Lecture Notes

Fundamentals of Computer Graphics

Prof. Michael Langer
School of Computer Science
McGill University

NOTE: These lecture notes are dynamic. The initial version of the document contained *drafts* only. The notes will be updated every lecture. Material will be added/deleted as need be, mistakes corrected, etc. Do not print out or annotate this PDF, expecting it to remain stable.

Lecture 1

Computer graphics, broadly defined, is a set of methods for using computers to create and manipulate images. There are many applications of computer graphics including entertainment (games, cinema, tv commercials and other advertising), design (architectural, manufacturing, ...), visualization (medical, scientific, information), simulation (training), and much more.

This course will look at methods for creating images that are defined on a rectangular pixel grid. The word *pixel* stands for "picture element". The coordinates of the grid are integers. For example, typical high definition images are 1920×1080. The resolution of a typical projector such as this one is 800×600 (SVGA).

The color of a pixel is defined by three numbers called RGB. These are the intensities of the red, green, and blue components. Typical images have 8 bits for each RGB value and hence 24 bits of intensity information per pixel i.e. values from 0 to 255. Here are some examples of colors:

(0,0,0)	black
(255,255,255)	white
(255,0,0)	red
(0, 255, 0)	green
(0, 0, 255)	blue
(255,255,0)	yellow (red + green)
(255,0,255)	magenta (red + blue)
(0,255,255)	cyan (green + blue)

RGB space defines a cube: $\{0, \ldots, 255\} \times \{0, \ldots, 255\} \times \{0, \ldots, 255\}$. The above list of colors define the corners of the cube. There are many other colors ($256 \times 256 \times 256 = 2^{24}$) with exotic names. You can amuse yourselves by having a look here:
`http://www.w3schools.com/tags/ref_colornames.asp`

What this course is

This course is an introduction to the fundamentals of the field of computer graphics. These fundamentals have not changed much in the past few decades. The course I am offering you is similar to what Prof. Paul Kry[1] offers when he teaches this course and to what you'll find in other universities as well, with some variations that reflect the tastes of the individual instructors.

What this course is not

Although the fundamentals of computer graphics have not changed much in the past few decades, many aspects of computer graphics *have* changed. Graphics cards (GPUs) in particular have become very powerful and many software tools have become available for taking full advantage of this power. To work as a software developer in the computer graphics industry today, you would need to learn

[1] Paul has taught the course the past six years. He is away on sabbatical this year but will return to teach the course again in Fall 2015.

at least the basics of these tools. I want to warn you up front: while these modern software tools are built on the fundamentals that we will cover in the course, this course itself will not teach you these software tools: this is not a course about modern computer graphics programming and software development.

A bit of history

To appreciate the above point, it may be helpful to consider a few key developments of the past 20 years in computer graphics:

Up to the early 1990's, it was relatively cumbersome to write graphics software. The standards for graphics software were limited and typically one would code up an algorithm from scratch. Although many companies had developed specialized graphics hardware (GPU's – graphical processing units), this hardware was difficult to use since it often required assembly language programming. New algorithms were typically implemented to run on the CPU, rather than GPU, and as a result they tended to be slow and impractical for typical users.

One of the big graphics companies at that time (SGI) built and sold workstations that were specialized for graphics. SGI developed an API called GL which had many libraries of common graphics routines. In 1992 SGI released a standard API called *OpenGL* which other hardware vendors could use as well.

[ASIDE: Microsoft followed with the DirectX API shortly thereafter. `http://en.wikipedia.org/wiki/DirectX`. DirectX is used in the "X Box". DirectX competes with OpenGL and has remained popular among many users especially those who work with Microsoft software. I will only discuss OpenGL in the course but it is good to be aware of DirectX as well.]

OpenGL 1.0 is said to have a *fixed function pipeline*. "Fixed function" means that the library offers a limited fixed set of functions. You cannot go beyond that library. You cannot program the graphics hardware to do anything other than what OpenGL offers.

Graphics card technology improved steadily from the mid 1990's and beyond the emergence of industry leaders NVIDIA, ATI (now part of AMD) and Intel. However, to program a graphics card (GPU) to make the most use of its enormous processing potential, one still had to write code that was at the assembly language level. This was relatively difficult and not much fun for most people.

In 2004, OpenGL 2.0 was released which allowed access to the graphics card using a high level C-like language called the OpenGL Shading Language (GLSL). Other "shading languages" have been developed since then, and more recent versions of OpenGL have been released as well. I'll say more later in the course about what such shading languages are for.

The most recent trend is WebGL (released in 2011) which is an API specific for graphics in web browsers. Google maps uses WebGL for example. There is a growing trend to introduce WebGL in "intro to graphics" courses.

Course Outline

I next went of the official Course Outline. Please read it.
`http://www.cim.mcgill.ca/~langer/557/CourseOutline.pdf`.

A bit about me

It may be helpful for you to know a bit about me, what I'm interested in, and what is my relationship to the field of computer graphics. Here is a brief overview. At the end of this course, I'll say more about my research interests. I also offer a course that is more directly related to my interests: Computational Perception (COMP 546/598). I hope to offer it in 2015-2016.

I did my undergrad here in the early 1980s, majoring in Math and minoring in Computer Science. I did a MSc in Computer Science at University of Toronto. In my MSc thesis, I looked at how to efficiently code natural images using statistics. This topic is more in the field of Data Compression than Vision, but I was motivated by visual coding and the work led me further into vision science.

After my M.Sc., I returned to McGill in 1989 to start a PhD. There were no course requirements for a Ph.D. here in those days and my supervisor was in Electrical Engineering so that's the department I was as well (but I know very little about EE). My PhD was in computer vision, specifically, how to compute the shape of surfaces from smooth shading and shadows.

I received my PhD in 1994 then spent six years as a post-doctoral research in the late 1990s. It was only then that I started learning about graphics. I used graphics as part of my vision research, namely it was a way to automatially generate images that I could use for testing computer vision algorithms. I also began doing research in human vision (visual perception) and I used graphics to make images for my psychology experiments.

I returned to McGill in 2000 as a professor. I've been doing a mix of computer vision and human vision since then. I've taught this course COMP 557 four times over the years.

