy to understand, let us ignore the issue of position, and just focus on orientations and rotations about the origin. Let F denote the standard coordinate frame, and consider another orthonormal frame G. We want some way to represent G concisely, relative to F, and generally to interpolate a motion from F to G (see Fig. 84).

Of course, we could just represent F and G by their three orthonormal basis vectors. But if we were to try to interpolate (linearly) between corresponding pairs of basis vectors, the

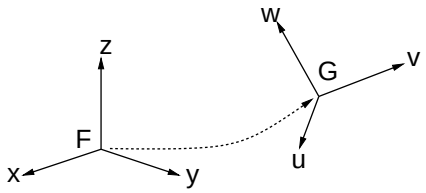


Fig. 84: Moving between frames.

intermediate vectors would not necessarily be orthonormal.

We will explore two methods for dealing with rotation, *Euler angles* and *quaternions*.

Euler Angles: Leonard Euler was a famous mathematician who lived in the 18th century. He proved many important theorems in geometry, algebra, and number theory, and he is credited as the inventor of graph theory. Among his many theorems is one that states that the composition any number of rotations in three-space can be expressed as a single rotation in 3-space about an appropriately chosen vector. Euler also showed that any rotation in 3-space could be broken down into exactly three rotations, one about each of the coordinate axes.

Suppose that you are a pilot, such that the x-axis points to your left, the y-axis points ahead of you, and the z-axis points up (see Fig. 85). Then a rotation about the x-axis, denoted by ϕ, is called the *pitch*. A rotation about the y-axis, denoted by θ, is called *roll*. A rotation about the z-axis, denoted by ψ, is called *yaw*. Euler's theorem states that any position in space can be expressed by composing three such rotations, for an appropriate choice of (ϕ, θ, ψ).

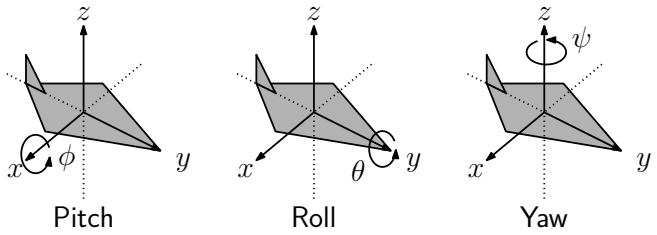


Fig. 85: Euler angles: pitch, roll, and yaw.

Shortcomings of Euler angles: There are some problems with Euler angles. One issue is the fact that this representation depends on the choice of coordinate system. In the plane, a 30 degree rotation is the same, no matter what direction the axes are pointing (as long as they are orthonormal and right-handed). However, the result of an Euler-angle rotation depends very much on the choice of the coordinate frame and on the order in which the axes are named. (Later, we will see that quaternions do provide such an intrinsic system.)

Another problem with Euler angles is called *gimbal lock*. Whenever we rotate about one axis, it is possible that we could bring the other two axes into alignment with each other. (This happens, for example if we rotate x by 90°.) This causes problems because the other two axes no longer rotate independently of each other, and we effectively lose one degree of freedom. Gimbal lock as induced by one ordering of the axes can be avoided by changing the order in

which the rotations are performed. But, this is rather messy, and it would be nice to have a system that is free of this problem.

Angular Displacement: Let us next consider an approach to rotation that is invariant under rigid changes of the coordinate system. This will eventually lead us to the concept of a quaternion.

In contrast to Euler angles, a more intrinsic way to express rotations (about the origin) in 3-space is in terms of two quantities, (θ, u), consisting of an angle θ, and an axis of rotation u. Let's consider how we might do this. First consider a vector v to be rotated. Let us assume that u is of unit length.

Our goal is to describe the rotation of a vector v as a function of θ and u. Let $R(v)$ denote this rotated vector (see Fig. 86(a)). In order to derive this, we begin by decomposing v as the sum of its components that are parallel to and orthogonal to u, respectively.

$$v_\parallel = (u \cdot v)u \qquad v_\perp = v - v_\parallel = v - (u \cdot v)u.$$

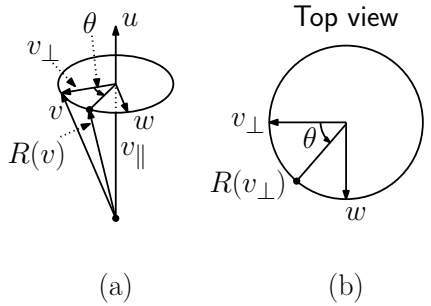


Fig. 86: Angular displacement.

Note that $v_\parallel$ is unaffected by the rotation, but $v_\perp$ is rotated to a new position $R(v_\perp)$. To determine this rotated position, we will first construct a vector that is orthogonal to $v_\perp$ lying in the plane of rotation.

$$w = u \times v_\perp = u \times (v - v_\parallel) = (u \times v) - (u \times v_\parallel) = u \times v.$$

The last step follows from the fact that u and $v_\parallel$ are parallel, and so the cross product is zero. Clearly w is orthogonal to both $v_\perp$ and u. Furthermore, because $v_\perp$ is orthogonal to the unit vector u, it follows from basic properties of the cross product that w is the same length as $v_\perp$.

Now, consider the plane spanned by $v_\perp$ and w (see Fig. 86(b)). We have

$$R(v_\perp) = (\cos\theta)v_\perp + (\sin\theta)w.$$

From this and the fact that $R(v_\parallel) = v_\parallel$, we have

$$\begin{aligned} R(v) &= R(v_\parallel) + R(v_\perp) \\ &= v_\parallel + (\cos\theta)v_\perp + (\sin\theta)w \\ &= (u \cdot v)u + (\cos\theta)(v - (u \cdot v)u) + (\sin\theta)w \\ &= (\cos\theta)v + (1 - \cos\theta)u(u \cdot v) + (\sin\theta)(u \times v). \end{aligned}$$

In summary, we have the following formula expressing the effect of the rotation of vector v by angle θ about a rotation axis u:

$$R(v) \;=\; (\cos\theta)v + (1-\cos\theta)u(u\cdot v) + (\sin\theta)(u\times v). \tag{1}$$

This expression is the image of v under the rotation. Notice that, unlike Euler angles, this is expressed entirely in terms of intrinsic geometric functions (such as dot and cross product), which do not depend on the choice of coordinate frame. This is a major advantage of this approach over Euler angles.

Quaternions: We will now delve into a subject, which at first may seem quite unrelated. But keep the above expression in mind, since it will reappear in most surprising way. This story begins in the early 19th century, when the great mathematician William Rowan Hamilton was searching for a generalization of the complex number system.

Imaginary numbers can be thought of as linear combinations of two basis elements, 1 and i, which satisfy the multiplication rules $1^2 = 1$, $i^2 = -1$ and $1\cdot i = i\cdot 1 = i$. (The interpretation of $i = \sqrt{-1}$ arises from the second rule.) A complex number $a + bi$ can be thought of as a vector in 2-dimensional space (a, b). Two important concepts with complex numbers are the *modulus*, which is defined to be $\sqrt{a^2+b^2}$, and the *conjugate*, which is defined to be $(a, -b)$. In vector terms, the modulus is just the length of the vector and the conjugate is just a vertical reflection about the x-axis. If a complex number is of modulus 1, then it can be expressed as $(\cos\theta, \sin\theta)$. Thus, there is a connection between complex numbers and 2-dimensional rotations. Also, observe that, given such a unit modulus complex number, its conjugate is $(\cos\theta, -\sin\theta) = (\cos(-\theta), \sin(-\theta))$. Thus, taking the conjugate is something like negating the associated angle.

Hamilton was wondering whether this idea could be extended to three dimensional space. You might reason that, to go from 2D to 3D, you need to replace the single imaginary quantity i with two imaginary quantities, say i and j. Unfortunately, this this idea does not work. After years of work, Hamilton came up with the idea of, rather than using two imaginaries, instead using three imaginaries i, j, and k, which behave as follows:

$$i^2 = j^2 = k^2 = ijk = -1 \qquad ij = k,\; jk = i,\; ki = j.$$

Combining these, it follows that $ji = -k$, $kj = -i$ and $ik = -j$. The skew symmetry of multiplication (e.g., $ij = -ji$) was actually a major leap, since multiplication systems up to that time had been commutative.)

Hamilton defined a *quaternion* to be a generalized complex number of the form

$$\mathbf{q} = q_0 + q_1 i + q_2 j + q_3 k.$$

Thus, a quaternion can be viewed as a 4-dimensional vector $\mathbf{q} = (q_0, q_1, q_2, q_3)$. The first quantity is a scalar, and the last three define a 3-dimensional vector, and so it is a bit more intuitive to express this as $\mathbf{q} = (s, u)$, where $s = q_0$ is a scalar and $u = (q_1, q_2, q_3)$ is a vector in 3-space. We can define the same concepts as we did with complex numbers:

Conjugate: $\mathbf{q}^* = (s, -u)$.

Modulus: $|\mathbf{q}| = \sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2} = \sqrt{s^2 + (u \cdot u)}$.

Unit Quaternion: $\mathbf{q}$ is said to be a unit quaternion if $|\mathbf{q}| = 1$.

Quaternion multiplication: Consider two quaternions $\mathbf{q} = (s, u)$ and $\mathbf{p} = (t, v)$:

$$\begin{aligned} \mathbf{q} &= (s, u) = s + u_x i + u_y j + u_z k \\ \mathbf{p} &= (t, v) = t + v_x i + v_y j + v_z k. \end{aligned}$$

If we multiply these two together, we'll get lots of cross-product terms, such as $(u_x i)(v_y j)$, but we can simplify these by using Hamilton's rules. That is, $(u_x i)(v_y j) = u_x v_h (ij) = u_x v_h k$. If we do this, simplify, and collect common terms, we get a very messy formula involving 16 different terms (see the appendix at the end of this lecture). The formula can be expressed somewhat succinctly in the following form:

$$\mathbf{qp} = (st - (u \cdot v), sv + tu + u \times v).$$

Note that the above expression is in the quaternion scalar-vector form. The first term $st - (u \cdot v)$ evaluates to a scalar (recalling that the dot product returns a scalar), and the second term $(sv + tu + u \times v)$ is a sum of three vectors, and so is a vector. It can be shown that quaternion multiplication is associative, but not commutative.

Quaternion multiplication and 3-dimensional rotation: Before considering rotations, we first define a *pure quaternion* to be one with a 0 scalar component

$$\mathbf{p} = (0, v).$$

Any quaternion of nonzero magnitude has a multiplicative *inverse*, which is defined to be

$$\mathbf{q}^{-1} = \frac{1}{|\mathbf{q}|^2}\mathbf{q}^*.$$

(To see why this works, try multiplying $\mathbf{q}\mathbf{q}^{-1}$, and see what you get.) Observe that if $\mathbf{q}$ is a unit quaternion, then it follows that $\mathbf{q}^{-1} = \mathbf{q}^*$.

As you might have guessed, our objective will be to show that there is a relation between rotating vectors and multiplying quaternions. In order apply this insight, we need to first show how to represent rotations as quaternions and 3-dimensional vectors as quaternions. After a bit of experimentation, the following does the trick:

Vector: Given a vector $v = (v_x, v_y, v_z)$ to be rotated, we will represent it by the pure quaternion $(0, v)$.

Rotation: To represent a rotation by angle θ about a unit vector u, you might think, we'll use the scalar part to represent θ and the vector part to represent u. Unfortunately, this doesn't quite work. After a bit of experimentation, you will discover that the right way to encode this rotation is with the quaternion $\mathbf{q} = (\cos(\theta/2), (\sin(\theta/2))u)$. (You might wonder, why we do we use $\theta/2$, rather than θ. The reason, as we shall see below, is that "this is what works.")

Rotation Operator: Given a vector v represented by the quaternion $\mathbf{p} = (0, v)$ and a rotation represented by a unit quaternion $\mathbf{q}$, we define the *rotation operator* to be:

$$R_{\mathbf{q}}(\mathbf{p}) \;=\; \mathbf{q}\mathbf{p}\mathbf{q}^{-1} \;=\; \mathbf{q}\mathbf{p}\mathbf{q}^{*}.$$

(The last equality results from the fact that $\mathbf{q}^{-1} = \mathbf{q}^{*}$, if $\mathbf{q}$ is a unit quaternion). We claim that the result of this operation will always be a unit quaternion, and so it is possible to interpret the result as a vector. In particular, this vector will be the result of applying the rotation $\mathbf{q}$ to v.

Why does this work? Let's begin by applying the multiplication rule and use the fact that $\mathbf{q}^{-1} = \mathbf{q}^{*}$ for a unit quaternion $\mathbf{q} = (s, u)$. Given $\mathbf{p} = (0, v)$, by expanding the rotation operator definition and simplifying we obtain:

$$R_{\mathbf{q}}(\mathbf{p}) \;=\; (0,\; (s^2 - (u \cdot u))v + 2u(u \cdot v) + 2s(u \times v)). \tag{2}$$

(We leave the derivation as an exercise, but a few nontrivial facts regarding dot products and cross products need to applied.)

Let us see if we can express this in a more suggestive form. Since $\mathbf{q}$ is of unit magnitude, we can express it as

$$\mathbf{q} = \left(\cos\frac{\theta}{2}, \left(\sin\frac{\theta}{2}\right) u\right), \qquad \text{where } \|u\| = 1.$$

Plugging this into the above expression and applying some standard trigonometric identities, we obtain

$$\begin{aligned} R_{\mathbf{q}}(\mathbf{p}) &= \left(0, \left(\cos^2\frac{\theta}{2} - \sin^2\frac{\theta}{2}\right) v + 2\left(\sin^2\frac{\theta}{2}\right) u(u \cdot v) + 2\cos\frac{\theta}{2}\sin\frac{\theta}{2}(u \times v)\right) \\ &= (0,\; (\cos\theta)v + (1 - \cos\theta)u(u \cdot v) + \sin\theta(u \times v)). \end{aligned}$$

Now, recall the rotation displacement equation presented earlier in the lecture. The vector part of this quaternion is *identical*, implying that the quaternion rotation operator achieves the desired rotation.

Example: Consider the 3-d rotation shown in Fig. 87. This rotation can be achieved by performing a rotation about the y-axis by $\theta = -90$ degrees. Thus $\theta = -\pi/2$, and $u = (0, 1, 0)$. Thus the quaternion that encodes this rotation is

$$\mathbf{q} \;=\; (\cos(\theta/2), (\sin(\theta/2))u) \;=\; \left(\cos\left(-\frac{\pi}{4}\right), \sin\left(-\frac{\pi}{4}\right)(0,1,0)\right) \;=\; \left(\frac{1}{\sqrt{2}}, \left(0, -\frac{1}{\sqrt{2}}, 0\right)\right).$$

Let us consider how the x-unit vector $v = (1, 0, 0)^{\mathsf{T}}$ is transformed under this rotation. To reduce this to a quaternion operation, we encode v as a pure quaternion $\mathbf{p} = (0, v) = (0, (1, 0, 0))$. We then apply the rotation operator, and so by Eq. (2) we have

$$\begin{aligned} R_{\mathbf{q}}(\mathbf{p}) &= (0,\; (1/2 - 1/2)(1,0,0) + 2(0,1,0)0 + (2/\sqrt{2})((0, -1/\sqrt{2}, 0) \times (1,0,0))) \\ &= (0, (0,0,0) + (0,0,0) + (-1)(0,0,-1)) \\ &= (0, (0,0,1)). \end{aligned}$$

Thus p is mapped to a point on the z-axis, as expected.

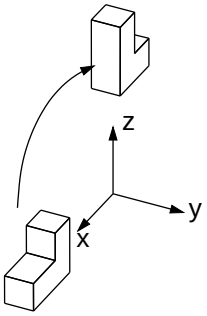


Fig. 87: Rotation example.

Composing Rotations: We have shown that each unit quaternion corresponds to a rotation in 3-space. This is an elegant representation, but can we manipulate rotations through quaternion operations? The answer is yes. In particular, the action of multiplying two unit quaternions results in another unit quaternion. Furthermore, the resulting product quaternion corresponds to the composition of the two rotations. In particular, given two unit quaternions $\mathbf{q}$ and $\mathbf{q}'$, a rotation by $\mathbf{q}$ followed by a rotation by $\mathbf{q}'$ is equivalent to a single rotation by the product $\mathbf{q}'' = \mathbf{q}'\mathbf{q}$. That is,

$$R_{\mathbf{q}'} R_{\mathbf{q}} = R_{\mathbf{q}''} \qquad \text{where } \mathbf{q}'' = \mathbf{q}'\mathbf{q}.$$

This follows from the associativity of quaternion multiplication, and the fact that $(\mathbf{q}\mathbf{q}')^{-1} = \mathbf{q}^{-1}\mathbf{q}'^{-1}$, as shown below.

$$\begin{aligned} R_{\mathbf{q}'}(R_{\mathbf{q}}(\mathbf{p})) &= \mathbf{q}'(\mathbf{q}\mathbf{p}\mathbf{q}^{-1})\mathbf{q}'^{-1} = (\mathbf{q}'\mathbf{q})\mathbf{p}(\mathbf{q}^{-1}\mathbf{q}'^{-1}) \\ &= (\mathbf{q}'\mathbf{q})\mathbf{p}(\mathbf{q}\mathbf{q}')^{-1} = \mathbf{q}''\mathbf{p}\mathbf{q}''^{-1} \\ &= R_{\mathbf{q}''}(\mathbf{p}). \end{aligned}$$

Quaternion Summary: In summary, quaternions are a generalization of the concept of complex numbers, which can be used to represent rotations in three dimensional space. Unlike Euler angles, quaternions are independent of the coordinate system. Also, they do not suffer from the problem of gimbal lock. Thus, from a mathematical perspective, they represent a much cleaner system for representing rotations.

- Quaternions can be used to represent the rotation (orientation) of an object in 3-dimensional space.
- A rotation by a given angle θ about a unit vector u can be represented by the unit quaternion $\mathbf{q} = (\cos(\theta/2), (\sin(\theta/2))u)$.
- A vector v is represented by the pure quaternion $\mathbf{p} = (0, v)$.
- The effect of applying this rotation to v is given by $R_{\mathbf{q}}(\mathbf{p}) = \mathbf{q}\mathbf{p}\mathbf{q}^*$. This is a pure quaternion, and so can be mapped back to a vector by discarding the scalar part.
- Given two rotation quaternions $\mathbf{q}$ and $\mathbf{q}'$, the product $\mathbf{q}'\mathbf{q}$ corresponds to applying $\mathbf{q}$ followed by $\mathbf{q}'$.

Supplemental Topics

Supplemental Lecture 1: Scan Conversion

Scan Conversion: We turn now to a number of miscellaneous issues involved in the implementation of computer graphics systems. In our top-down approach we have concentrated so far on the high-level view of computer graphics. In the next few lectures we will consider how these things are implemented. In particular, we consider the question of how to map 2-dimensional geometric objects (as might result from projection) to a set of pixels to be colored. This process is called *scan conversion* or *rasterization*. We begin by discussing the simplest of all rasterization problems, drawing a single line segment.

Let us think of our raster display as an integer grid, in which each pixel is a circle of radius 1/2 centered at each point of the grid. We wish to illuminate a set of pixels that lie on or close to the line. In particular, we wish to draw a line segment from $q = (q_x, q_y)$ to $r = (r_x, r_y)$, where the coordinates are integer grid points (typically by a process of rounding). Let us assume further that the slope of the line is between 0 and 1, and that $q_x < r_x$. This may seem very restrictive, but it is not difficult to map any line drawing problem to satisfy these conditions. For example, if the absolute value of the slope is greater than 1, then we interchange the roles of x and y, thus resulting in a line with a reciprocal slope. If the slope is negative, the algorithm is very easy to modify (by decrementing rather than incrementing). Finally, by swapping the endpoints we can always draw from left to right.

Bresenham's Algorithm: We will discuss an algorithm, which is called *Bresenham's algorithm*. It is one of the oldest algorithms known in the field of computer graphics. It is also an excellent example of how one can squeeze every but of efficiency out an algorithm. We begin by considering an *implicit* representation of the line equation. (This is used only for deriving the algorithm, and is not computed explicitly by the algorithm.)

$$f(x, y) = ax + by + c = 0.$$

If we let $d_x = r_x - q_x$, $d_y = r_y - q_y$, it is easy to see (by substitution) that $a = d_y$, $b = -d_x$, and $c = -(q_x r_y - r_x q_y)$. Observe that all of these coefficients are all integers. Also observe that $f(x, y) > 0$ for points that lie below the line and $f(x, y) < 0$ for points above the line. For reasons that will become apparent later, let us use an equivalent representation by multiplying by 2

$$f(x, y) = 2ax + 2by + 2c = 0.$$

Here is the intuition behind Bresenham's algorithm. For each integer x value, we wish to determine which integer y value is closest to the line. Suppose that we have just finished drawing a pixel (p_x, p_y) and we are interested in figuring out which pixel to draw next. Since the slope is between 0 and 1, it follows that the next pixel to be drawn will either be the pixel to our East ($E = (p_x + 1, p_y)$) or the pixel to our NorthEast ($NE = (p_x + 1, p_y + 1)$).

Let q denote the exact y-value (a real number) of the line at $x = p_x + 1$. Let $m = p_y + 1/2$ denote the y-value midway between E and NE. If $q < m$ then we want to select E next, and otherwise we want to select NE. IF $q = m$ then we can pick either, say E. See the figure.

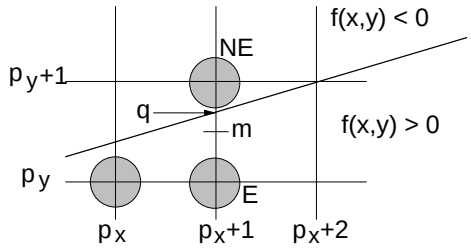


Fig. 88: Bresenham's midpoint algorithm.

To determine which one to pick, we have a *decision variable* D which will be the value of f at the midpoint. Thus

$$\begin{aligned} D &= f(p_x + 1, p_y + (1/2)) \\ &= 2a(p_x + 1) + 2b\left(p_y + \frac{1}{2}\right) + 2c \\ &= 2ap_x + 2bp_y + (2a + b + 2c). \end{aligned}$$

If $D > 0$ then m is below the line, and so the NE pixel is closer to the line. On the other hand, if $D \leq 0$ then m is above the line, so the E pixel is closer to the line. (Note: We can see now why we doubled $f(x, y)$. This makes D an integer quantity.)

The good news is that D is an integer quantity. The bad news is that it takes at least at least two multiplications and two additions to compute D (even assuming that we precompute the part of the expression that does not change). One of the clever tricks behind Bresenham's algorithm is to compute D *incrementally*. Suppose we know the current D value, and we want to determine its next value. The next D value depends on the action we take at this stage.

We go to E next: Then the next midpoint will have coordinates $(p_x + 2, p_y + (1/2))$ and hence the new D value will be

$$\begin{aligned} D_{new} &= f(p_x + 2, p_y + (1/2)) \\ &= 2a(p_x + 2) + 2b\left(p_y + \frac{1}{2}\right) + 2c \\ &= 2ap_x + 2bp_y + (4a + b + 2c) \\ &= 2ap_x + 2bp_y + (2a + b + 2c) + 2a \\ &= D + 2a = D + 2d_y. \end{aligned}$$

Thus, the new value of D will just be the current value plus $2d_y$.

We go to *NE* next: Then the next midpoint will have coordinates $(p_x + 2, p_y + 1 + (1/2))$ and hence the new D value will be

$$\begin{aligned} D_{new} &= f(p_x + 2, p_y + 1 + (1/2)) \\ &= 2a(p_x + 2) + 2b\left(p_y + \frac{3}{2}\right) + 2c \\ &= 2ap_x + 2bp_y + (4a + 3b + 2c) \\ &= 2ap_x + 2bp_y + (2a + b + 2c) + (2a + 2b) \\ &= D + 2(a + b) = D + 2(d_y - d_x). \end{aligned}$$

Thus the new value of D will just be the current value plus $2(d_y - d_x)$.

Note that in either case we need perform only one addition (assuming we precompute the values $2d_y$ and $2(d_y - d_x)$. So the inner loop of the algorithm is quite efficient.

The only thing that remains is to compute the initial value of D. Since we start at (q_x, q_y) the initial midpoint is at $(q_x + 1, q_y + 1/2)$ so the initial value of D is

$$\begin{aligned} D_{init} &= f(q_x + 1, q_y + 1/2) \\ &= 2a(q_x + 1) + 2b\left(q_y + \frac{1}{2}\right) + 2c \\ &= (2aq_x + 2bq_y + 2c) + (2a + b) \\ &= 0 + 2a + b \qquad \text{Since } (q_x, q_y) \text{ is on line} \\ &= 2d_y - d_x. \end{aligned}$$

We can now give the complete algorithm. Recall our assumptions that $q_x < r_x$ and the slope lies between 0 and 1. Notice that the quantities $2d_y$ and $2(d_y - d_x)$ appearing in the loop can be precomputed, so each step involves only a comparison and a couple of additions of integer quantities.

Bresenham's midpoint algorithm

```
void bresenham(IntPoint q, IntPoint r) {
    int dx, dy, D, x, y;
    dx = r.x - q.x;                 // line width and height
    dy = r.y - q.y;
    D = 2*dy - dx;                  // initial decision value
    y = q.y;                        // start at (q.x,q.y)
    for (x = q.x; x <= r.x; x++) {
        writePixel(x, y);
        if (D <= 0) D += 2*dy;      // below midpoint - go to E
        else {                      // above midpoint - go to NE
            D += 2*(dy - dx); y++;
        }
    }
}
```

Bresenham's algorithm can be modified for drawing other sorts of curves. For example, there is a Bresenham-like algorithm for drawing circular arcs. The generalization of Bresenham's algorithm is called the *midpoint algorithm*, because of its use of the midpoint between two pixels as the basic discriminator.

Filling Regions: In most instances we do not want to draw just a single curve, and instead want to fill a region. There are two common methods of defining the region to be filled. One is polygon-based, in which the vertices of a polygon are given. We will discuss this later. The other is *pixel based*. In this case, a boundary region is defined by a set of pixels, and the task is to fill everything inside the region. We will discuss this latter type of filling for now, because it brings up some interesting issues.

The intuitive idea that we have is that we would like to think of a set of pixels as defining the *boundary* of some region, just as a closed curve does in the plane. Such a set of pixels should be connected, and like a curve, they should split the infinite grid into two parts, an *interior* and an *exterior*. Define the *4-neighbors* of any pixel to be the pixels immediately to the north, south, east, and west of this pixel. Define the *8-neighbors* to be the union of the 4-neighbors and the 4 closest diagonal pixels. There are two natural ways to define the notion of being connected, depending on which notion of neighbors is used.

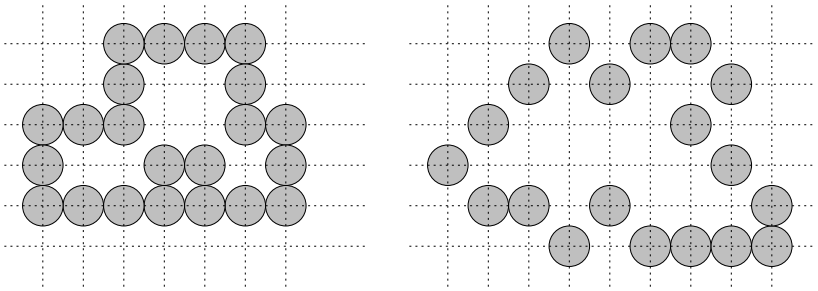


Fig. 89: 4-connected (left) and 8-connected (right) sets of pixels.

4-connected: A set is 4-connected if for any two pixels in the set, there is path from one to the other, lying entirely in the set and moving from one pixel to one of its 4-neighbors.

8-connected: A set is 8-connected if for any two pixels in the set, there is path from one to the other, lying entirely in the set and moving from one pixel to one of its 8-neighbors.

Observe that a 4-connected set is 8-connected, but not vice versa. Recall from the Jordan curve theorem that a closed curve in the plane subdivides the plane into two connected regions, and interior and an exterior. We have not defined what we mean by a closed curve in this context, but even without this there are some problems. Observe that if a boundary curve is 8-connected, then it is generally not true that it separates the infinite grid into two 8-connected regions, since (as can be seen in the figure) both interior and exterior can be joined to each other by a 8-connected path. There is an interesting way to fix this problem. In particular, if we require that the boundary curve be 8-connected, then we require that the region it define be 4-connected. Similarly, if we require that the boundary be 4-connected, it is common to assume that the region it defines be 8-connected.

Recursive Flood Filling: Irrespective of how we define connectivity, the algorithmic question we want to consider is how to fill a region. Suppose that we are given a starting pixel $p = (p_x, p_y)$. We wish to visit all pixels in the same *connected component* (using say, 4-connectivity), and assign them all the same color. We will assume that all of these pixels initially share some common *background color*, and we will give them a new *region color*. The idea is to walk around, as whenever we see a 4-neighbor with the background color we assign it color the region color. The problem is that we may go down dead-ends and may need to backtrack. To handle the backtracking we can keep a stack of unfinished pixels. One way to implement this stack is to use recursion. The method is called *flood filling*. The resulting procedure is simple to write down, but it is not necessarily the most efficient way to solve the problem. See the book for further consideration of this problem.

Recursive Flood-Fill Algorithm (4-connected)

```
void floodFill(intPoint p) {
    if (getPixel(p.x, p.y) == backgroundColor) {
        setPixel(p.x, p.y, regionColor);
        floodFill(p.x - 1, p.y);            // apply to 4-neighbors
        floodFill(p.x + 1, p.y);
        floodFill(p.x, p.y - 1);
        floodFill(p.x, p.y + 1);
    }
}
```

Supplemental Lecture 2: Scan Conversion of Circles

Midpoint Circle Algorithm: Let us consider how to generalize Bresenham's midpoint line drawing algorithm for the rasterization of a circle. We will make a number of assumptions to simplify the presentation of the algorithm. First, let us assume that the circle is centered at the origin. (If not, then the initial conditions to the following algorithm are changed slightly.) Let R denote the (integer) radius of the circle.

The first observations about circles is that it suffices to consider how to draw the arc in the positive quadrant from $\pi/4$ to $\pi/2$, since all the other points on the circle can be determined from these by *8-way symmetry*.

What are the comparable elements of Bresenham's midpoint algorithm for circles? As before, we need an implicit representation of the function. For this we use

$$F(x, y) = x^2 + y^2 - R^2 = 0.$$

Note that for points *inside* the circle (or under the arc) this expression is negative, and for points *outside* the circle (or above the arc) it is positive.

Let's assume that we have just finished drawing pixel (x_p, y_p), and we want to select the next pixel to draw (drawing clockwise around the boundary). Since the slope of the circular arc is between 0 and -1, our choice at each step our choice is between the neighbor to the east

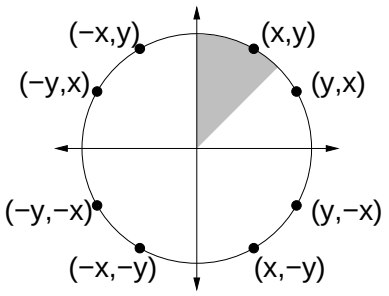


Fig. 90: 8-way symmetry for circles.

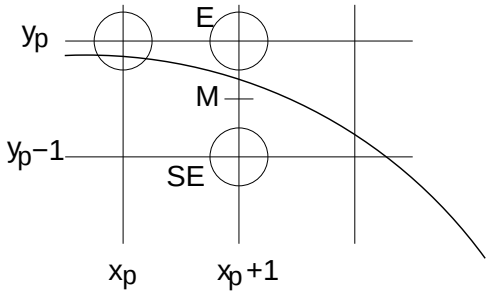


Fig. 91: Midpoint algorithm for circles.

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

$$\begin{aligned} D_{init} &= F(1, R - \frac{1}{2}) \\ &= 1 + (R - \frac{1}{2})^2 - R^2 \\ &= 1 + R^2 - R + \frac{1}{4} - R^2 \\ &= \frac{5}{4} - R. \end{aligned}$$

This is something of a pain, because we have been trying to avoid floating point arithmetic. However, there is a very clever observation that can be made at this point. We are only

interested in testing whether D is positive or negative. Whenever we change the value of D, we do so by a integer increment. Thus, D is always of the form $D' + 1/4$, where D' is an integer. Such a quantity is positive if and only if D' is positive. Therefore, we can just ignore this extra $1/4$ term. So, we initialize $D_{init} = 1 - R$ (subtracting off exactly $1/4$), and the algorithm behaves *exactly* as it would otherwise!