My research combines computer vision, human vision, and computer graphics. Most of the problems I work on addressed 3D vision, including depth perception and shape perception. I have worked on many types of vision problems including shading, motion, stereopsis, and defocus blur. I'll mention some of the problems I've worked on later in the course when the topics are more relevant.

Ok, enough about me, let's get started...

Review of some basic linear algebra: dot and cross products

We'll use of many basic tools from linear algebra so let's review some basics now.

- The **length of 3D vector u** is:

$$|\mathbf{u}| \equiv \sqrt{u_x^2 + u_y^2 + u_z^2}$$

- The **dot product** (or "scalar product") of two 3D vectors **u** and **v** is defined to be:

$$\mathbf{u} \cdot \mathbf{v} \equiv u_x v_x + u_y v_y + u_z v_z \ .$$

  A more geometric way to define the dot product is to let θ be the angle between these two vectors, namely within the 2D plane spanned by the vectors.[2] Then,

$$\mathbf{u} \cdot \mathbf{v} \equiv |\mathbf{u}| \ |\mathbf{v}| \cos(\theta).$$

[2] If the two vectors are in the same direction, then they span a line, not a plane.

It may not be obvious why these two definitions are equivalent, *in general*. The way I think of it is this. Take the case that the vectors already lie in a plane, say the xy plane. Suppose $\mathbf{u} = (1,0,0)$ and $\mathbf{v} = (\cos\theta, \sin\theta, 0)$. In this case, it is easy to see that the two definitions of dot product given above are equal. As an exercise: why does this equivalence holds in general?

It is important to realize that the dot product is just a matrix multiplication, namely $\mathbf{u}^T\mathbf{v}$, where $\mathbf{u}$ and $\mathbf{v}$ are 3×1 vectors. Thus for a fixed $\mathbf{u}$, the dot product $\mathbf{u} \cdot \mathbf{v}$ is a linear transformation of $\mathbf{v}$. Similarly for a fixed $\mathbf{v}$, the dot product $\mathbf{u} \cdot \mathbf{v}$ is a linear transformation of $\mathbf{u}$.

Another way to think about the dot product is to define $\hat{\mathbf{x}} \cdot \hat{\mathbf{x}} = \hat{\mathbf{y}} \cdot \hat{\mathbf{y}} = \hat{\mathbf{z}} \cdot \hat{\mathbf{z}} = 1$, and $\hat{\mathbf{x}} \cdot \hat{\mathbf{y}} = 0$, $\hat{\mathbf{x}} \cdot \hat{\mathbf{z}} = 0$, $\hat{\mathbf{y}} \cdot \hat{\mathbf{z}} = 0$ and to define the dot product to be linear, then we have

$$\mathbf{u} \cdot \mathbf{v} = (u_x\hat{\mathbf{x}} + u_y\hat{\mathbf{y}} + u_z\hat{\mathbf{z}}) \cdot (u_x\hat{\mathbf{x}} + u_y\hat{\mathbf{y}} + v_z\hat{\mathbf{z}}) \ = \ u_x v_x + u_y v_y + u_z v_z$$

where we expanded the dot product into 9 terms and used the fact that $\hat{\mathbf{x}} \cdot \hat{\mathbf{x}} = 1, \hat{\mathbf{x}} \cdot \hat{\mathbf{y}} = 0, ...$

- The **cross product** of two 3D vectors $\mathbf{u}$ and $\mathbf{v}$ is written $\mathbf{u} \times \mathbf{v}$. The cross product is a 3D vector that is perpendicular to both $\mathbf{u}$ and $\mathbf{v}$, and hence to the plane spanned by $\mathbf{u}$ and $\mathbf{v}$ in the case that these two vectors are not parallel.

  Given two vectors $\mathbf{u}$ and $\mathbf{v}$, the cross product $\mathbf{u} \times \mathbf{v}$ is usually defined by taking the 2D determinants of "co-factor" matrices:

$$\mathbf{u} \times \mathbf{v} = \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix} \times \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} \equiv \begin{bmatrix} u_y v_z - u_z v_y \\ u_z v_x - u_x v_z \\ u_x v_y - u_y v_x \end{bmatrix}$$

  Sometimes in linear algebra courses, this definition seems to come out of nowhere. I prefer to think of the above, not as a "definition", but rather as a result of more basic definition, as follows. We *define* the cross product on the unit vectors $\hat{\mathbf{x}}, \hat{\mathbf{y}}, \hat{\mathbf{z}}$, namely

$$\hat{\mathbf{z}} = \hat{\mathbf{x}} \times \hat{\mathbf{y}}$$

$$\hat{\mathbf{x}} = \hat{\mathbf{y}} \times \hat{\mathbf{z}}$$

$$\hat{\mathbf{y}} = \hat{\mathbf{z}} \times \hat{\mathbf{x}}$$

  and we obtain six other equations by assuming for all $\mathbf{u}$ and $\mathbf{v}$,

$$\mathbf{u} \times \mathbf{v} = -\mathbf{v} \times \mathbf{u}$$

$$\mathbf{u} \times \mathbf{u} = \mathbf{0} \ ,$$

  for example, $\hat{\mathbf{x}} \times \hat{\mathbf{x}} = 0$ and $\hat{\mathbf{y}} \times \mathbf{x} = -\hat{\mathbf{z}}$. We also assume that, for any $\mathbf{u}$, the cross product $\mathbf{u} \times \mathbf{v}$ is a linear transformation on $\mathbf{v}$. Writing

$$\mathbf{u} = u_x\hat{\mathbf{x}} + u_y\hat{\mathbf{y}} + u_z\hat{\mathbf{z}}$$

  and similarly for $\mathbf{v}$, we use the linearity to expand $\mathbf{u} \times \mathbf{v}$ into nine terms ($9 = 3^2$), only six of which are non-zero. If we group these terms, we get the mysterious "determinants of co-factor bla bla" formulas above.

Finally, verify from the above definition that, for any fixed 3D vector $\mathbf{u}$, there is a 3×3 matrix which transforms any other 3D vector $\mathbf{v}$ to the cross product $\mathbf{u} \times \mathbf{v}$, namely :

$$\mathbf{u} \times \mathbf{v} = \begin{bmatrix} 0 & -u_z & u_y \\ u_z & 0 & -u_x \\ -u_y & u_x & 0 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$$

Notice that this matrix is not invertible. It has rank 2, not 3. In particular, when $\mathbf{u} = \mathbf{v}$, we get $\mathbf{u} \times \mathbf{u} = \mathbf{0}$.