Supplemental Lecture 3: Cohen-Sutherland Line Clipping

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.

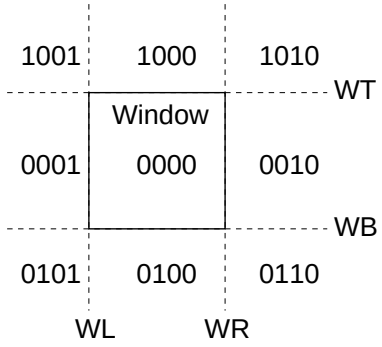


Fig. 92: Cohen-Sutherland region codes.

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

$$\frac{y_c - y_0}{y_1 - y_0} = \frac{WL - x_0}{x_1 - x_0}.$$

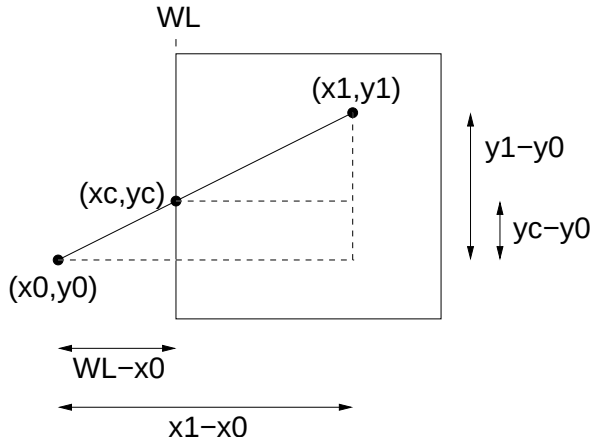


Fig. 93: Clipping on left side of window.

From this we can solve for y_c giving

$$y_c = \frac{WL - x_0}{x_1 - x_0}(y_1 - y_0) + y_0.$$

Thus, we replace (x_0, y_0) with (x_c, y_c), recompute the code values, and continue. This is repeated until the line is trivially accepted (all code bits = 0) or until the line is completely rejected. We can do the same for each of the other cases.

Supplemental Lecture 4: Hidden Surface Removal

Hidden-Surface Removal: We consider algorithmic approaches to an important problem in computer graphics, *hidden surface removal*. We are given a collection of objects in 3-space, represented, say, by a set of polygons, and a viewing situation, and we want to render only the

visible surfaces. Each polygon face is assumed to be flat and opaque. (Extensions to hidden-surface elimination of curved surfaces is an interesting 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.

Object precision: The algorithm computes its results to machine precision (the precision used to represent object coordinates). The resulting image may be enlarged many times without significant loss of accuracy. The output is a set of visible object faces, and the portions of faces that are only partially visible.

Image precision: The algorithm computes its results to the precision of a pixel of the image. Thus, once the image is generated, any attempt to enlarge some portion of the image will result in reduced resolution.

Although image precision approaches have the obvious drawback that they cannot be enlarged without loss of resolution, the fastest and simplest algorithms usually operate by this approach.

The hidden-surface elimination problem for object precision is interesting from the perspective of algorithm design, because it is an example of a problem that is rather hard to solve in the worst-case, and yet there exists a number of fast algorithms that work well in practice. As an example of this, consider a patch-work of n thin horizontal strips in front of n thin vertical strips. (See Fig. 94.) If we wanted to output the set of visible polygons, observe that the complexity of the projected image with hidden-surfaces removed is $O(n^2)$. Hence, it is impossible to beat $O(n^2)$ in the worst case. However, almost no one in graphics uses worst-case complexity as a measure of how good an algorithm is, because these worst-case scenarios do not happen often in practice. (By the way there is an "optimal" $O(n^2)$ algorithm, which is never used in practice.)

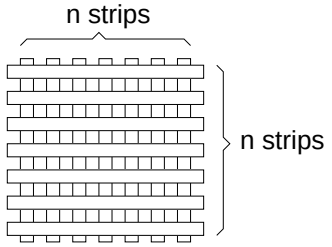


Fig. 94: Worst-case example for hidden-surface elimination.

Culling: Before performing a general hidden surface removal algorithm, it is common to first apply heuristics to remove objects that are obviously not visible. This process is called *culling*. There are three common forms of culling.

Back-face Culling: This is a simple trick, which can eliminate roughly half of the faces from consideration. Assuming that the viewer is never inside any of the objects of the

scene, then the back sides of objects are never visible to the viewer, and hence they can be eliminated from consideration.

For each polygonal face, we assume an outward pointing normal has been computed. 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. (See Fig. 95.)

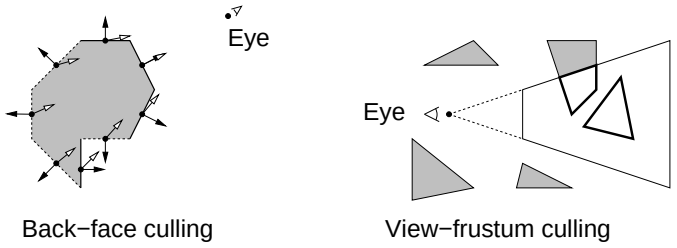


Fig. 95: Culling.

View Frustum Culling: If a polygon does not lie within the view frustum (recall from the lecture on perspective), that is, the region that is visible to the viewer, then it does not need to be rendered. This automatically eliminates polygons that lie behind the viewer. (See Fig. 95.)

This amounts to clipping a 2-dimensional polygon against a 3-dimensional frustum. The Liang-Barsky clipping algorithm can be generalized to do this.

Visibility Culling: Sometimes a polygon can be culled because it is "known" that the polygon cannot be visible, based on knowledge of the domain. For example, if you are rendering a room of a building, then it is reasonable to infer that furniture on other floors or in distant rooms on the same floor are not visible. This is the hardest type of culling, because it relies on knowledge of the environment. This information is typically precomputed, based on expert knowledge or complex analysis of the environment.

Depth-Sort Algorithm: A fairly simple hidden-surface algorithm is based on the principle of painting objects from back to front, so that more distant polygons are overwritten by closer polygons. This is called the *depth-sort algorithm*. This suggests the following algorithm: sort all the polygons according to increasing distance from the viewpoint, and then scan convert them in reverse order (back to front). This is sometimes called the *painter's algorithm* because it mimics the way that oil painters usually work (painting the background before the foreground). The painting process involves setting pixels, so the algorithm is an image precision algorithm.

There is a very quick-and-dirty technique for sorting polygons, which unfortunately does not generally work. Compute a *representative point* on each polygon (e.g. the centroid or the farthest point to the viewer). Sort the objects by decreasing order of distance from the viewer to the representative point (or using the pseudodepth which we discussed in discussing perspective) and draw the polygons in this order. Unfortunately, just because the representative points are ordered, it does not imply that the entire polygons are ordered. Worse yet, it may be *impossible* to order polygons so that this type of algorithm will work. The Fig. 96 shows such an example, in which the polygons overlap one another cyclically.

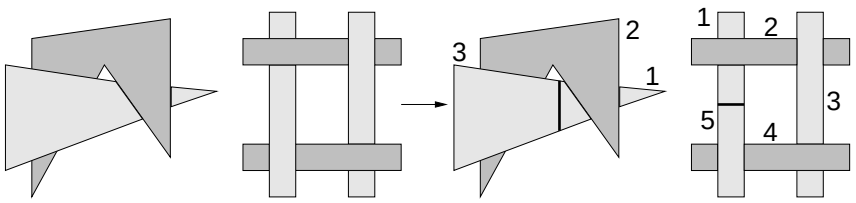


Fig. 96: Hard cases to depth-sort.

In these cases we may need to *cut* one or more of the polygons into smaller polygons so that the depth order can be uniquely assigned. Also observe that if two polygons do not overlap in x, y space, then it does not matter what order they are drawn in.

Here is a snapshot of one step of the depth-sort algorithm. Given any object, define its z-*extents* to be an interval along the z-axis defined by the object's minimum and maximum z-coordinates. We begin by sorting the polygons by depth using farthest point as the representative point, as described above. Let's consider the polygon P that is currently at the end of the list. Consider all polygons Q whose z-extents overlaps P's. This can be done by walking towards the head of the list until finding the first polygon whose maximum z-coordinate is less than P's minimum z-coordinate. Before drawing P we apply the following tests to each of these polygons Q. If any answers is "yes", then we can safely draw P before Q.

(1) Are the x-extents of P and Q disjoint?

(2) Are the y-extents of P and Q disjoint?

(3) Consider the plane containing Q. Does P lie entirely on the opposite side of this plane from the viewer?

(4) Consider the plane containing P. Does Q lie entirely on the same side of this plane from the viewer?

(5) Are the projections of the polygons onto the view window disjoint?

In the cases of (1) and (2), the order of drawing is arbitrary. In cases (3) and (4) observe that if there is any plane with the property that P lies to one side and Q and the viewer lie to the other side, then P may be drawn before Q. The plane containing P and the plane containing Q are just two convenient planes to test. Observe that tests (1) and (2) are very fast, (3) and (4) are pretty fast, and that (5) can be pretty slow, especially if the polygons are nonconvex.

If all tests fail, then the only way to resolve the situation may be to split one or both of the polygons. Before doing this, we first see whether this can be avoided by putting Q at the end of the list, and then applying the process on Q. To avoid going into infinite loops, we mark each polygon once it is moved to the back of the list. Once marked, a polygon is never moved to the back again. If a marked polygon fails all the tests, then we need to split. To do this, we use P's plane like a knife to split Q. We then take the resulting pieces of Q, compute the farthest point for each and put them back into the depth sorted list.

In theory this partitioning could generate $O(n^2)$ individual polygons, but in practice the number of polygons is much smaller. The depth-sort algorithm needs no storage other than the frame buffer and a linked list for storing the polygons (and their fragments). However, it

suffers from the deficiency that each pixel is written as many times as there are overlapping polygons.

Depth-buffer Algorithm: The *depth-buffer algorithm* is one of the simplest and fastest hidden-surface algorithms. Its main drawbacks are that it requires a lot of memory, and that it only produces a result that is accurate to pixel resolution and the resolution of the depth buffer. Thus the result cannot be scaled easily and edges appear jagged (unless some effort is made to remove these effects called "aliasing"). It is also called the *z-buffer algorithm* because the z-coordinate is used to represent depth. 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. The depth-buffer values are initially set to the maximum possible 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 (recall Lecture 12), 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.

Implementation of the Depth-Buffer Algorithm: Consider the scan-conversion of a triangle shown in Fig. 97 using a depth-buffer. Our task is to convert the triangle into a collection of pixels, and assign each pixel a depth value. Because this step is executed so often, it is necessary that it be performed efficiently. (In fact, graphics cards implement some variation of this approach in hardware.)

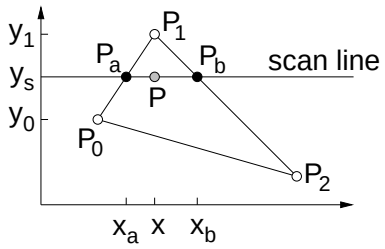


Fig. 97: Depth-buffer scan conversion.

Let P_0, P_1, and P_2 be the vertices of the triangle after the perspective-plus-depth transformation has been applied, and the points have been scaled to the screen size. Let $P_i = (x_i, y_i, z_i)$ be the coordinates of each vertex, where (x_i, y_i) are the final screen coordinates and z_i is the depth of this point.

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 $\overline{P_0P_1}$. The depth of point P_a, can be interpolated by the following affine combination

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

(Is this really an accurate interpolation of the depth information? Remember that the projection transformation maps lines to lines, but depth is mapped nonlinearly. It turns out that this does work, but we'll leave the explanation as an exercise.) We can derive a similar expression for z_b.

Then as we scan along the scan line, for each value of y we have

$$\alpha = \frac{x - x_a}{x_b - x_a},$$

and the depth of the scanned point is just the affine combination

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

It is more efficient (from the perspective of the number of arithmetic operations) to do this by computing z_a accurately, and then adding a small incremental value as we move to each successive pixel on the line. The scan line traverses $x_b - x_a$ pixels, and over this range, the depth values change over the range $z_b - z_a$. Thus, the change in depth per pixel is

$$\Delta_z = \frac{z_b - z_a}{x_b - x_a}.$$

Starting with z_a, we add the value Δ_z to the depth value of each successive pixel as we scan across the row. An analogous trick may be used to interpolate the depth values along the left and right edges.

Supplemental Lecture 5: Light and Color

correction.

Achromatic Light: Light and its perception are important to understand for anyone interested in computer graphics. Before considering color, we begin by considering some issues in the perception of light intensity and the generation of light on most graphics devices. Let us consider color-free, or *achromatic light*, that is gray-scale light. It is characterized by one attribute: *intensity* which is a measure of energy, or *luminance*, which is the intensity that we perceive. Intensity affects brightness, and hence low intensities tend to black and high intensities tend to white. Let us assume for now that each intensity value is specified as a number from 0 to 1, where 0 is black and 1 is white. (Actually intensity has no limits, since it is a measure of energy. However, from a practical perspective, every display device has some maximum intensity that it can display. So think of 1 as the brightest white that your monitor can generate.)

Perceived Brightness: You would think that intensity and luminance are linearly proportional to each other, that is, twice the intensity is perceived as being twice as bright. However, the human perception of luminance is nonlinear. For example, suppose we want to generate 10 different intensities, producing a uniform continuous variation from black to white on a typical CRT display. It would seem logical to use equally spaced intensities: $0.0, 0.1, 0.2, 0.3, \ldots, 1.0$. 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. Thus, 0.2 appears to be twice as bright as 0.1, but 0.6 only appears to be 20% brighter than 0.5. In other words, the response R of the human visual system to a light of a given intensity I can be approximated (up to constant factors) by a logarithmic function

$$R(I) = \log I.$$

This is called the *Weber-Fechner law*. (It is not so much a physical law as it is a model of the human visual system.)

For example, suppose that we want to generate intensities that appear to varying linearly between two intensities I_0 to I_1, as α varies from 0 to 1. Rather than computing an affine (i.e., arithmetic) combination of I_0 and I_1, instead we should compute a *geometric combination* of these two intensities

$$I_\alpha = I_0^{1-\alpha} \cdot I_1^{\alpha}.$$

Observe that, as with affine combinations, this varies continuously from I_0 to I_1 as α varies from 0 to 1. The reason for this choice is that the response function varies linearly, that is,

$$R(I_\alpha) \;=\; \log(I_0^{1-\alpha} \cdot I_1^{\alpha}) \;=\; (1-\alpha)\log I_0 + \alpha \log I_1 \;=\; (1-\alpha)R(I_0) + \alpha R(I_1).$$

Gamma Correction: Just to make things more complicated, there is not a linear relation between the voltage supplied to the electron gun of a CRT and the intensity of the resulting phosphor. Thus, the RGB value $(0.2, 0.2, 0.2)$ does not emit twice as much illumination energy as the RGB value $(0.1, 0.1, 0.1)$, when displayed on a typical monitor.

The relationship between voltage and brightness of the phosphors is more closely approximated by the following function:

$$I = V^\gamma,$$

where I denotes the intensity of the pixel and V denotes the voltage on the signal (which is proportional to the RGB values you store in your frame buffer), and γ is a constant that depends on physical properties of the display device. For typical CRT monitors, it ranges from 1.5 to 2.5. (2.5 is typical for PC's and Sun workstations.) The term *gamma* refers to the nonlinearity of the transfer function.

Users of graphics systems need to correct this in order to get the colors they expect. *Gamma correction* is the process of altering the pixel values in order to compensate for the monitor's nonlinear response. In a system that does not do gamma correction, the problem is that low voltages produce unnaturally dark intensities compared to high voltages. The result is that dark colors appear unusually dark. In order to correct this effect, modern monitors provide the capability of gamma correction. In order to achieve a desired intensity I, we instead aim to produce a corrected intensity:

$$I' = I^{1/\gamma}$$

which we display instead of I. Thus, when the gamma effect is taken into account, we will get the desired intensity.

Some graphics displays (like SGIs and Macs) provide an automatic (but typically partial) gamma correction. In most PC's the gamma can be adjusted manually. (However, even with gamma correction, do not be surprised if the same RGB values produce different colors on different systems.) There are resources on the Web to determine the gamma value for your monitor.

Light and Color: Light as we perceive it is *electromagnetic radiation* from a narrow band of the complete spectrum of electromagnetic radiation called the *visible spectrum*. The physical nature of light has elements that are like particle (when we discuss photons) and as a wave. Recall that wave can be described either in terms of its *frequency*, measured say in cycles per second, or the inverse quantity of *wavelength*. The electro-magnetic spectrum ranges from very low frequency (high wavelength) radio waves (greater than 10 centimeter in wavelength) to microwaves, infrared, visible light, ultraviolet and x-rays and high frequency (low wavelength) gamma rays (less than 0.01 nm in wavelength). Visible light lies in the range of wavelengths from around 400 to 700 nm, where *nm* denotes a nanometer, or 10^{-9} of a meter.

Physically, the light energy that we perceive as color can be described in terms of a function of wavelength λ, called the *spectral distribution function* or simply *spectral function*, $f(\lambda)$. As we walk along the wavelength axis (from long to short wavelengths), the associated colors that we perceive varying along the colors of the rainbow red, orange, yellow, green, blue, indigo, violet. (Remember the "Roy G. Biv" mnemonic.) Of course, these color names are human interpretations, and not physical divisions.

The Eye and Color Perception: Light and color are complicated in computer graphics for a number of reasons. The first is that the *physics* of light is very complex. Secondly, our *perception* of light is a function of our optical systems, which perform numerous unconscious corrections and modifications to the light we see.

The retina of the eye is a light sensitive membrane, which contains two types of light-sensitive receptors, *rods* and *cones*. Cones are color sensitive. There are three different types, which are selectively more sensitive to red, green, or blue light. There are from 6 to 7 million cones concentrated in the *fovea*, which corresponds to the center of your view. The *tristimulus theory* states that we perceive color as a mixture of these three colors.

Blue cones: peak response around 440 nm with about 2% of light absorbed by these cones.

Green cones: peak response around 545 nm with about 20% of light absorbed by these cones.

Red cones: peak response around 580 nm, with about 19% of light absorbed by these cones.

The different absorption rates comes from the fact that we have far fewer blue sensitive cones in the fovea as compared with red and green. Rods in contrast occur in lower density in the fovea, and do not distinguish color. However they are sensitive to low light and motion, and hence serve a function for vision at night.

It is possible to produce light within a very narrow band of wavelengths using lasers. Note that because of our limited ability to sense light of different colors, there are many different

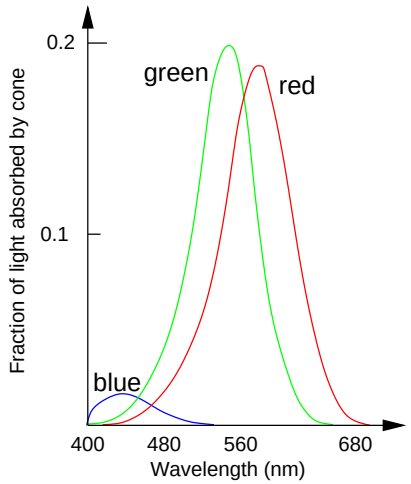


Fig. 98: Spectral response curves for cones (adapted from Foley, vanDam, Feiner and Hughes).

spectra that appear to us to be the same color. These are called *metamers*. Thus, spectrum and color are not in 1–1 correspondence. Most of the light we see is a mixture of many wavelengths combined at various strengths. For example, shades of gray varying from white to black all correspond to fairly flat spectral functions.

Describing Color: Throughout this semester we have been very lax about defining color carefully. We just spoke of RGB values as if that were enough. However, we never indicated what RGB means, independently from the notion that they are the colors of the phosphors on your display. How would you go about describing color precisely, so that, say, you could unambiguously indicate exactly what shade you wanted in a manner that is independent of the display device? Obviously you could give the spectral function, but that would be overkill (since many spectra correspond to the same color) and it is not clear how you would find this function in the first place.

There are three components to color, which seem to describe color much more predictably than does RGB. These are hue, saturation, and lightness. The *hue* describes the dominant wavelength of the color in terms of one of the pure colors of the spectrum that we gave earlier. The *saturation* describes how pure the light is. The red color of a fire-engine is highly saturated, whereas pinks and browns are less saturated, involving mixtures with grays. Gray tones (including white and black) are the most highly unsaturated colors. Of course lightness indicates the intensity of the color. But although these terms are somewhat more intuitive, they are hardly precise.

The tristimulus theory suggests that we perceive color by a process in which the cones of the three types each send signals to our brain, which sums these responses and produces a color. This suggests that there are three "primary" spectral distribution functions, $R(\lambda)$, $G(\lambda)$, and $B(\lambda)$, and every saturated color that we perceive can be described as a positive linear combination of these three:

$$C = rR + gG + bB \qquad \text{where } r, g, b \geq 0.$$

Note that R, G and B are functions of the wavelength λ, and so this equation means we weight each of these three functions by the scalars r, g, and b, respectively, and then integrate over the entire visual spectrum. C is the color that we perceive.

Extensive studies with human subjects have shown that it is indeed possible to define saturated colors as a combination of three spectra, but the result has a very strange outcome. Some colors can only be formed by allowing some of the coefficients r, g, or b to be negative. E.g., there is a color C such that

$$C = 0.7R + 0.5G - 0.2B.$$

We know what it means to form a color by adding light, but we cannot subtract light that is not there. The way that this equation should be interpreted is that we cannot form color C from the primaries, but we can form the color $C + 0.2B$ by combining $0.7R + 0.5G$. When we combine colors in this way they are no longer pure, or saturated. Thus such a color C is in some sense *super-saturated*, since it cannot be formed by a purely additive process.

The CIE Standard: In 1931, a commission was formed to attempt to standardize the science of colorimetry. This commission was called the Commission Internationale de l'Éclairage, or CIE.

The results described above lead to the conclusion that we cannot describe all colors as positive linear combinations of three primary colors. So, the commission came up with a standard for describing colors. They defined three special *super-saturated* primary colors X, Y, and Z, which do not correspond to any real colors, but they have the property that every real color can be represented as a positive linear combination of these three.

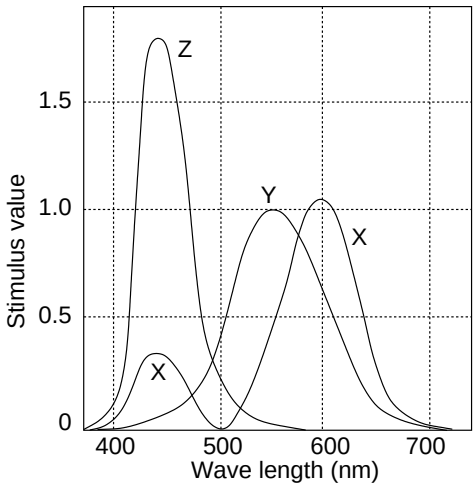


Fig. 99: CIE primary colors (adapted from Hearn and Baker).

The resulting 3-dimensional space, and hence is hard to visualize. A common way of drawing the diagram is to consider a single 2-dimensional slice, by normalize by cutting with the plane $X + Y + Z = 1$. We can then project away the Z component, yielding the *chromaticity coordinates*:

$$x = \frac{X}{X + Y + Z} \qquad y = \frac{Y}{X + Y + Z}$$

(and z can be defined similarly). These components describe just the color of a point. Its brightness is a function of the Y component. (Thus, an alternative, but seldom used, method of describing colors is as xyY.)

If we plot the various colors in this (x, y) coordinates produce a 2-dimensional "shark-fin" convex shape shown in Fig. 100. Let's explore this figure a little. Around the curved top of the shark-fin we see the colors of the spectrum, from the long wavelength red to the short wavelength violet. The top of the fin is green. Roughly in the center of the diagram is white. The point C corresponds nearly to "daylight" white. As we get near the boundaries of the diagram we get the purest or most *saturated* colors (or *hues*). As we move towards C, the colors become less and less saturated.

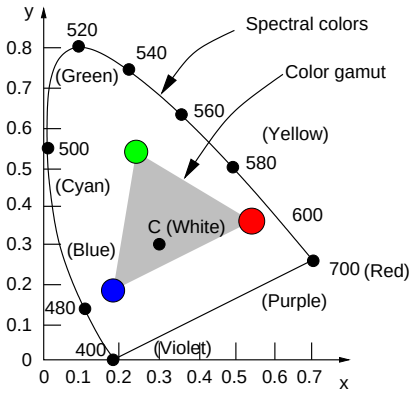


Fig. 100: CIE chromaticity diagram and color gamut (adapted from Hearn and Baker).

An interesting consequence is that, since the colors generated by your monitor are linear combinations of three different colored phosphors, there exist regions of the CIE color space that your monitor cannot produce. (To see this, find a bright orange sheet of paper, and try to imitate the same saturated color on your monitor.)

The CIE model is useful for providing formal specifications of any color as a 3-element vector. Carefully designed image formats, such as TIFF and PNG, specify colors in terms of the CIE model, so that, in theory, two different devices can perform the necessary corrections to display the colors as true as possible. 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.

Supplemental Lecture 6: Halftone Approximation

Halftone Approximation: Not all graphics devices provide a continuous range of intensities. Instead they provide a discrete set of choices. The most extreme case is that of a monochrom display with only two colors, black and white. Inexpensive monitors have look-up tables (LUT's) with only 256 different colors at a time. Also, when images are compressed, e.g. as in the gif format, it is common to reduce from 24-bit color to 8-bit color. The question is,

how can we use a small number of available colors or shades to produce the perception of many colors or shades? This problem is called *halftone approximation*.

We will consider the problem with respect to monochrome case, but the generalization to colors is possible, for example by treating the RGB components as separate monochrome subproblems.

Newspapers handle this in reproducing photographs by varying the dot-size. Large black dots for dark areas and small black dots for white areas. However, on a CRT we do not have this option. The simplest alternative is just to round the desired intensity to the nearest available gray-scale. However, this produces very poor results for a monochrome display because all the darker regions of the image are mapped to black and all the lighter regions are mapped to white.

One approach, called *dithering*, is based on the idea of grouping pixels into groups, e.g. 3×3 or 4×4 groups, and assigning the pixels of the group to achieve a certain affect. For example, suppose we want to achieve 5 halftones. We could do this with a 2×2 dither matrix.

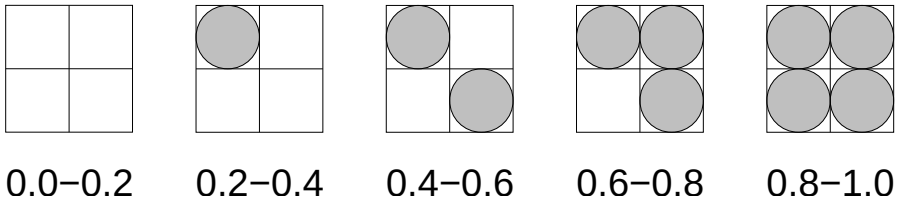


Fig. 101: Halftone approximation with dither patterns.

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.

Supplemental Lecture 7: 3-d Rotation and Quaternions

Rotation and Orientation in 3-Space: One of the trickier problems 3-d geometry is that of parameterizing rotations and the orientation of frames. We have introduced the notion of orientation before (e.g., clockwise or counterclockwise). Here we mean the term in a somewhat different sense, as a directional position in space. Describing and managing rotations in 3-space is a somewhat more difficult task (at least conceptually), compared with the relative simplicity of rotations in the plane.

Why do we care about rotations? Suppose that you are an animation programmer for a computer graphics studio. The object that you are animating is to be moved smoothly from one location to another. If the object is in the same directional orientation before and after, we can just translate from one location to the other. If not, we need to find a way of interpolating between its two orientations. This usually involves rotations in 3-space. But how should these rotations be performed so that the animation looks natural? Another example is one in which the world is stationary, but the camera is moving from one location and viewing situation to another. Again, how can we move smoothly and naturally from one to the other?

Since smoothly interpolating positions by translation is pretty easy to understand, let us ignore the issue of position, and just focus on orientations and rotations about the origin. Let F denote the standard coordinate frame, and consider another orthonormal frame G. We

Fig. 102: Floyd-Steinberg Algorithm (Source: Poulbére and Bousquet, 1999).

want some way to represent G concisely, relative to F, and generally to interpolate a motion from F to G (see Fig. 103).

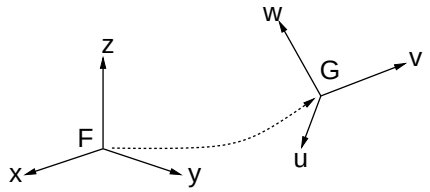


Fig. 103: Moving between frames.

Of course, we could just represent F and G by their three orthonormal basis vectors. But if we were to try to interpolate (linearly) between corresponding pairs of basis vectors, the intermediate vectors would not necessarily be orthonormal.

We will explore two methods for dealing with rotation, *Euler angles* and *quaternions*.

Euler Angles: Leonard Euler was a famous mathematician who lived in the 18th century. He proved many important theorems in geometry, algebra, and number theory, and he is credited as the inventor of graph theory. Among his many theorems is one that states that the composition any number of rotations in three-space can be expressed as a single rotation in 3-space about an appropriately chosen vector. Euler also showed that any rotation in 3-space could be broken down into exactly three rotations, one about each of the coordinate axes.

Suppose that you are a pilot, such that the x-axis points to your left, the y-axis points ahead of you, and the z-axis points up (see Fig. 104). Then a rotation about the x-axis, denoted by ϕ, is called the *pitch*. A rotation about the y-axis, denoted by θ, is called *roll*. A rotation about the z-axis, denoted by ψ, is called *yaw*. Euler's theorem states that any position in space can be expressed by composing three such rotations, for an appropriate choice of (ϕ, θ, ψ).

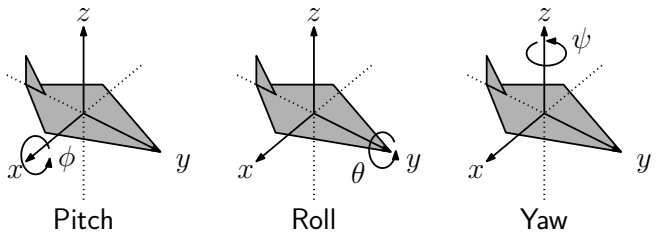


Fig. 104: Euler angles: pitch, roll, and yaw.

Let's explore how we can derive this result. Consider the orthonormal frame G, described earlier. Suppose that we want to rotate the standard frame F so that its three coordinate vectors coincide with G's respective coordinate vectors. Let us consider the process in reverse. We will see how to rotate G so that it coincides with the standard frame. The desired transformation from F to G comes about by reversing the process.

This process is easier to describe than it is to visualize. Suppose that we label G's basis vectors u, v and w.

Pitch: Let w' denote the projection of w onto the yz-coordinate plane. First rotate about the x-axis, until the vector w' coincides with the z-axis (see Fig. 105(a)). Call this angle ϕ. The original vector w will now lie on the xz-coordinate plane.

Roll: Next, rotate about the y-axis (thus keeping the xz-coordinate plane fixed) until w coincides with the z-axis (see Fig. 105(b)). Call this angle θ. Afterwards, the two vectors u and v (being orthogonal to w) must now lie on the xy-plane.

Yaw: Finally, we rotate about the z-axis until u coincides with the x-axis (see Fig. 105(c)). Call this angle ψ. At this point we have $w = z$ and $u = x$. Assuming that G is orthonormal and right-handed, it follows that $v = y$.

Thus, by these three rotations (ϕ, θ, ψ), one about each of the axes, we can bring G into alignment with F.

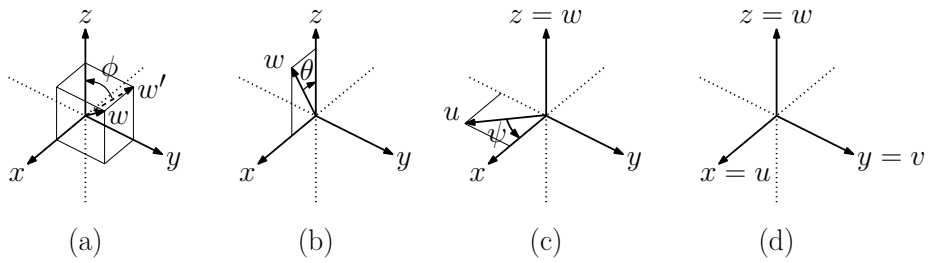


Fig. 105: Rotating a frame to coincide with the standard frame.