Computing the normal of a triangle, and its plane

A key application for the cross product is to compute a vector normal to a planar surface such as a triangle or more generally a polygon. Let's just deal with a triangle for now. Let the vertices be $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2$ and assume we are looking at the triangle from a direction so that these vertices go counter-clockwise. We will define the surface normal (perpendicular) such that it points into the half space containing us (the viewer).

Let $\mathbf{e}_0$ denote the vector $\mathbf{p}_1 - \mathbf{p}_0$, i.e. the vector from $\mathbf{p}_0$ to $\mathbf{p}_1$, and let $\mathbf{e}_1$ denote the vector $\mathbf{p}_2 - \mathbf{p}_1$. Then the *outward* surface normal (pointing toward the viewer) is defined

$$\mathbf{N} = \mathbf{e}_0 \times \mathbf{e}_1.$$

The vertex point $\mathbf{p}_0 = (p_{0,x}, p_{0,y}, p_{0,z})$ is in the plane containing the triangle. So the equation of the plane passing through this point can be parameterized by:

$$\mathbf{N} \cdot (x - p_{0,x}, y - p_{0,y}, z - p_{0,z}) = 0,$$

This equation just says that, for any point (x, y, z) on the plane, the vector from this point to $\mathbf{p}_0$ is perpendicular to the surface normal.

Lecture 2

When defining a scene that we want to draw, we will often need to perform transformations on the objects we want to draw by rotating, scaling, and translation them. To do so, we will perform linear transformations on the points that define the object. Let's begin by discussing rotation. We begin with the 2D case.

2D Rotations

If we wish to rotate by θ degrees *counterclockwise*, then we perform a matrix multiplication

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \mathbf{R} \begin{bmatrix} x \\ y \end{bmatrix}$$

where

$$\mathbf{R} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}.$$

The rotation is counterclockwise when $\theta > 0$. The way to write down this transformation in case you forget it is to note that the rotation maps (1,0) to $(\cos\theta, \sin\theta)$ and it maps (0, 1) to $(-\sin\theta, \cos\theta)$. These two observations determine the first and second columns of $\mathbf{R}$. See the slides if need be.

There are two ways to think about what's happening. First, applying the rotation by multiplying by $\mathbf{R}$ moves point (x, y) in a fixed 2D coordinate system. Second, multiplying by $\mathbf{R}$ projects the vector (x, y) onto vectors that are defined by the rows of $\mathbf{R}$. The first interpretation says that we are keeping the coordinate system but moving (rotating) the points within this fixed coordinate system. The second says we are defining a new coordinate system to represent our space $\Re^2$. Verify that this new coordinate system itself is rotated *clockwise* by θ relative to the original one. This distinction will turn out to be useful as we'll see in the next few lectures when we discuss world versus camera coordinate systems.

3D Rotations

For 3D vectors, it is less clear how to define a clockwise versus counterclockwise rotation. It depends on how we define these terms in 3D and whether the xyz coordinate system is "right handed" or "left handed". The standard coordinate *xyz* system in computer graphics is *right handed*. The axes correspond to thumb ($\hat{\mathbf{x}}$), index finger ($\hat{\mathbf{y}}$), and middle finger ($\hat{\mathbf{z}}$). We will occasionally have to switch from right to left handed coordinates, but this is something we will try to avoid.

As in the 2D case, one way to think of a 3D rotation is to move a set of points (x, y, z) in the world relative to some fixed coordinate frame. For example, you might rotate your head or rotate your chair (and your body, if its sitting on the chair). In this interpretation, we define a 3D *axis of rotation* and an *angle of rotation*. Whenever we want to express a rotation, we need a coordinate system and we rotate *about the origin* of that coordinate system, so that the origin doesn't move. For the example of rotating your head, the origin would be some point in your neck.

We can rotate about many different axes. For example, to rotate about the x axis, we perform:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

This transformation leaves the x coordinate of each 3D point fixed, but changes the y and z coordinates, at least for points that are not on the axis i.e. $(y, z) \neq (0, 0)$. Notice that the xz rotation is essentially a 2D rotation.

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

In a right handed coordinate system, the above rotation $\mathbf{R}_x$ is clockwise if we are looking in the $\mathbf{x}$ direction, and is counterclockwise if we are looking in the $-\mathbf{x}$ direction.

If we rotate about the z axis (leaving z fixed), we use

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

This rotates counter clockwise in the xy plane, if we are looking in the $-\mathbf{z}$ direction.

If we rotate about the y axis, this leaves the values of y fixed. We write this as

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

Verify for yourself that this is counter-clockwise if we are looking in the -**y** direction for a right handed coordinate system. Curiously, the "-" sign is at the bottom left of the matrix rather than top right. Verify that this is correct, but don't stress about remembering it.

In general, by a *3D rotation matrix* we will just mean a real 3×3 invertible matrix $\mathbf{R}$ such that

$$\mathbf{R}^T = \mathbf{R}^{-1}$$

i.e.

$$\mathbf{R}^T\mathbf{R} = \mathbf{R}\mathbf{R}^T = \mathbf{I},$$

where $\mathbf{I}$ is the identity matrix. Note the rows (and columns) of $\mathbf{R}$ are orthogonal to each other and of unit length. Recall such matrices are called *orthonormal*. We also require that the determinant of $\mathbf{R}$ is 1 (see below). In linear algebra, rotation matrices are special cases of *unitary* matrices.

Rotation matrices preserve the inner product between two vectors. To see this, let $\mathbf{p}_1$ and $\mathbf{p}_2$ be two 3D vectors, possibly the same. Then

$$(\mathbf{R}\mathbf{p}_1) \cdot (\mathbf{R}\mathbf{p}_2) = \mathbf{p}_1^T\mathbf{R}^T\mathbf{R}\mathbf{p}_2 = \mathbf{p}_1^T\mathbf{p}_2 = \mathbf{p}_1 \cdot \mathbf{p}_2.$$

It follows immediately that rotations preserve the length of vectors (consider $\mathbf{p}_1 = \mathbf{p}_2$). Intuitively, rotations also preserve the angle between two vectors. A few pages from now, you should be able prove why.