In summary, we have established Euler's theorem. A change in orientation between any two orthonormal frames can be accomplished with three rotations, one about each of the coordinate axes. Hence, such a transformation can be represented by a triple of three angles, (ϕ, θ, ψ). These define a general rotation matrix, by composing the three basic rotations:

$$R(\phi, \theta, \psi) = R_z(\psi) R_y(\theta) R_x(\phi).$$

(As usual, recall that, because we post-multiply vectors times matrices, it follows that the x-rotation is performed first, followed by the y-rotation, and the z-rotation.) Thus, these three angles are the *Euler angles* for the rotation that aligns G with F.

Interpolating using Euler angles: Now, given two orientations in space, say given by the Euler angles $E_1 = (\phi_1, \theta_1, \psi_1)$ and $E_2 = (\phi_2, \theta_2, \psi_2)$, we can interpolate between then, say by taking convex combinations. Given any $\alpha \in [0, 1]$, we can define

$$R(\alpha) = R((1 - \alpha)E_1 + \alpha E_2),$$

for example. As α varies from 0 to 1, this will smoothly rotate from one orientation to the other. In practice, this approach works fairly well if the two orientations are very close to each other. It should be noted, however, that the interplations resulting from Euler angles are unintuitive, and in particular, Euler-angle interpolation does not follow the shortest path between two rotations. (This is remedied by quaternions, which will discuss later.)

Shortcomings of Euler angles: There are some problems with Euler angles. One issue is the fact that this representation depends on the choice of coordinate system. In the plane, a 30 degree rotation is the same, no matter what direction the axes are pointing (as long as they are orthonormal and right-handed). However, the result of an Euler-angle rotation depends very much on the choice of the coordinate frame and on the order in which the axes are named. (Later, we will see that quaternions do provide such an intrinsic system.)

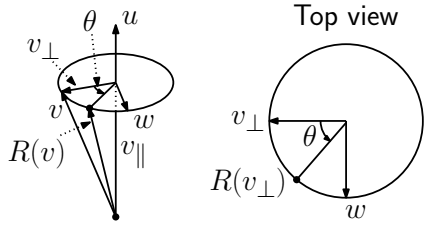
Another problem with Euler angles is called *gimbal lock*. Whenever we rotate about one axis, it is possible that we could bring the other two axes into alignment with each other. (This happens, for example if we rotate x by $90°$.) This causes problems because the other two axes no longer rotate independently of each other, and we effectively lose one degree of freedom. Gimbal lock as induced by one ordering of the axes can be avoided by changing the order in which the rotations are performed. But, this is rather messy, and it would be nice to have a system that is free of this problem.

Angular Displacement: Let us next consider an approach to rotation that is invariant under rigid changes of the coordinate system. This will eventually lead us to the concept of a quaternion.

In contrast to Euler angles, a more intrinsic way to express rotations (about the origin) in 3-space is in terms of two quantities, (θ, u), consisting of an angle θ, and an axis of rotation u. Let's consider how we might do this. First consider a vector v to be rotated. Let us assume that u is of unit length.

Our goal is to describe the rotation of a vector v as a function of θ and u. Let $R(v)$ denote this rotated vector (see Fig. 106(a)). In order to derive this, we begin by decomposing v as the sum of its components that are parallel to and orthogonal to u, respectively.

$$v_\parallel = (u \cdot v)u \qquad v_\perp = v - v_\parallel = v - (u \cdot v)u.$$



(a) (b)

Fig. 106: Angular displacement.

Note that $v_\parallel$ is unaffected by the rotation, but $v_\perp$ is rotated to a new position $R(v_\perp)$. To determine this rotated position, we will first construct a vector that is orthogonal to $v_\perp$ lying in the plane of rotation.

$$w = u \times v_\perp = u \times (v - v_\parallel) = (u \times v) - (u \times v_\parallel) = u \times v.$$

The last step follows from the fact that u and $v_\parallel$ are parallel, and so the cross product is zero. Clearly w is orthogonal to both $v_\perp$ and u. Furthermore, because $v_\perp$ is orthogonal to the unit

vector u, it follows from basic properties of the cross product that w is the same length as $v_\perp$.

Now, consider the plane spanned by $v_\perp$ and w (see Fig. 106(b)). We have

$$R(v_\perp) = (\cos\theta)v_\perp + (\sin\theta)w.$$

From this and the fact that $R(v_\parallel) = v_\parallel$, we have

$$\begin{aligned} R(v) &= R(v_\parallel) + R(v_\perp) \\ &= v_\parallel + (\cos\theta)v_\perp + (\sin\theta)w \\ &= (u \cdot v)u + (\cos\theta)(v - (u \cdot v)u) + (\sin\theta)w \\ &= (\cos\theta)v + (1 - \cos\theta)u(u \cdot v) + (\sin\theta)(u \times v). \end{aligned}$$

In summary, we have the following formula expressing the effect of the rotation of vector v by angle θ about a rotation axis u:

$$R(v) = (\cos\theta)v + (1 - \cos\theta)u(u \cdot v) + (\sin\theta)(u \times v). \tag{3}$$

This expression is the image of v under the rotation. Notice that, unlike Euler angles, this is expressed entirely in terms of intrinsic geometric functions (such as dot and cross product), which do not depend on the choice of coordinate frame. This is a major advantage of this approach over Euler angles.

Quaternions: We will now delve into a subject, which at first may seem quite unrelated. But keep the above expression in mind, since it will reappear in most surprising way. This story begins in the early 19th century, when the great mathematician William Rowan Hamilton was searching for a generalization of the complex number system.

Imaginary numbers can be thought of as linear combinations of two basis elements, 1 and i, which satisfy the multiplication rules $1^2 = 1$, $i^2 = -1$ and $1 \cdot i = i \cdot 1 = i$. (The interpretation of $i = \sqrt{-1}$ arises from the second rule.) A complex number $a + bi$ can be thought of as a vector in 2-dimensional space (a, b). Two important concepts with complex numbers are the *modulus*, which is defined to be $\sqrt{a^2 + b^2}$, and the *conjugate*, which is defined to be $(a, -b)$. In vector terms, the modulus is just the length of the vector and the conjugate is just a vertical reflection about the x-axis. If a complex number is of modulus 1, then it can be expressed as $(\cos\theta, \sin\theta)$. Thus, there is a connection between complex numbers and 2-dimensional rotations. Also, observe that, given such a unit modulus complex number, its conjugate is $(\cos\theta, -\sin\theta) = (\cos(-\theta), \sin(-\theta))$. Thus, taking the conjugate is something like negating the associated angle.

Hamilton was wondering whether this idea could be extended to three dimensional space. You might reason that, to go from 2D to 3D, you need to replace the single imaginary quantity i with two imaginary quantities, say i and j. Unfortunately, this this idea does not work. After years of work, Hamilton came up with the idea of, rather than using two imaginaries, instead using three imaginaries i, j, and k, which behave as follows:

$$i^2 = j^2 = k^2 = ijk = -1 \qquad ij = k,\ jk = i,\ ki = j.$$

Combining these, it follows that $ji = -k$, $kj = -i$ and $ik = -j$. The skew symmetry of multiplication (e.g., $ij = -ji$) was actually a major leap, since multiplication systems up to that time had been commutative.)

Hamilton defined a *quaternion* to be a generalized complex number of the form

$$\mathbf{q} = q_0 + q_1 i + q_2 j + q_3 k.$$

Thus, a quaternion can be viewed as a 4-dimensional vector $\mathbf{q} = (q_0, q_1, q_2, q_3)$. The first quantity is a scalar, and the last three define a 3-dimensional vector, and so it is a bit more intuitive to express this as $\mathbf{q} = (s, u)$, where $s = q_0$ is a scalar and $u = (q_1, q_2, q_3)$ is a vector in 3-space. We can define the same concepts as we did with complex numbers:

Conjugate: $\mathbf{q}^* = (s, -u)$.
Modulus: $|\mathbf{q}| = \sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2} = \sqrt{s^2 + (u \cdot u)}$.
Unit Quaternion: $\mathbf{q}$ is said to be a unit quaternion if $|\mathbf{q}| = 1$.

Quaternion multiplication: Consider two quaternions $\mathbf{q} = (s, u)$ and $\mathbf{p} = (t, v)$:

$$\begin{aligned} \mathbf{q} &= (s, u) = s + u_x i + u_y j + u_z k \\ \mathbf{p} &= (t, v) = t + v_x i + v_y j + v_z k. \end{aligned}$$

If we multiply these two together, we'll get lots of cross-product terms, such as $(u_x i)(v_y j)$, but we can simplify these by using Hamilton's rules. That is, $(u_x i)(v_y j) = u_x v_h (ij) = u_x v_h k$. If we do this, simplify, and collect common terms, we get a very messy formula involving 16 different terms (see the appendix at the end of this lecture). The formula can be expressed somewhat succinctly in the following form:

$$\mathbf{qp} = (st - (u \cdot v), sv + tu + u \times v).$$

Note that the above expression is in the quaternion scalar-vector form. The first term $st - (u \cdot v)$ evaluates to a scalar (recalling that the dot product returns a scalar), and the second term $(sv + tu + u \times v)$ is a sum of three vectors, and so is a vector. It can be shown that quaternion multiplication is associative, but not commutative.

Quaternion multiplication and 3-dimensional rotation: Before considering rotations, we first define a *pure quaternion* to be one with a 0 scalar component

$$\mathbf{p} = (0, v).$$

Any quaternion of nonzero magnitude has a multiplicative *inverse*, which is defined to be

$$\mathbf{q}^{-1} = \frac{1}{|\mathbf{q}|^2}\mathbf{q}^*.$$

(To see why this works, try multiplying $\mathbf{q}\mathbf{q}^{-1}$, and see what you get.) Observe that if $\mathbf{q}$ is a unit quaternion, then it follows that $\mathbf{q}^{-1} = \mathbf{q}^*$.

As you might have guessed, our objective will be to show that there is a relation between rotating vectors and multiplying quaternions. In order apply this insight, we need to first show how to represent rotations as quaternions and 3-dimensional vectors as quaternions. After a bit of experimentation, the following does the trick:

Vector: Given a vector $v = (v_x, v_y, v_z)$ to be rotated, we will represent it by the pure quaternion $(0, v)$.

Rotation: To represent a rotation by angle θ about a unit vector u, you might think, we'll use the scalar part to represent θ and the vector part to represent u. Unfortunately, this doesn't quite work. After a bit of experimentation, you will discover that the right way to encode this rotation is with the quaternion $\mathbf{q} = (\cos(\theta/2), (\sin(\theta/2))u)$. (You might wonder, why we do we use $\theta/2$, rather than θ. The reason, as we shall see below, is that "this is what works.")

Rotation Operator: Given a vector v represented by the quaternion $\mathbf{p} = (0, v)$ and a rotation represented by a unit quaternion $\mathbf{q}$, we define the *rotation operator* to be:

$$R_{\mathbf{q}}(\mathbf{p}) \;=\; \mathbf{q}\mathbf{p}\mathbf{q}^{-1} \;=\; \mathbf{q}\mathbf{p}\mathbf{q}^{*}.$$

(The last equality results from the fact that $\mathbf{q}^{-1} = \mathbf{q}^{*}$, if $\mathbf{q}$ is a unit quaternion). We claim that the result of this operation will always be a unit quaternion, and so it is possible to interpret the result as a vector. In particular, this vector will be the result of applying the rotation $\mathbf{q}$ to v.

Why does this work? Let's begin by applying the multiplication rule and use the fact that $\mathbf{q}^{-1} = \mathbf{q}^{*}$ for a unit quaternion $\mathbf{q} = (s, u)$. Given $\mathbf{p} = (0, v)$, by expanding the rotation operator definition and simplifying we obtain:

$$R_{\mathbf{q}}(\mathbf{p}) \;=\; (0,\; (s^2 - (u \cdot u))v + 2u(u \cdot v) + 2s(u \times v)). \tag{4}$$

(We leave the derivation as an exercise, but a few nontrivial facts regarding dot products and cross products need to applied.)

Let us see if we can express this in a more suggestive form. Since $\mathbf{q}$ is of unit magnitude, we can express it as

$$\mathbf{q} = \left(\cos\frac{\theta}{2}, \left(\sin\frac{\theta}{2}\right) u\right), \qquad \text{where } \|u\| = 1.$$

Plugging this into the above expression and applying some standard trigonometric identities, we obtain

$$\begin{aligned} R_{\mathbf{q}}(\mathbf{p}) &= \left(0, \left(\cos^2\frac{\theta}{2} - \sin^2\frac{\theta}{2}\right) v + 2\left(\sin^2\frac{\theta}{2}\right) u(u \cdot v) + 2\cos\frac{\theta}{2}\sin\frac{\theta}{2}(u \times v)\right) \\ &= (0,\; (\cos\theta)v + (1 - \cos\theta)u(u \cdot v) + \sin\theta(u \times v)). \end{aligned}$$

Now, recall the rotation displacement equation presented earlier in the lecture. The vector part of this quaternion is *identical*, implying that the quaternion rotation operator achieves the desired rotation.

Example: Consider the 3-d rotation shown in Fig. 107. This rotation can be achieved by performing a rotation about the y-axis by $\theta = -90$ degrees. Thus $\theta = -\pi/2$, and $u = (0, 1, 0)$. Thus the quaternion that encodes this rotation is

$$\mathbf{q} \;=\; (\cos(\theta/2), (\sin(\theta/2))u) \;=\; \left(\cos\left(-\frac{\pi}{4}\right), \sin\left(-\frac{\pi}{4}\right)(0,1,0)\right) \;=\; \left(\frac{1}{\sqrt{2}}, \left(0, -\frac{1}{\sqrt{2}}, 0\right)\right).$$

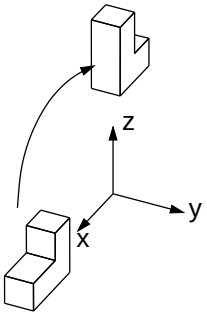


Fig. 107: Rotation example.

Let us consider how the x-unit vector $v = (1, 0, 0)^{\mathsf{T}}$ is transformed under this rotation. To reduce this to a quaternion operation, we encode v as a pure quaternion $\mathbf{p} = (0, v) = (0, (1, 0, 0))$. We then apply the rotation operator, and so by Eq. (4) we have

$$\begin{aligned} R_{\mathbf{q}}(\mathbf{p}) &= (0,\ (1/2 - 1/2)(1,0,0) + 2(0,1,0)0 + (2/\sqrt{2})((0,-1/\sqrt{2},0) \times (1,0,0))) \\ &= (0, (0,0,0) + (0,0,0) + (-1)(0,0,-1)) \\ &= (0, (0,0,1)). \end{aligned}$$

Thus p is mapped to a point on the z-axis, as expected.

Composing Rotations: We have shown that each unit quaternion corresponds to a rotation in 3-space. This is an elegant representation, but can we manipulate rotations through quaternion operations? The answer is yes. In particular, the action of multiplying two unit quaternions results in another unit quaternion. Furthermore, the resulting product quaternion corresponds to the composition of the two rotations. In particular, given two unit quaternions $\mathbf{q}$ and $\mathbf{q}'$, a rotation by $\mathbf{q}$ followed by a rotation by $\mathbf{q}'$ is equivalent to a single rotation by the product $\mathbf{q}'' = \mathbf{q}'\mathbf{q}$. That is,

$$R_{\mathbf{q}'} R_{\mathbf{q}} = R_{\mathbf{q}''} \qquad \text{where } \mathbf{q}'' = \mathbf{q}'\mathbf{q}.$$

This follows from the associativity of quaternion multiplication, and the fact that $(\mathbf{q}\mathbf{q}')^{-1} = \mathbf{q}^{-1}\mathbf{q}'^{-1}$, as shown below.

$$\begin{aligned} R_{\mathbf{q}'}(R_{\mathbf{q}}(\mathbf{p})) &= \mathbf{q}'(\mathbf{q}\mathbf{p}\mathbf{q}^{-1})\mathbf{q}'^{-1} = (\mathbf{q}'\mathbf{q})\mathbf{p}(\mathbf{q}^{-1}\mathbf{q}'^{-1}) \\ &= (\mathbf{q}'\mathbf{q})\mathbf{p}(\mathbf{q}\mathbf{q}')^{-1} = \mathbf{q}''\mathbf{p}\mathbf{q}''^{-1} \\ &= R_{\mathbf{q}''}(\mathbf{p}). \end{aligned}$$

Lerp, nlerp, and slerp: (No, these are not cartoon characters nor a new drink at 7-Eleven.) An important question is, given two unit quaternions $\mathbf{q}_0$ and $\mathbf{q}_1$, how can we smoothly interpolate between them? We need this in order to perform smooth animations involving rotating objects.