Why did we require that the determinant of $\mathbf{R}$ is 1 ? An example of an orthogonal matrix that does *not* have this property is:

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

since its determinant is -1, rather than 1. This matrix performs a mirror reflection of the scene about the $x = 0$ plane. The length of vectors is preserved, as is the angle between any two vectors. But the matrix is not a rotation because of the failure of the determinant property. More intuitively, this matrix transforms objects into their mirror reflections; it turns left hands into right hands.

Another orthonormal matrix that fails the "det $\mathbf{R} = 1$" condition is

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

which swaps the x and y variables. This matrix is also a mirror reflection, namely reflects about the plane $x = y$. Swapping yz or xz also fails, of course.

Example 1

Suppose we wish to find a rotation matrix that maps $\hat{\mathbf{z}}$ to some given vector $\mathbf{p}$ which has unit length. i.e.

$$\mathbf{p} \;=\; \mathbf{R} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}.$$

We suppose that $\mathbf{p} \neq \hat{\mathbf{z}}$, since the problem would be trivial in this case.

Here is the solution. By inspection, the 3rd column of $\mathbf{R}$ has to be $\mathbf{p}$ itself. Moreover, in order for $\mathbf{R}$ to be a rotation matrix, it is necessary that the first two columns of $\mathbf{R}$ are perpendicular to the vector $\mathbf{p}$, that is, to the third column. What might these first two columns of $\mathbf{R}$ be ?

Since we have assumed $\mathbf{p} \neq \hat{\mathbf{z}}$, we consider the vector

$$\mathbf{p}' = \frac{\mathbf{p} \times \hat{\mathbf{z}}}{\| \mathbf{p} \times \hat{\mathbf{z}} \|}$$

and note that $\mathbf{p}'$ is perpendicular to $\mathbf{p}$. Hence we can use it as one of the columns of our matrix $\mathbf{R}$. (More generally, note $\mathbf{p}'$ is perpendicular to the plane spanned by $\mathbf{p}$ and $\mathbf{z}$.)

We now need a third vector, which is perpendicular to both $\mathbf{p}$ and **p'**. We chose $\mathbf{p}' \times \mathbf{p}$, noting that it is of unit length, since **p'** is already perpendicular to $\mathbf{p}$ and both $\mathbf{p}$ and **p'** are of unit length.

We also need to satisfy the determinant equals 1 condition i.e. want to build a matrix $\mathbf{R}$ which preserves handedness. For example, since $\hat{\mathbf{x}} = \hat{\mathbf{y}} \times \hat{\mathbf{z}}$, we want that $\mathbf{R}\hat{\mathbf{x}} = \mathbf{R}\hat{\mathbf{y}} \times \mathbf{R}\hat{\mathbf{z}}$. How do we do it? I didn't go over this in class, but I'll do it here so you can see. Recall $\mathbf{p} = \mathbf{R}\hat{\mathbf{z}}$, so $\mathbf{p}$ is in the third column of $\mathbf{R}$. By inspection, if were to put $\mathbf{p}'$ in the second column of $\mathbf{R}$, we would have

$$\mathbf{R}\hat{\mathbf{y}} = \mathbf{p}'$$

and similarly if we were to put $\mathbf{p}' \times \mathbf{p}$ in the first column of $\mathbf{R}$, we would have

$$\mathbf{R}\hat{\mathbf{x}} = \mathbf{p}' \times \mathbf{p}.$$

So

$$\mathbf{R}\hat{\mathbf{x}} = \mathbf{p}' \times \mathbf{p} = \mathbf{R}\hat{\mathbf{y}} \times \mathbf{R}\hat{\mathbf{z}}$$

which was what we wanted. i.e. to preserve the handedness of the axes. Thus, we could indeed let the three columns of $\mathbf{R}$ be defined:

$$\mathbf{R} = [\mathbf{p}' \times \mathbf{p}\ ,\ \mathbf{p}'\ ,\ \mathbf{p}].$$

Notice that there are many possible solutions to the problem we posed. For example, we can perform *any* rotation $\mathbf{R}_z$ about the z axis prior to applying a rotation $\mathbf{R}$ and we would still have a solution. That is, if $\mathbf{R}$ is a solution then so is $\mathbf{R}\ \mathbf{R}_z(\theta)$ for any θ since any such matrix would still take the z axis to the proscribed vector $\mathbf{p}$.

Example 2

Suppose we wish to have a rotation matrix that rotates a given unit length vector $\mathbf{p}$ to the z axis, i.e.

$$\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \mathbf{R}\ \mathbf{p}.$$

It should be obvious that this can be done by taking the transpose of the solution of Example 1, since the problem just asks for the inverse of the problem of Example 1 and the transpose of a rotation matrix is its inverse.

Axis of rotation

You may be finding it puzzling to use the term "rotation" for these matrices. Think of a spinning body such as the Earth or a baseball. When we say it is rotating, we usually mean that there is an axis about which the object spins. If you take some finite time interval, the body goes continuously from one orientation to another and the axis plays a special role in the change in orientation. It is this continous change in orientation that we usually mean by term "rotation". So, what do we mean by "rotation" for a discrete matrix $\mathbf{R}$, in particular, what do we mean by the term "axis of rotation " ?

From linear algebra, you can verify that a 3D rotation matrix (as defined earlier) must have at least one real eigenvalue. Because rotation matrices preserve vector lengths, this eigenvalue must be 1. The eigenvector $\mathbf{p}$ that corresponds to this eigenvalue satisfies

$$\mathbf{p} = \mathbf{R}\mathbf{p}\ .$$

Note that this vector doesn't change under the rotation. This vector is what we mean by the "axis of rotation" defined by the rotation matrix $\mathbf{R}$.

Given a rotation matrix $\mathbf{R}$, you could compute the axis of rotation by finding the eigenvector $\mathbf{p}$ whose eigenvalue is of length 1. This tells you the axis of rotation but it doesn't tell you the amount of rotation. For example, the family of matrices $\mathbf{R}_z(\theta)$ all have the same axis of rotation $\hat{\mathbf{z}}$ and shared eigenvalue 1. But their other eigenvalues depend on θ. [ASIDE: Their other eigenvalues are complex numbers $\cos(\theta) \pm i \sin(\theta)$. See Exercises 1.]