We have already learned about one means of interpolation, called *linear interpolation* (or *lerp* for short). Given two points p_0 and p_1, for any α, where $0 \le \alpha \le 1$, we can express a point between p_0 and p_1 as the affine combination

$$lerp(p_0, p_1; \alpha) = (1 - \alpha)p_0 + \alpha p_1.$$

As α ranges from 0 to 1, the value of $lerp(p_0, p_1; \alpha)$ varies linearly from p_0 to p_1 (see Fig. 108).

As the name suggests, this interpolates between p_0 and p_1 along a straight line between the two points. In some cases, however, these may be points on a sphere, and rather than have the interpolation pass through the interior of the sphere, we would like to interpolation to follow the shortest path along the surface of the sphere. To do this, we need a different sort of interpolation, a spherical interpolation.

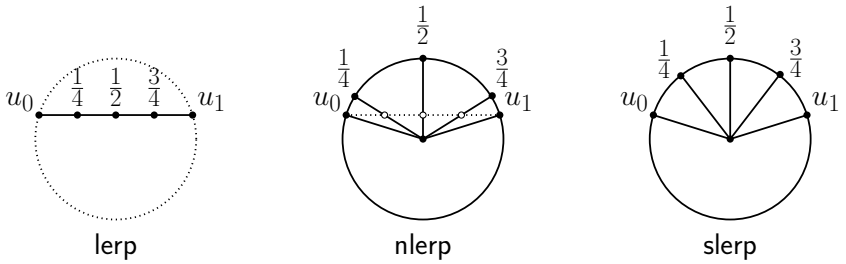


Fig. 108: Lerp, nlerp, and slerp.

To develop the notion of a spherical interpolation, suppose that u_0 and u_1 are two unit vectors, which we can think of points on a unit sphere. There are two ways to perform a spherical interpolation. The first starts with a linear interpolation. Since such an interpolation passes through the interior of the sphere, in order to get the point to lie on the sphere, we simply normalize it to unit length. The result is called a *normalized linear interpolation* (or *nlerp* for short). Here is the formula.

$$nlerp(p_0, p_1; \alpha) \;=\; normalize(lerp(p_0, p_1; \alpha) \;=\; normalize((1-\alpha)p_0 + \alpha p_1).$$

Recall that the function $normalize(u)$ simply divides u by its length, $u/\|u\|$, thus always producing a unit vector.

The nlerp produces very reasonable results when the amount of rotation is small. It has one defect, however, in that, if the two points are far apart from each other (e.g., one close to the north pole and one close to the south pole) then the motion is not constant along the sphere. It moves much more rapidly in the middle of the arc than at the ends (see Fig. 108). To fix this, we need a truly spherical approach to interpolation. This is called the *spherical interpolation* (or *slerp* for short). To define it, let $\omega = \arccos(u_0 \cdot u_1)$ denote the angle between the unit vectors u_0 and u_1. For $0 \le \alpha \le 1$, we define

$$slerp(p_0, p_1; \alpha) \;=\; \frac{\sin(1-\alpha)\omega}{\sin\omega} u_0 + \frac{\sin\alpha\omega}{\sin\omega} u_1.$$

Although, it is not obvious why this works, this produces a path along the sphere that has constant velocity from u_0 to u_1, as α varies from 0 to 1 (see Fig. 108).

Interpolating Quaternions: When interplating rotations represented as unit quaternions, we can use either nlerp or slerp. The nlerp definition for quaternions is identical to the one given above (it is just applied in 4-dimensional space).

When slerping between quaternions, however, we need to be aware of one issue. In particular, recall that when we encode a rotation by angle θ as a quaternion, the scalar and vector

components are multiplied by $\cos(\theta/2)$ and $\sin(\theta/2)$, respectively. This implies that, if we have two unit quaternions $\mathbf{q}_0$ and $\mathbf{q}_1$, which involve a relative rotation angle of θ between them, then the 4-dimensional dot product $(\mathbf{q}_0 \cdot \mathbf{q}_1)$ produces the arccosine of $\theta/2$, not θ. For this reason, it is recommended that, if $(\mathbf{q}_0 \cdot \mathbf{q}_1) < 0$ (implying that $\theta \geq \pi$) then replace $\mathbf{q}_1$ with $-\mathbf{q}_1$. (Note that $\mathbf{q}_1$ and $-\mathbf{q}_1$ are equivalent rotations.)

Rather than implementing your own quaternion slerp, I would suggest downloading code from the web to do this. There are computationally more efficient versions of slerp, which are not directly based on the above formula.

Matrices and Quaternions: Quaternions provide a very elegant way of representing rotations in 3-space. Returning to the problem of interpolating smoothly between two orientations, we can see that we can describe the before and after orientations of any object by two quaternions, $\mathbf{q}$ and $\mathbf{p}$. Then, to interpolate smoothly between these two orientations, we spherically interpolate between $\mathbf{q}$ and $\mathbf{p}$ in quaternion space.

However, once we have a quaternion representation, we need a way to inform our graphics API (like OpenGL) about the actual transformation. In particular, given a unit quaternion

$$\mathbf{q} \;=\; \left(\cos\frac{\theta}{2}, \sin\frac{\theta}{2}\,u\right) \;=\; (s, (u_x, u_y, u_z)),$$

what is the corresponding affine transformation (expressed as a rotation matrix). By simply expanding the definition of of $R_{\mathbf{q}}(\mathbf{p})$, it is not hard to show that the following (homogeneous) matrix is equivalent

$$\begin{pmatrix} 1-2u_y^2-2u_z^2 & 2u_xu_y-2su_z & 2u_xu_z+2su_y & 0 \\ 2u_xu_y+2su_z & 1-2u_x^2-2u_z^2 & 2u_yu_z-2su_x & 0 \\ 2u_xu_z-2su_y & 2u_yu_z+2su_x & 1-2u_x^2-2u_y^2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Thus, given your quaternion interpolant, you apply this rotation by invoking `glMultMatrix()`, and all subsequently drawn points will be rotated in accordance with the quaternion.

Quaternion Summary: In summary, quaternions are a generalization of the concept of complex numbers, which can be used to represent rotations in three dimensional space. Unlike Euler angles, quaternions are independent of the coordinate system. Also, they do not suffer from the problem of gimbal lock. Thus, from a mathematical perspective, they represent a much cleaner system for representing rotations.

- Quaternions can be used to represent the rotation (orientation) of an object in 3-dimensional space.
- A rotation by a given angle θ about a unit vector u can be represented by the unit quaternion $\mathbf{q} = (\cos(\theta/2), (\sin(\theta/2))u)$.
- A vector v is represented by the pure quaternion $\mathbf{p} = (0, v)$.
- The effect of applying this rotation to v is given by $R_{\mathbf{q}}(\mathbf{p}) \;=\; \mathbf{q}\mathbf{p}\mathbf{q}^*$. This is a pure quaternion, and so can be mapped back to a vector by discarding the scalar part.

- Given two rotation quaternions $\mathbf{q}$ and $\mathbf{q}'$, the product $\mathbf{q}'\mathbf{q}$ corresponds to applying $\mathbf{q}$ followed by $\mathbf{q}'$.
- Given two rotation quaternions $\mathbf{q}_0$ and $\mathbf{q}_1$, it is possible to interpolate smoothly between them using either the nlerp (normalized linear interpolation) or slerp (spherical interpolation).

Appendix (Deriving quaternion multiplication): Earlier, we stated that, given two quaternions:

$$\begin{aligned}\mathbf{q} &= (s,u) = s + u_x i + u_y j + u_z k \\ \mathbf{p} &= (t,v) = t + v_x i + v_y j + v_z k,\end{aligned}$$

their product is given by

$$\mathbf{qp} = (st - (u \cdot v), sv + tu + u \times v).$$

In this appendix, we derive this fact. Although, this is just a rather tedious exercise in algebra, it is nice to see that everything follows from the basic laws that Hamilton discovered for the multiplication of the quaternion imaginaries i, j and k. (And perhaps it explains why Hamilton was so excited when he realized that he got it right!)

First, let us recall that, given two vectors u and v, the definitions of the dot and cross products are:

$$\begin{aligned}(u \cdot v) &= u_x v_x + u_y v_y + u_z v_z \\ u \times v &= (u_y v_z - u_z v_y, u_z v_x - u_x v_z, u_x v_y - u_y v_x)^{\mathsf{T}}.\end{aligned}$$

Given this, let's start by multiplying $\mathbf{q}$ and $\mathbf{p}$. Multiplication between scalars and imaginaries is commutative ($si = is$) but multiplication between imaginaries is not ($ij \neq ji$), so we need to be careful to preserve the order of the imaginary quantities.

$$\begin{aligned}\mathbf{qp} &= (s + u_x i + u_y j + u_z k)(t + v_x i + v_y j + v_z k) \\ &= (st + sv_x i + sv_y j + sv_z k) + (u_x t i + u_x v_x i^2 + u_x v_y ij + u_x v_z ik) + \\ &\quad (u_y t j + u_y v_x ji + u_y v_y j^2 + u_y v_z jk) + (u_z t k + u_z v_x ki + u_z v_y kj + u_z v_z k^2).\end{aligned}$$

Next, let us apply the multiplication rules for the imaginary quantities. Recall that

$$i^2 = j^2 = k^2 = -1 \qquad ij = -(ji) = k, \quad jk = -(kj) = i, \quad ki = -(ik) = j.$$

Using these, we can express the product as

$$\begin{aligned}\mathbf{qp} &= (st + sv_x i + sv_y j + sv_z k) + (u_x t i - u_x v_x + u_x v_y k - u_x v_z j) + \\ &\quad (u_y t j - u_y v_x k - u_y v_y + u_y v_z i) + (u_z t k + u_z v_x j - u_z v_y i - u_z v_z).\end{aligned}$$

Now, let us collect common terms based on i, j, and k.

$$\begin{aligned}\mathbf{qp} &= (st - u_x v_x - u_y v_y - u_z v_z) + \\ &\quad (sv_x + u_x t + u_y v_z - u_z v_y) i + \\ &\quad (sv_y + u_y t - u_x v_z + u_z v_x) j + \\ &\quad (sv_z + u_z t + u_x v_y - u_y v_x) k.\end{aligned}$$

Observe that right-hand side above is a valid quaternion. The first (scalar) term can be expressed more succintly as $st - (u \cdot v)$. The other three terms share a common structure. If we think of them as the components of the three-element vector whose components are the i, j, and k terms, respectively, they can be simplified to

$$s \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} + t \begin{pmatrix} u_x \\ u_y \\ u_z \end{pmatrix} + \begin{pmatrix} +u_y v_z - u_z v_y \\ -u_x v_z + u_z v_x \\ +u_x v_y - u_y v_x \end{pmatrix} = sv + tu + u \times v.$$

Finally, if we put the scalar and vector parts together, we obtain

$$\mathbf{qp} = (st - (u \cdot v), sv + tu + u \times v),$$

just as desired. (Whew!)

Supplemental Lecture 8: Physically-Based Modeling

Physical Modeling: Traditionally, computer graphics was just about producing images from geometric models. Over recent years, the field has grown to encompass comptuational aspects of many other areas. The need for understanding physics grew from a need to produce realistic renderings of physical phenomena, such as moving clouds, flowing water, and breaking glass.

The good news to programmers is that there exist software systems that provide basic physical simulations. However, in order to use these systems, it is necessary to understand the basic elements of physics. It is also important to understand a bit about how these systems work, in order to understand what tasks they perform well, and what tasks they struggle with.

Basic Physics Concepts: Let us begin by discussing the basic elements of physics, which every graphics programmer needs to know.

Kinematics: This is the study of motion (ignoring forces). For example, it considers questions like: How does acceleration affect velocity? How does velocity affect position? There are two common models that are considered in kinematics:

Particles: This involves the motions of point masses. In particular, body rotation is ignored and only translation is considered. At first this may seem rather restrictive, but many complex phenomena, such as dust and water, can be modeled by simulating the motion of the massive number of individual particles.

Rigid bodies: For objects that are not points, the rotation of the body needs to be considered.

Force: Understanding forces and the effects they have on objects is central to physical modeling. Objects change their motion only when forces are applied. There are a number of different types of forces (see Fig. 109):

Contact forces and Torques: Contact forces arise when one or more objects collide with each other, such as a bowling ball striking a bowling pin. Torques refer to a special designation of contact forces that induce rotation, such as turning a steering wheel.

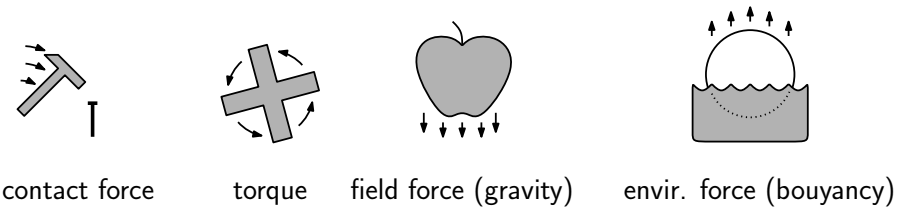


Fig. 109: Types of forces.

Field forces: These are forces like gravity or magnetism, which act without any explicit contact occuring.

Environmental forces: These include forces that are induced by the medium (air or water) in which the object resides or the surface on which the object is placed. These include friction, buoyancy (in water or air), and drag and lift for airplanes.

Kinetics: (also called Dynamics) In contrast to kinematics, which considers motion independent of force, kinematics explains how forces effect motion. The study of kinetics can be further decomposed into the nature of the object on which the kinetics is being applied:

Rigid Bodies: This means that a body moves (translates and rotates) as a single unit (e.g., a rock hurdling through the air).

Non-rigid Objects: These are bodies composed of multiple parts that move semi-independently, although the movement of one part may influence the movement of other parts. Examples include jointed-assemblies (like the joints and bones making up the human body), mass-spring systems (like the cloths that make up a fabric), hair and rope, water and soft plastics.

Physical Properties: Basic physics is about how forces affect the motion of objects. Forces induce acceleration, acceleration changes velocity, and velocity dictates motion. The manner in which forces affect an object depends on simple properties of the object. For a rigid body, the following quantities are important.

Mass: This is a scalar quantity, which describes the amount of "stuff" there is to an object. More practically, mass indicates the degree to which an object resists a change in its velocity. This is sometimes refered to as an object's *inertial mass*.

Formally, letting B be our body, we can define the mass by integrating the total volume of an object times its density per unit area. Let $dV = dx \times dy \times dz$ denote a differential volume element (an infinitesimally small cube at the point (x, y, z)) and let us assume that the object is of unit density. Letting m denote mass, we have

$$m = \int_B dV.$$

Center of mass (or center of gravity): This is a point about which all rotations occurs (assuming that the object is not tied down to anything). Formally, it is defined to be

the point $c = (c_x, c_y, c_z)^\mathsf{T}$, where

$$c_x \;=\; \frac{1}{m}\int_B x\,dV, \qquad c_y \;=\; \frac{1}{m}\int_B y\,dV, \qquad c_z \;=\; \frac{1}{m}\int_B z\,dV.$$

Observe that this is just the continuous equivalent of computing the average x-, y- and z- coordinates of the body's mass. Note that the center of mass need not lie within the object (see Fig. 110). For example, the center of mass of a hoop is the center of the hoop.

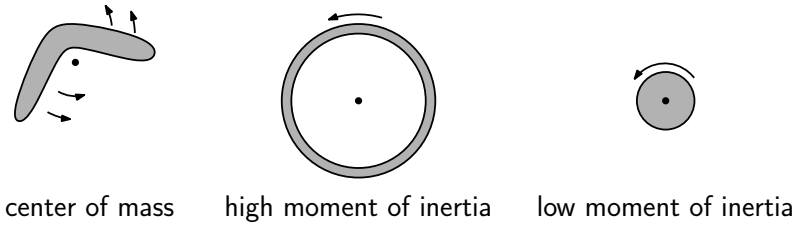


Fig. 110: Center of mass and moment of inertia.

Moment of Inertia: This is a scalar quantity, which corresponds to mass, but for torquing motions. In particular, in indicates how much an object resists a change in its angular velocity relative to a given rotation axis. Assuming for simplicity that the rotation is about the z-axis, we define the moment of inertia to be

$$I \;=\; \int_B (x^2 + y^2)dV.$$

For example, a hoop has a higher moment of inertia than a spherical lump of equal mass, since most of its mass is far from its center of mass (see Fig. 110). Given the same torque, the hoop will spin more slowly than the compact lump. (There is also a more complex physical quantity, called the *inertial tensor*, which encodes the moment of inertia for all possible rotation axes, but we will not discuss this.)

Physical State: Physical properties remain constant throughout an object's lifetime. There are also a number of physical properties that vary with time. These are refered to as the object's *physical state*. The physical state of a particle is described by two values:

Position: This is a point $p = (p_x, p_y, p_z)^\mathsf{T}$ that indicates the particle's location in space. (For rigid bodies, it is typically the location of the object's center of mass.)

Velocity: This is the derivative of position with respect to time, expressed as a vector $v = (v_x, v_y, v_z)^\mathsf{T}$.

Note that position and velocity are time dependent. When we want to emphasize position and velocity at a particular time t, we will write these as $p(t)$ and $v(t)$, respectively.

When dealing with rigid bodies, we also need to consider rotation. We add the following two additional elements:

Angular position: Angular position, which we will denote by q, indicates the amount of rotation relative to an initial (default) position. It can be expressed in a number of ways (e.g., Euler angles, rotation matrix, or unit quaternion). We will assume that it is represented as a unit quaternion $\mathbf{q} = (s, u)$. Recall that a rotation by angle θ about axis u can be expressed as the unit quaternion

$$\mathbf{q} = \left(\cos\frac{\theta}{2}, \left(\sin\frac{\theta}{2}\right) u\right), \qquad \text{where } \|u\| = 1.$$

Angular velocity: The angular velocity, denoted by ω, is typically represented as a 3-dimensional vector. The direction indicates the axis of rotation, and the length of the vector is the speed of rotation, given, say, in radians per second.

As above, rotation and angular velocity are functions of time, and so may be expressed as $q(t)$ and $\omega(t)$ to emphasize this dependence.

Physical Simulation and Kinematics: Simple physical simulations are performed by a process called *integration*. Intuitively, this involves updating the state of each physical object in your environment through a small time step. Collisions are first detected and contact and torque forces are computed. These forces result in accelerations that change the linear and angular velocities of objects. Finally, velocities change positions.

Assuming that the current state of a rigid body at time t is given by its position $p(t)$, its velocity $v(t)$, its angular position $q(t)$, and its angular velocity $\omega(t)$, the process of integration involves updating these quantities over a small time interval Δt. We can express this as

$$[p(t), v(t), q(t), \omega(t)] \rightarrow [p(t+\Delta t), v(t+\Delta t), q(t+\Delta t), \omega(t+\Delta t)]\,.$$

How this is done is the subject of *kinematics*, that is, the study of how quantities such as position, velocity, and acceleration interact. By definition $v(t)$ is the instantaneous change of position over time. This may also be expressed as dp/dt or $\dot{p}(t)$. Similarly, the angular velocity $\omega(t)$ is the instantaneous change of angular position $q(t)$ over time, which is often denoted by dq/dt or $\dot{q}(t)$. Assuming that the velocities $v(t)$ and $\omega(t)$ have been computed (from the known forces) we can then update the position and angular position as follows for a small time step Δt as:

$$\begin{aligned} p(t+\Delta t) &\approx p(t) + \dot{p}(t)\Delta t = p(t) + v(t)\Delta t \\ q(t+\Delta t) &\approx q(t) + \dot{q}(t)\Delta t = q(t) + \omega(t)\Delta t. \end{aligned}$$

Let us discuss rotation in a bit more detail. Let us assume that the angular position is expressed as a quaternion $\mathbf{q}(t)$, and the angular velocity is expressed as a vector $\omega(t) = (\omega_x(t), \omega_y(t), \omega_z(t))^{\mathsf{T}}$ (as described above). In order to update the angular position, we need to compute the derivative of the quaternion, assuming that the object is rotating with angular velocity $\omega(t)$. Consulting a reference on quaternions, we find that, in order to compute the desired quaternion derivative, we first compute the pure quaternion whose vector part is $\omega(t)$, that is, let $\mathbf{w}(t) = (0, \omega(t))$. The derivative, which we denote by $\dot{\mathbf{q}}(t)$, is

$$\dot{\mathbf{q}}(t) = \frac{1}{2}\mathbf{w}(t) \cdot \mathbf{q}(t),$$

where the $\cdot$ denotes quaternion multiplication.

Since rotation quaternions are required to be of unit length, we should normalize the result of applying the derivative in order to avoid it drifting away from unit length. Thus, we have the following rule for updating the angular position of a rotating object:

$$\mathbf{q}(t + \Delta t) \approx \mathit{normalize}\left(\mathbf{q}(t) + \dot{\mathbf{q}}(t)\Delta t\right).$$

Physics for the Programming Project: The programming project involves physics in a number of ways. Each physical event can be viewed as a force, which serves to modify an object's velocity (linear and/or angular).

User Impulses: Through keyboard input, the user can cause objects to move by applying various impulse forces. These forces have the effect of changing the current linear velocity by adding some 3-dimensional vector to the current velocity. Letting v denote the current object velocity and v_i denote the additional impulse velocity, we have:

$$v \leftarrow v + v_i.$$

Gravity and Friction: The force of gravity decreases the vertical component of the object's linear velocity. Friction and air resistance decreases the magnitude of the linear velocity, without changing its direction. Air resistance can also cause an object's angular velocity to decrease slowly over time. Let Δt denote the elapsed small time interval, let g be a constant related to gravity, and let δ_1 be a small positive constant related to friction and air resistance. We have

$$\begin{aligned} v_z &\leftarrow (1 - g \cdot \Delta t) v_z \qquad \text{(if the object is above the ground)} \\ v &\leftarrow (1 - \delta_1) \cdot \Delta t \cdot v \qquad \text{(friction or air resistance).} \end{aligned}$$

Rolling: Assuming that we have a spherical body that rolls along the ground. As it rolls (assuming no slippage), it also rotates. Suppose that the body is moving horizontally with linear velocity v, and assume that the "up" direction is the z-axis. Then, the axis of rotation passes through the center of mass and is directed to the object's left. (See Fig. 111.)

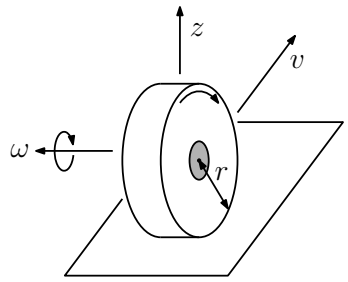


Fig. 111: Rolling rotation.

For a given linear velocity, the body's angular velocity varies inversely with its radius. (That is, a small radius wheel must spin faster to maintain the same speed as a large

radius wheel.) To achieve this, let r denote the body's radius. The angular velocity (in radians per second) is set to

$$\omega \leftarrow \frac{1}{r}(z \times v).$$

Note that, this is only applied when the object is rolling on the ground. If the body ceases to have contact with the ground, its angular velocity should remain constant (or possibly slow due to air resistance).

Collisions: When a collision occurs, we need to consider the impact of the body in terms of both translation and rotation. For simplicity, let us assume that all other objects in the scene are fixed, so that hitting an object is like hitting a wall. All of the force of the impact is directed back to the moving body. Assuming that the moving body is a sphere, whenever a collision occurs, we need to know the body's velocity vector v and the vector u from the center of the body to the point of contact (see Fig. 112(a)). Since the body is a sphere, the vector u is also the normal vector to the point of contact. There are two types of collision response to consider, translational and rotational.

Translational response: To determine the collision response, we decompose the vector v into two components, $v = v' + v''$, where v' is parallel to u and v'' is perpendicular to u (see Fig. 112(b)). As mentioned in an earlier lecture, this can be done with the following vector operations:

$$v' = \frac{(u \cdot v)}{(u \cdot u)}u \qquad \text{and} \qquad v'' = v - v'.$$

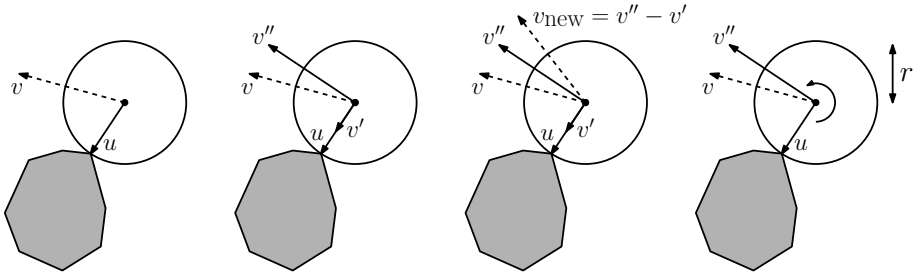


Fig. 112: Geometry of collisions.

To compute the effect of the collision on the object, the obstacle exerts an impulse to the object in the direction $-v'$. Thus, we update the linear velocity to

$$v \leftarrow v'' - \alpha \cdot v',$$

where $0 < \alpha \leq 1$ (called the *coefficient of restitution*) is a factor that takes into consideration various physical issues, such as the elasticity of the collision (see Fig. 112(c)).

Rotational response: Let u, v', and v'' be as defined above. The rotation induced by the collision is proportional to the length of v''. (To see this, observe that a purely head-on collision produces no rotation, whereas a glancing blow causes rotation.) As with rolling on the ground, the speed of the rotation is inversely proportional to the object's radius r. In order to determine the axis of rotation, we need a vector that is orthogonal to the rotation plane, which means that it must be orthogonal to both v'' and u. We take the rotation axis to be the cross product of v'' with the normalized vector u, which we scale by the inverse of the body radius (see Fig. 112(d)). Thus we have

$$\omega \leftarrow \frac{1}{r}\left(v'' \times \frac{u}{\|u\|}\right).$$

This assumes that the object's rotation is determined entirely by the collision. More generally, there may be slippage, and the rotation may be some linear combination of this rotation and its rotation prior to the collision.

Updating the Positions: Once the new velocities v and ω have been computed, we can update position and angular position. To do this, let $\mathbf{w}$ be the quaternion $(0, \omega)$, and define $\dot{\mathbf{q}} = \frac{1}{2}\mathbf{w} \cdot \mathbf{q}$ (using quaternion multiplication). Then we set:

$$p \leftarrow p + \Delta t \cdot v \qquad \mathbf{q} \leftarrow \mathit{normalize}\,(\mathbf{q} + \dot{\mathbf{q}}\Delta t).$$

Rendering: Finally, to render the object we use the following conceptual ordering. First draw the object, next apply the quaternion rotation. You can either use the rotation matrix (from the quaternion lecture) with glMultMatrix or you can extract the rotation angle and rotation axis from the quaternion and apply glRotatef. Finally apply the translation p using glTranslatef. As always, this order will be reversed and nested within in a matrix push-pop pair.

Supplemental Lecture 9: 3-D Modeling: Constructive Solid Geometry

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

Volume Based Representations: One of the most popular volume-based representations is *constructive solid geometry*, or *CSG* for short. It is widely used in manufacturing applications. 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.

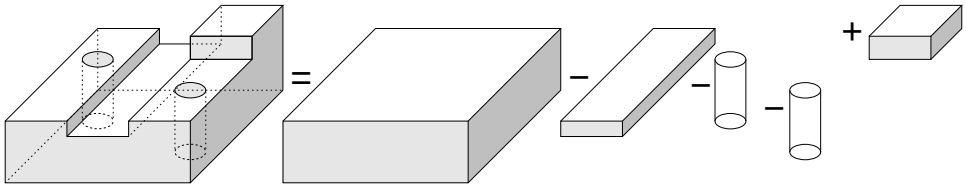


Fig. 113: Constructive Solid Geometry.

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 ($X \cup Y$), intersection ($X \cap Y$), difference ($X - Y$), etc. For example, the object above might be described with a tree of the following sort. (In the figure we have used + for union.)

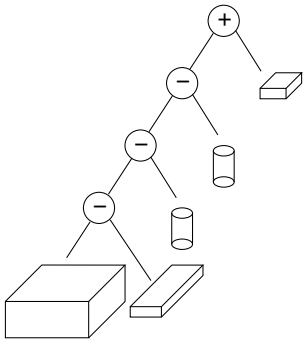


Fig. 114: CSG Tree.

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 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 isMember(Point P, CSGnode 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.

Membership Test for CSG Tree

```
bool isMember(Point P, CSGnode T) {
    if (T.isLeaf)
        return (membership test appropriate to T's type)
    else if (T.isUnion)
        return isMember(P, T.left ||  isMember(P, T.right)
    else if (T.isIntersect)
        return isMember(P, T.left &&  isMember(P, T.right)
    else if (T.isDifference)
        return isMember(P, T.left && !isMember(P, T.right)
}
```

Note that the semantics of operations "||" and "&&" 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.

CSG and Ray Tracing: CSG objects can be handled very naturally in ray tracing. Suppose that R is a ray, and T is a CSG tree. The intersection of the ray with any CSG object can be described as a (possibly empty) sorted set of intervals in the parameter space.

$$I = \langle [t_0, t_1], [t_2, t_3], \ldots \rangle.$$

(See Fig. 115.) This means that we intersect the object whenever $t_0 \leq t \leq t1$ and $t_2 \leq t \leq t_3$, and so on. At the leaf level, the set of intervals is either empty (if the ray misses the object) or is a single interval (if it hits). Now, we evaluate the CSG tree through a post-order traversal, working from the leaves up to the root. Suppose that we are at a union node v and we have the results from the left child I_L and the right child I_R.

We compute the union of these two sets of intervals. This is done by first sorting the endpoints of the intervals. With each interval endpoint we indicate whether this is an entry or exit. Then we traverse this sorted list. We maintain a depth counter, which is initialized to zero, and is incremented whenever we enter an interval and decremented when we exit an interval. Whenever this count transitions from 0 to 1, we output the endpoint as the start of a new interval in the union, and whenever the depth count transitions from 1 to 0, we output the resulting count as the endpoint of an interval. An example is shown in Fig. 115. (A similar procedure applies for intersection and difference. As an exercise, determine the depth count

transitions that mark the start and end of each interval.) The resulting set of sorted intervals is then associated with this node. When we arrive at the root, we select the smallest interval enpoint whose t-value is positive.

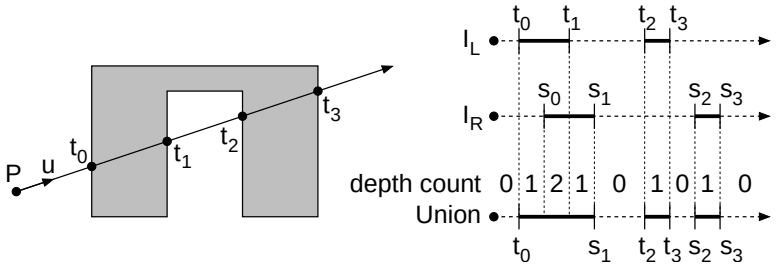


Fig. 115: Ray tracing in a CSG Tree.

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 Fig. 116. The result contains a "dangling" piece that has no width. That is, it is locally two-dimensional.

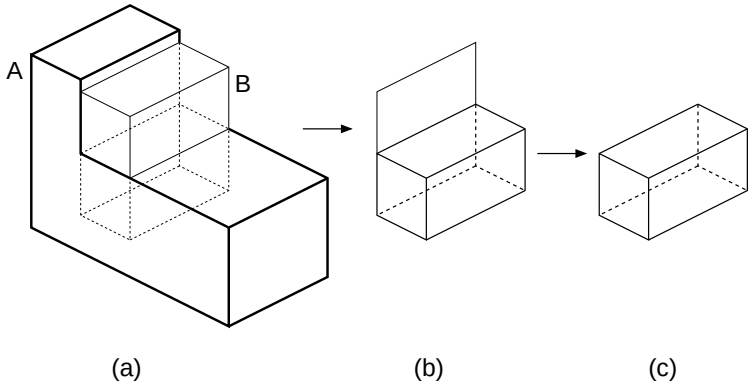


Fig. 116: (a) A and B, (b) $A \cap B$, (c) $A \cap^* B$.