Example 3

Let's now come at this last problem from the other direction. Suppose we wish to define a rotation matrix that rotates by an angle θ about a given axis $\mathbf{p}$ which is a unit length vector. Obviously if $\mathbf{p}$ is one of the canonical axes ($\hat{\mathbf{x}}, \hat{\mathbf{y}}$, or $\hat{\mathbf{z}}$), then it is easy. So, let's assume that $\mathbf{p}$ is not one of the canonical axes.

One easy way to come up with such a matrix is to first rotate $\mathbf{p}$ to one of the canonical axes (Example 2), then perform the rotation by θ about this canonical axis, and then rotate the canonical axis back to $\mathbf{p}$ (Example 1). Symbolically,

$$\mathbf{R}_{p\leftarrow z}\ \mathbf{R}_z(\theta)\ \mathbf{R}_{z\leftarrow p}\ =\ \mathbf{R}_p(\theta).$$

Example 4

Let's look at a slightly trickier version of this last problem. Now we only allow for rotations about axes $\hat{\mathbf{x}}, \hat{\mathbf{y}}, \hat{\mathbf{z}}$. Here is one solution which I waved my hands through in class. First, rotate about the x axis by some angle θ_1 to bring $\mathbf{p}$ into the xy plane (z=0). Then rotate about the z axis by some angle θ_2 to bring the vector $\mathbf{R}_x(\theta_1)\mathbf{p}$ to the y axis. Now we have $\mathbf{R}_z(\theta_2)\mathbf{R}_x(\theta_1)\mathbf{p}$ and we do the desired rotation by θ around the y axis. We then undo the first two rotations to bring the y axis back to $\mathbf{p}$. This gives:

$$\mathbf{R}_x^T(\theta_1)\mathbf{R}_z^T(\theta_2)\mathbf{R}_y(\theta)\mathbf{R}_z(\theta_2)\mathbf{R}_x(\theta_1)$$

and we're done!

ASIDE: Representations of rotations in computer graphics

There are three very common methods for representing rotations in computer graphics. Here they are, from simple to complicated:

1. The rotation matrix is specified by a rotation axis $\mathbf{p}$ and by an angle of rotation θ. There is no need for $\mathbf{p}$ to be a unit vector (since it can be normalized i.e. the user doesn't need to do that). This is known as the *axis-angle* representation. OpenGL uses something like this as we'll see next week, namely `glRotate`.

2. There exists a solution for Example 4 that uses only three canonical rotations, say

$$\mathbf{R}_x(\theta_x)\ \mathbf{R}_y(\theta_y)\ \mathbf{R}_z(\theta_z).$$

The rotations are called "*Euler angles*". For simplicity, I am omitting how you find these three angles, given say an axis-angle representation of $\mathbf{R}$.

One problem with Euler angles arises when they are used in animation. If you want to rotate an object from one orientation to another over a sequence of image frames, you often specify an orientation in one frame and then another orientation n frames later. These are called *key frames.* You then ask your application (say *Blender* `http://www.blender.org/`) to fill in all the frames in between. A simple way for the application to do this is to calculate a sequence of rotations

$$\mathbf{R}_i = \mathbf{R}_x(\frac{\theta_x}{n}i)\ \mathbf{R}_y(\frac{\theta_x}{n}i)\ \mathbf{R}_z(\frac{\theta_x}{n}i)$$

for $i \in 1, ..n$. The problem (finally) is that this sequence of rotations can often be wonky. (In the slides, I gave a few links to videos.)

3. A more advanced technique is to use *quaternions*. This is considered by experts to be the proper way to do rotations because it performs best from a numerical analysis point of view. This technique is rather complex. If you end up doing more advanced animation then you will need to learn it, but I feel it is not worth spending our time on now.

Scaling

Scaling changes the size and shape of an object. Scaling can be done in one direction only (turning a square into a rectangle, or a cube into a rectanguloid), or in two or three directions. To scale a scene by factors (s_x, s_y, s_z) in the three canonical directions, we can multiply by a diagonal matrix as follows:

$$\begin{bmatrix} s_x x \\ s_y y \\ s_z z \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}.$$

I neglected to mention this in class but, suppose we wanted to scale the scene in other directions. How would we do it? If we wanted to scale by factors (s_1, s_2, s_3) in the direction of axes defined by the rows of some rotation matrix $\mathbf{R}$, then we can do so by the transformation:

$$\mathbf{R}^T \begin{bmatrix} s_1 & 0 & 0 \\ 0 & s_2 & 0 \\ 0 & 0 & s_3 \end{bmatrix} \mathbf{R}.$$

This transformation projects to new canonical coordinate axes (namely, the ones we want to scale, which are the rows of $\mathbf{R}$), performs the scaling, then rotates back.

Translation

Another common transformation is translation. We may need to translate the location of objects in the world, or we may need to translate the location of the camera, or we may need to transform between coordinate systems (as we will do later in the lecture). Suppose we wish to translate all points (x, y, z) by adding some constant vector (t_x, t_y, t_z) to each point. This transformation cannot be achieved by a 3×3 matrix, unfortunately. So how do we do it?

The trick is to write out scene points (x, y, z) as points in $\Re^4$, namely we append a fourth coordinate with value 1. We can then translate by performing a matrix operation as follows:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}.$$

Interestingly, we can perform rotations and scaling using a 4×4 matrix $(x, y, z, 1)$ as well. Rotation and scaling matrices go into the upper-left 3×3 corner of the 4×4 matrix.

$$\begin{bmatrix} * & * & * & 0 \\ * & * & * & 0 \\ * & * & * & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

i.e. nothing happens to the fourth coordinate. For example, we could write a scaling transformation as follows:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}.$$

We will see over the next few lectures that this "trick" is quite useful other transformations that we wish to perform.

Homogeneous coordinates

We have represented a 3D point (x, y, z) as a point $(x, y, z, 1)$ in $\Re^4$. We now generalize this by allowing ourselves to represent (x, y, z) as *any* 4D vector of the form (wx, wy, wz, w) where $w \neq 0$. Note that the set of points

$$\{ \ (wx, wy, wz, w) \ : \ w \neq 0 \ \}$$

is a line in $\Re^4$ which passes through the origin (0,0,0) and through the point $(x, y, z, 1)$. We thus associate each point in $\Re^3$ with a line in $\Re^4$, in particular, a line that passes through the origin. All points along that line except the origin are considered equivalent in the sense that they all represent the same 3D scene point.