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.

In order to define this formally, we must introduce some terms from topology. We can think of every (reasonable) shape as consisting of three disjoint parts, its *interior* of a shape A, denoted $\text{int}(A)$, its *exterior*, denoted $\text{ext}(A)$, and its *boundary*, denoted $\text{bnd}(A)$. Define the *closure* of any set to be the union of itself and its boundary, that is, $\text{closure}(A) = A \cup \text{bnd}(A)$.

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 $-$. Eliminating these dangling elements tends to complicate CSG algorithms, because it requires a bit more care in how geometric intersections are represented.

Supplemental Lecture 10: Fractals

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.

Iterated Function Systems and Attractors: 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,

$$z \to z^2,$$

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 system* (IFS).

In general, given any iterated function system in the complex plane, the *attractor set* is a subset of nonzero points that remain fixed under the mapping. This may also be called the *fixed-point set* of the system. Note that it is the set as a whole that is fixed, even though the individual points tend to move around. (See Fig. 117.)

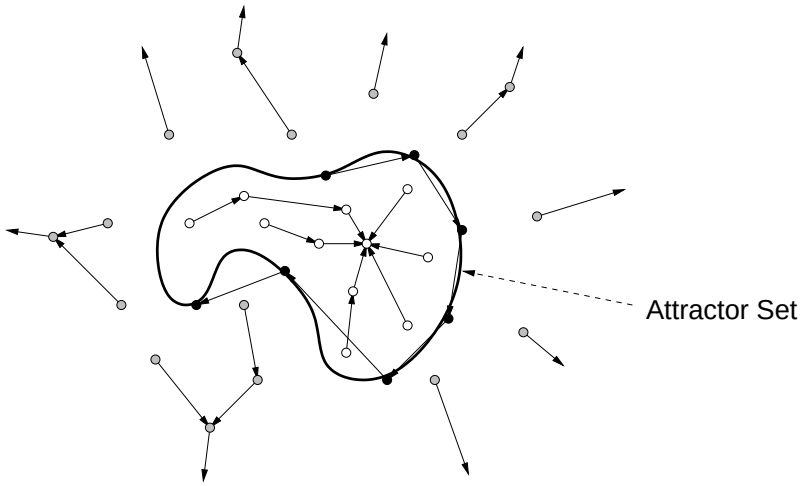


Fig. 117: Attractor set for an interated function system.

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

$$z \to z^2 + c$$

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 Fig. 118.

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.

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 set is called the *Mandelbrot set*. (See Fig. 119.)

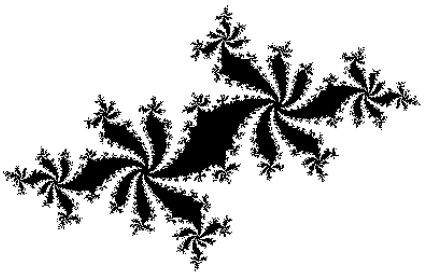


Fig. 118: A Julia Set.

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 $z \to z^2 + d$, 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.

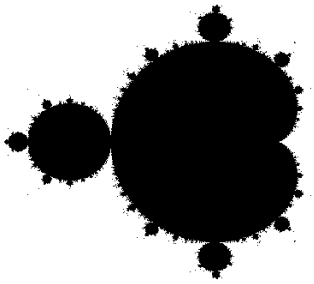


Fig. 119: The Mandelbrot Set.

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*. Suppose 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 is self-similar (at different scales) and 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. (See Fig. 120.)

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

$$\begin{aligned} D &= \lim_{\epsilon \to 0} \frac{\ln N(A, \epsilon)}{\ln(1/\epsilon)} \\ &= \lim_{k \to \infty} \frac{\ln N(A, (1/2^k))}{\ln(1/(1/2^k))} \\ &= \lim_{k \to \infty} \frac{\ln 3^k}{\ln 2^k} = \lim_{k \to \infty} \frac{k \ln 3}{k \ln 2} \\ &= \lim_{k \to \infty} \frac{\ln 3}{\ln 2} = \frac{\ln 3}{\ln 2} \approx 1.58496\ldots. \end{aligned}$$

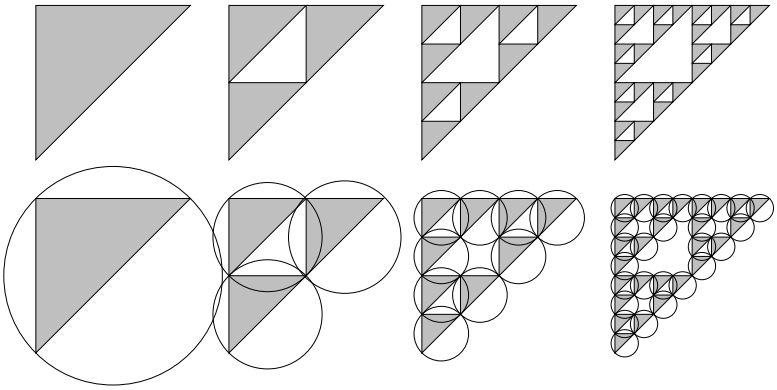


Fig. 120: The Sierpinski triangle.

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 Fig. 120. 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.

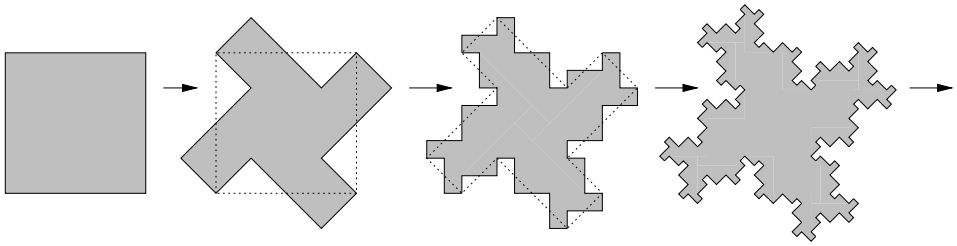


Fig. 121: An object with a fractal boundary.

Supplemental Lecture 11:
Ray Tracing: Triangle Intersection

Ray-Triangle Intersection: Suppose that we wish to intersect a ray with a polyhedral object. There are two standard approaches to this problem. The first works only for convex polyhedra. In this method, we represent a polyhedron as the intersection of a set of halfspaces. In this case, we can easily modify the 2-d line segment clipping algorithm presented in Lecture 9 to perform clipping against these halfspaces. We will leave this as an exercise. The other method involves representing the polyhedron by a set of polygonal faces, and intersecting the ray with these polygons. We will consider this approach here.

There are two tasks which are needed for ray-polygon intersection tests. The first is to extract the equation of the (infinite) plane that supports the polygon, and determine where the ray intersects this plane. The second step is to determine whether the intersection occurs within the bounds of the actual polygon. This can be done in a 2-step process. We will consider a slightly different method, which does this all in one step.

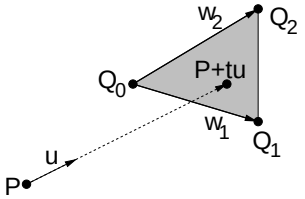


Fig. 122: Ray-triangle intersection.

Let us first consider how to extract the plane containing one of these polygons. In general, a plane in 3-space can be represented by a quadruple of coefficients (a, b, c, d), such that a point $P = (p_x, p_y, p_z)$ lies on the plane if and only if

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

Note that the quadruple (a, b, c, d) behaves much like a point represented in homogeneous coordinates, because any scalar multiple yields the same equation. Thus $(a/d, b/d, c/d, 1)$ would give the same equation (provided that $d \neq 0$).

Given any three (noncollinear) vertices of the polygon, we can compute these coefficients by solving a set of three linear equations. Such a system will be underdetermined (3 equations and 4 unknowns) but we can find a unique solution by adding a fourth normalizing equation, e.g. $a + b + c + d = 1$. We can also represent a plane by giving a normal vector $\vec{n}$ and a point on the plane Q. In this case (a, b, c) will just be the coordinates of $\vec{n}$ and we can derive d from the fact that

$$aq_x + bq_y + cq_z + d = 0.$$

To determine the value of t where the ray intersect the plane, we could plug the ray's parametric representation into this equation and simply solve for t. If the ray is represented by $P + t\vec{u}$, then we have the equation

$$a(p_x + tu_x) + b(p_y + tu_y) + c(p_z + tu_z) + d = 0.$$

Soving for t we have

$$t = -\frac{ap_x + bp_y + cp_z}{au_x + bu_y + cu_z}.$$

Note that the denominator is 0 if the ray is parallel to the plane. We may simply assume that the ray does not intersect the polygon in this case (ignoring the highly unlikely case where the ray hits the polygon along its edge). Once the intersection value t' is known, the actual point of intersection is just computed as $P + t\vec{u}$.

Let us consider the simplest case of a triangle. Let Q_0, Q_1, and Q_2 be the vertices of the triangle in 3-space. Any point Q' that lies on this triangle can be described by a convex combination of these points

$$Q' = \alpha_0 Q_0 + \alpha_1 Q_1 + \alpha_2 Q_2,$$

where $\alpha_i \geq 0$ and $\sum_i \alpha_i = 1$. From the fact that the α_i's sum to 1, we can set $\alpha_0 = 1 - \alpha_1 - \alpha_2$ and do a little algebra to get

$$Q' = Q_0 + \alpha_1(Q_1 - Q_0) + \alpha_2(Q_2 - Q_0),$$

where $\alpha_i \geq 0$ and $\alpha_1 + \alpha_2 \leq 1$. Let

$$\vec{w}_1 = Q_1 - Q_0, \qquad \vec{w}_2 = Q_2 - Q_0,$$

giving us the following

$$Q' = Q_0 + \alpha_1 \vec{w}_1 + \alpha_2 \vec{w}_2.$$

Recall that our ray is given by $P + t\vec{u}$ for $t > 0$. We want to know whether there is a point Q' of the above form that lies on this ray. To do this, we just substitute the parametric ray value for Q' yielding

$$\begin{aligned} P + t\vec{u} &= Q_0 + \alpha_1 \vec{w}_1 + \alpha_2 \vec{w}_2 \\ P - Q_0 &= -t\vec{u} + \alpha_1 \vec{w}_1 + \alpha_2 \vec{w}_2. \end{aligned}$$

Let $\vec{w}_P = P - Q_0$. This is an equation, where t, α_1 and α_2 are unknown (scalar) values, and the other values are all 3-element vectors. Hence this is a system of three equations with three unknowns. We can write this as

$$\left(\begin{array}{c|c|c} -\vec{u} & \vec{w}_1 & \vec{w}_2 \end{array} \right) \begin{pmatrix} t \\ \alpha_1 \\ \alpha_2 \end{pmatrix} = \begin{pmatrix} \vec{w}_P \end{pmatrix}.$$

To determine t, α_1 and α_2, we need only solve this system of equations. Let M denote the 3×3 matrix whose columns are $-\vec{u}$, $\vec{w}_1$ and $\vec{w}_2$. We can do this by computing the inverse matrix M^{-1} and then we have

$$\begin{pmatrix} t \\ \alpha_1 \\ \alpha_2 \end{pmatrix} = M^{-1} \begin{pmatrix} \vec{w}_P \end{pmatrix}.$$

There are a number of things that can happen at this point. First, it may be that the matrix is singular (i.e., its columns are not linearly independent) and no inverse exists. This happens

if $\vec{t}$ is parallel to the plane containing the triangle. In this case we will report that there is no intersection. Otherwise, we check the values of α_1 and α_2. If either is negative then there is no intersection and if $\alpha_1 + \alpha_2 > 1$ then there is no intersection.

Normal Vector: In addition to computing the intersection of the ray with the object, it is also desirable to compute the normal vector at the point of intersection. In the case of the triangle, this can be done by computing the cross product

$$\vec{n} = \mathit{normalize}((Q_1 - Q_0) \times (Q_2 - Q_0)) = \mathit{normalize}(\vec{w}_1 \times \vec{w}_2).$$

But which direction should we take for the normal, $\vec{n}$ or $-\vec{n}$? This depends on which side of the triangle the ray arrives. The normal should be directed opposite to the directional ray of the vector. Thus, if $\vec{n} \cdot \vec{u} > 0$, then negate $\vec{n}$.

Supplemental Lecture 12: Ray Tracing Bezier Surfaces

Issues in Ray Tracing: Today we consider a number of miscellaneous issues in the ray tracing process.

Ray and Bézier Surface Intersection: Let us consider a more complex but more realistic ray intersection problem, namely that of intersecting a ray with a Bézier surface. One possible approach would be to derive an implicit representation of infinite algebraic surface on which the Bézier patch resides, and then determine whether the ray hits the portion of this infinite surface corresponding to the patch. This leads to a very complex algebraic task.

A simpler approach is based on using circle-ray and triangle-ray intersection tests (which we have already discussed) and the deCasteljau procedure for subdividing Bézier surfaces. The idea is to construct a simple enclosing shape for the curve, which we will use as a *filter*, to rule out clear misses. Let us describe the process for a Bézier curve, and we will leave the generalization to surfaces as an exercise.

What enclosing shape shall we use? We could use the convex hull of the control points. (Recall the convex hull property, which states that a Bézier curve or surface is contained within the convex hull of its control points.) However, computing convex hulls, especially in 3-space, is a tricky computation.

We will instead apply a simpler test, by finding an enclosing circle for the curve. We do this by first computing a center point C for the curve. This can be done, for example, by computing the centroid of the control points. (That is, the average of the all the point coordinates.) Alternatively, we could take the midpoint between the first and last control points. Given the center point C, we then compute the distance from each control point and C. Let $d_{\max}$ denote the largest such distance. The circle with center C and radius $d_{\max}$ encloses all the control points, and hence it encloses the convex hull of the control points, and hence it encloses the entire curve. We test the ray for intersection with this circle. An example is shown in Fig. 123.

If it does not hit the circle, then we may safely say that it does not hit the Bézier curve. If the ray does hit the circle, it still may miss the curve. Here we apply the deCasteljau algorithm

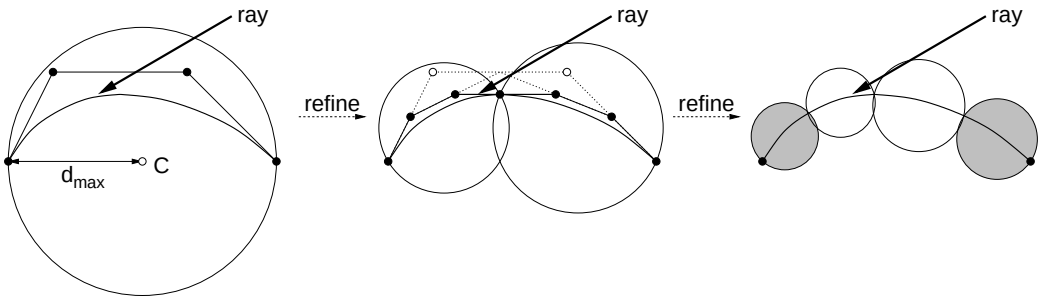


Fig. 123: Ray tracing Bézier curves through filtering and subdivision.

to subdivide the Bezier curve into two Bezier subcurves. Then we apply the ray intersection algorithm recursively to the two subcurves. (Using the same circle filter.) If both return misses, then we miss. If either or both returns a hit, then we take the closer of the two hits. We need some way to keep this recursive procedure from looping infinitely. To do so, we need some sort of stopping criterion. Here are a few possibilities:

Fixed level decomposition: Fix an integer k, and decompose the curve to a depth of k levels (resulting in 2^k) subcurves in all. This is certainly simple, but not a very efficient approach. It does not consider the shape of the curve or its distance from the viewer.

Decompose until flat: For each subcurve, we can compute some function that measures how *flat*, that is, close to linear, the curve is. For example, this might be done by considering the ratio of the length of the line segment between the first and last control points and distance of the furthest control point from this line. At this point we reduce the ray intersection to line segment intersection problem.

Decompose to pixel width: We continue to subdivide the curve until each subcurve, when projected back to the viewing window, overlaps a region of less than one pixel. Clearly it is unnecessary to continue to subdivide such a curve. This solves the crack problem (since cracks are smaller than a pixel) but may produce an unnecessarily high subdivision for nearly flat curves. Also notice that this the notion of back projection is easy to implement for rays emanating from the eye, but this is much harder to determine for reflection or refracted rays.