Note that if you add two 4D vectors together then the resulting 4D vector typically represents a different 3D point than what you might naively expect, that is,

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} + \begin{bmatrix} a' \\ b' \\ c' \\ d' \end{bmatrix} \not\equiv \begin{bmatrix} a + a' \\ b + b' \\ c + c' \\ d + d' \end{bmatrix}$$

[**Careful notation:**] where the + operator on the left means I am adding the corresponding 3D vectors, not the 4D vectors shown there. If this notation makes you uncomfortable, then what I really mean is:

$$\begin{bmatrix} a/d \\ b/d \\ c/d \end{bmatrix} + \begin{bmatrix} a'/d' \\ b'/d' \\ c'/d' \end{bmatrix} \neq \begin{bmatrix} (a + a')/(d + d') \\ (b + b')/(d + d') \\ (c + c')/(d + d') \end{bmatrix}$$

~~Only if $d = d'$ can we be sure that the equivalence holds. (Think why.)~~

Points at infinity

We have considered points (wx, wy, wz, w) under the condition that $w \neq 0$. Let's look at the remaining points $(x, y, z, 0)$, where at least one of x, y, z is non-zero i.e. we do not consider the case $(0, 0, 0, 0)$.

Consider (x, y, z, ϵ) where $\epsilon \neq 0$ and consider what happens as $\epsilon \to 0$. We can write

$$(x, y, z, \epsilon) \equiv (\frac{x}{\epsilon}, \frac{y}{\epsilon}, \frac{z}{\epsilon}, 1)$$

Thus as $\epsilon \to 0$, the corresponding 3D point goes to infinity, and stays along the line from the origin through the point $(x, y, z, 1)$. We identify the limit

$$lim_{\epsilon \to 0}\ (x, y, z, \epsilon) = (x, y, z, 0)$$

with a particular "point at infinity".

What happens to a point at infinity when we perform a rotation, translation, or scaling? Since the bottom row of each of these 4×4 matrices is (0,0,0,1), it is easy to see that these transformations map points at infinity to points at infinity. In particular, verify for yourself that:

- a translation matrix does not affect a point at infinity; i.e. it maps the point at infinity to itself.
- a rotation matrix maps a point at infinity in exactly the same way it maps a finite point, namely, $(x, y, z, 1)$ rotates to $(x', y', z', 1)$ if and only if $(x, y, z, 0)$ rotates to $(x', y', z', 0)$.
- a scale matrix maps a point at infinity in exactly the same way it maps a finite point, namely, $(x, y, z, 1)$ scales to $(s_x x, s_y y, s_z z, 1)$ if and only if $(x, y, z, 0)$ scales to $(s_x x, s_y y, s_z z, 0)$. Note that all scaling does is change the relative sizes of the x,y,z coordinates.

Lecture 3

View/eye/camera coordinates

In order to define an image of a scene, we need to specify the 3D position of the viewer and we also need to specify the orientation of the viewer. The way this is traditionally done is to specify a 3D "eye" point (viewer/camera position), and to specify a point in the scene where the eye is looking at – called the "look at" point. This specifies the viewer's z axis. We also need to orient the viewer to specify where the viewer's x and y axes point. This involves spinning the viewer around the viewer's z axis. I'll explain how this is done next. I will use the term "viewer", "eye", "camera" interchangably.

Let $\mathbf{p}_c$ be the position of the viewer in world coordinates. Let $\mathbf{p}_{lookat}$ be a position that the viewer is looking at. This defines a vector $\mathbf{p}_c - \mathbf{p}_{lookat}$ from the look at point to the viewer. Note that this vector points *towards the viewer*, rather than toward the look-at point. The camera's z axis is defined to be the normalized vector

$$\hat{\mathbf{z}}_c = \frac{\mathbf{p}_c - \mathbf{p}_{lookat}}{\parallel \mathbf{p}_c - \mathbf{p}_{lookat} \parallel}.$$

[ASIDE: in the lecture itself, I used the traditional notation and defined $\mathbf{N} = \mathbf{p}_c - \mathbf{p}_{lookat}$ and $\mathbf{n}$ to be the normalized version. But afterwards I changed my mind and decided this extra notation was not necessary. I mention it now only in case you are looking at the video of the lecture and wondering what the heck happened to the $\mathbf{N}$ and $\mathbf{n}$.]

We have specified the position of the camera and the direction where the camera is looking. We next need to orient the camera and specify which direction is "up", that is, where the y axis of the camera is pointing in world coordinates. The x axis is then specified at the same time since it must perpendicular to the camera's y and z axes.

We use the following notation. We specify the camera's x, y and z axes with the vectors $\hat{\mathbf{x}}_c$, $\hat{\mathbf{y}}_c$, and $\hat{\mathbf{z}}_c$. These vectors are typically different from the world coordinate x, y, z axes which are written $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$, and $\hat{\mathbf{z}}$ without the c subscript.

To construct the camera coordinate axes, recall Examples 1 and 2 from last lecture, when we specified one row of a rotation matrix and we needed to find two other rows. Here we face a similar task in defining the camera coordinates axes. Essentially we need to define a 3×3 rotation matrix whose rows are the vectors $\hat{\mathbf{x}}_c, \hat{\mathbf{y}}_c$, and $\hat{\mathbf{z}}_c$.

We will specify the camera's x and y axes as follows. We choose a vector – called the *view up* $\mathbf{V}_{up}$ vector – that is *not* parallel to $\hat{\mathbf{z}}_c$. It is common to define $\mathbf{V}_{up}$ to be the world $\hat{\mathbf{y}}$ vector, i.e. $(0, 1, 0)$. This works provided that the viewer is not looking in the $\mathbf{y}$ direction. Notice that the $\mathbf{V}_{up}$ vector does not need to be perpendicular to $\mathbf{z}_c$. Rather the constraint is much weaker: it only needs to be not parallel to $\mathbf{z}_c$.

In practice, the user specifies $\mathbf{V}_{up}$, $\mathbf{p}_c$, and $\mathbf{p}_{lookat}$. Then $\hat{\mathbf{x}}_c$ and $\hat{\mathbf{y}}_c$ are computed:

$$\hat{\mathbf{x}}_c \equiv \frac{\mathbf{V_{up}} \times (\mathbf{p_c} - \mathbf{p_{lookat}})}{\parallel \mathbf{V_{up}} \times (\mathbf{p_c} - \mathbf{p_{lookat}}) \parallel}, \qquad \hat{\mathbf{y}}_c \equiv \hat{\mathbf{z}}_\mathbf{c} \times \hat{\mathbf{x}}_\mathbf{c} \ .$$

These definitions are a bit subtle. Although we define $\mathbf{V}_{up}$ with $\mathbf{y}_c$ in mind, in fact $\mathbf{V}_{up}$ is used directly to define the $\hat{\mathbf{x}}_c$ vector, which then in turn defines $\hat{\mathbf{y}}_c$. Bottom line: we have a simple method for computing orthonormal camera coordinate vectors $\hat{\mathbf{x}}_c$, $\hat{\mathbf{y}}_c$, and $\hat{\mathbf{z}}_c$.

Example

Suppose that we place a viewer at world coordinate position $(2, 1, 1)$ and we orient the viewer to be looking at the origin. Then

$$\mathbf{p}_c - \mathbf{p}_{lookat} = (2, 1, 1) - (0, 0, 0) = (2, 1, 1).$$

Then $\mathbf{z}_c$ is just the normalized vector $\frac{1}{\sqrt{6}}(2, 1, 1)$. Suppose we let $\mathbf{V}_{up} = (0, 1, 0)$. What are $\mathbf{x}_c$ and $\mathbf{y}_c$? It is easy to calculate that

$$\mathbf{V}_{up} \times (\mathbf{p}_c - \mathbf{p}_{lookat}) = (1, 0, -2)$$

and so

$$\mathbf{x}_c = \frac{1}{\sqrt{5}}(1, 0, -2)$$

and

$$\mathbf{y}_c = \hat{\mathbf{z}}_c \times \hat{\mathbf{x}}_c = \frac{1}{\sqrt{30}}(-2, 5, -1).$$

Eye coordinates in OpenGL

OpenGL makes it very easy to specify the camera/eye coordinates. The user needs to specify three vectors: where the eye is located, the 3D point that the eye is looking at, and the "up" vector.

```
eye    =  ...    #  specify 3D coordinates of eye
lookat =  ...    #  discussed below
up     =  ...    #     "

gluLookAt(eye[0], eye[1], eye[2], lookat[0], lookat[1], lookat[2], up[0],up[1],up[2])
```

Transforming from world to viewer/camera/eye coordinates

We would like to make images from the viewpoint of an camera (eye, viewer) and so we need to express the positions of points in the world in camera coordinates. We transform from world coordinates to viewer coordinates using a 3D translation matrix $\mathbf{T}$ and a rotation matrix $\mathbf{R}$. Suppose we have a point that is represented in world coordinates by $(x, y, z)_{world}$. We wish to represent that point in camera coordinates $(x, y, z)_c$. To do so, we multiply by the matrices $\mathbf{R}\ \mathbf{T}$. What should these matrices be and in which order should we apply them?

We first use the matrix $\mathbf{T}$ to translate the eye position (expressed in world coordinates) to the origin. The rotation matrix $\mathbf{R}$ then should rotate $\Re^3$ such that the eye's axes (expressed in world coordinates) are mapped to the canonical vectors $(1, 0, 0), (0, 1, 0), (0, 0, 1)$. Together we have

$$\mathbf{M}_{viewer \leftarrow world} \ = \ \mathbf{R}\ \mathbf{T}.$$

The translation matrix $\mathbf{T}$ that takes $\mathbf{p}_c$ to the origin must be:

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & -p_{c,x} \\ 0 & 1 & 0 & -p_{c,y} \\ 0 & 0 & 1 & -p_{c,z} \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Note that we are transforming from the world coordinate to eye coordinates and so the position of the eye (in world coordinates) should be transformed to the origin (in eye coordinates).

These two vectors are the first two rows of **R**. The rotation matrix that rotates the world coordinates to viewer coordinates is **R** below. To simplify the notation, let **u**, **v**, **n** be $\hat{\mathbf{x}}_c$, $\hat{\mathbf{y}}_c$, and $\hat{\mathbf{z}}_c$, respectively, where these vectors are expressed in world coordinates.

$$\mathbf{R} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

[ASIDE: After class, one student asked about the "axis" of this rotation matrix. Such an axis certainly exists (see last lecture), but it is not usually how we think of the camera orientation. Rather, we usually think of the camera orientation in terms of the direction in which the camera is looking and the up vector.]

Object coordinates

Now that the camera has been specified, let's return to the issue of model transformations which we introduced last lecture. We will now start thinking about the objects that we want to draw! We begin by introducing some basic object models in OpenGL. Then we will look at how to transform these models so as to place them in 3D space i.e. world coordinates. We will finish the lecture by combining all our transformations together: from object coordinates to world coordinates to camera coordinates.

In classic OpenGL, one defines a point with a `glVertex()` command. There are several forms depending on whether the point is 2D or 3D or 4D, whether the coordinates are integers or floats, etc. whether the coordinates are a vector ("tuple") or are separate arguments.
`https://www.opengl.org/sdk/docs/man2/xhtml/glVertex.xml`
For example, if x, y, and z are floats, we can define a point:

```
glVertex3f(x,  y , z)
```

To define a line segment, we need two vertices which are grouped together:

```
glBegin(GL_LINES)
glVertex3f(x1,  y1 , z1)
glVertex3f(x2,  y2 , z2)
glEnd()
```

To define n line segments, we can list 2 n vertices. For example, to define two line segments:

```
glBegin(GL_LINES)
glVertex3f(x1,  y1 , z1)
glVertex3f(x2,  y2 , z2)
glVertex3f(x3,  y3, z3)
glVertex3f(x4,  y4 , z4)
glEnd()
```

To define a sequence of lines where the endpoint of one line is the begin point of the next, i.e. a polygon, we write:

```
glBegin(GL_POLYGON)
glVertex3f(x1,  y1 , z1)
glVertex3f(x2,  y2 , z2)
glVertex3f(x3,  y3 , z3)
glVertex3f(x4,  y4 , z4)
glEnd()
```

This automatically connects the last point to the first.

Typically we define surfaces out of triangles or quadrangles ("quads"). Quads are four sided polygons and ultimately are broken into triangles. One can declare triangles or quads similar to the above, but use GL_TRIANGLES or GL_QUADS instead of GL_POLYGON.

Quadrics and other surfaces

One simple class of surfaces to draw is the sphere and cylinder and other quadric (quadratic) surfaces such as cones and hyperboloids. A general formula for a quadric surface in $\Re^3$ is

$$ax^2 + by^2 + cz^2 + dxy + eyz + fxz + gx + hy + iz + j = 0.$$

Simple examples are:

- an ellipsoid centered at (x_0, y_0, z_0), namely
$$a(x - x_0)^2 + b(y - y_0)^2 + c(z - z_0)^2 - 1 = 0$$
where $a, b, c > 0$. Note that this includes the special case of a sphere.
- a cone with apex at (x_0, y_0, z_0), and axis parallel to x axis
$$a(x - x_0)^2 = (y - y_0)^2 + (z - z_0)^2$$
- a paraboloid with axis parallel to x axis
$$ax = (y - y_0)^2 + (z - z_0)^2$$
- etc

The above general formula for a quadric can be rewritten as:

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} a & \frac{d}{2} & \frac{f}{2} & \frac{g}{2} \\ \frac{d}{2} & b & \frac{e}{2} & \frac{h}{2} \\ \frac{f}{2} & \frac{e}{2} & c & \frac{i}{2} \\ \frac{g}{2} & \frac{h}{2} & \frac{i}{2} & j \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = 0$$

or

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} \mathbf{Q} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = 0$$

where $\mathbf{Q}$ is a symmetric 4×4 matrix. For example, if a, b, c are all positive, then

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = 0$$

is an ellipsoid centered at the origin.

[————————**The rest of the page was modified Feb. 16**————————————-]

Suppose that we *arbitrarily* insert a rotation and translation matrix as follows:

$$(\mathbf{RTx})^T \mathbf{Q} (\mathbf{RTx}) = 0 \qquad (*)$$

Intuitively, this must give us a new ellipsoid since all we are doing is translating and rotating. So what is the new quadratic matrix that describes this new ellipsoid?

To answer this question, consider a new variable **x'**,

$$\mathbf{x}' = \mathbf{RTx}. \qquad (**)$$

Since

$$(\mathbf{x}')^{\mathbf{T}} \mathbf{Q} \mathbf{x}' = \mathbf{0}$$

the points **x'** that satisfy this equation must be the points on the original ellipsoid. We now rewrite (*) to get

$$\mathbf{x}^T (\mathbf{T}^T \mathbf{R}^T \mathbf{QRT}) \mathbf{x} = 0$$

that is,

$$\mathbf{x}^T \mathbf{Q}' \mathbf{x} = 0$$

for some new quadratic matrix **Q'**. This is the quadratic matrix for the new ellipsoid that we obtain when we map **x** using (**).

How is the new ellipsoid related to the original one? Well, from (**),

$$\mathbf{T}^{-1} \mathbf{R}^{\mathbf{T}} \mathbf{x}' = \mathbf{x}.$$

Thus, if we take the points **x'** on the original ellipsoid, rotate them by $\mathbf{R}^T$ and apply the inverse translation $\mathbf{T}^{-1}$, we get the points **x** on the new ellipsoid. So, the new ellipsoid is obtained from the original by a rotation and then a translation.

OpenGL does not include general quadrics as geometric primitives but there is an OpenGL *Utility Library* (GLU library) that includes some quadrics, in particular, spheres and cylinders.

```
myQuadric = gluNewQuadric()
gluSphere(myQuadric,   ...)
gluCylinder(myQuadric, ...)
```

where `gluSphere()` and `gluCylinder()` have parameters that need to be defined such as the number of vertices used to approximate the shape. The cylinder is really a more general shape which includes a cone. `https://www.opengl.org/sdk/docs/man2/xhtml/gluCylinder.xml`

There is another library called the OpenGL Utility Toolkit (GLUT) which allows us to define other shapes:

```
glutSolidCone()
glutSolidCube()
glutSolidTorus()
glutSolidTeapot()     //  yes, a teapot
```

How to transform an object in OpenGL?

When you define an object, whether it is a vertex or line or a teapot, the object is defined in "object coordinates". To place the object somewhere in the world and to orient it in the world, you need to apply a transformation – typically a rotation and translation, but also possibly a scaling. To transform object, OpenGL provides translation, rotation and scaling transformations:

```
glTranslatef(x,y,z)
glRotatef(vx, vy, vz, angle )
glScalef(sx, sy, sz)
```

Notice from the syntax that these do not operate on particular objects (or geometric primitives). Rather, they are applied to all (subsequently defined) geometry primitives.

Before we see how this works, let me sketch the general pipeline. You define an object – say its a dog – by specifying a bunch of spheres (head and torso) and cylinders (legs, tail) or maybe you are more sophisticated and build the dog out of thousands of triangles. These primitives are defined with respect to an object coordinate system. You place each sphere, triangle, etc at some position in this coordinate system and you orient the primitives. This object modelling step is done with a bunch of transform commands (translate, rotate, scale) and primitive declarations. Let's refer to one these transform commands specifically as $\mathbf{M}_{object \leftarrow vertex}$. It places a vertex (or primitive e.g. the dog's ear) somewhere in the dog coordinate system.

The dog object then needs to be placed in the scene and oriented in the scene, and so another set of transformations must be applied, namely a translation and rotation that map all the vertices of the dog from dog coordinates to world coordinates. Finally, as we discussed in the first part of this lecture, we will need to transform from world coordinates to viewer coordinates.

$$\mathbf{M}_{viewer \leftarrow world} \, \mathbf{M}_{world \leftarrow object} \, \mathbf{M}_{object \leftarrow vertex}$$

OpenGL GL_MODELVIEW matrix

We have covered all the transformations we need. We just now need to assemble them together and order them properly. Specifically, how is this done in OpenGL?

OpenGL is a "state machine" which means that it has a bunch of global variables (which you can access through getters and setters). One of these states is a 4×4 transformation matrix called GL_MODELVIEW. Whenever your program declares a vertex (glVertex) or any other geometric object, this vertex is immediately sent through the pipeline of transformations mentioned above, from object to world to camera coordinates.

The GL_MODELVIEW matrix should be initialized to be the identity matrix.

```
glMatrixMode(GL_MODELVIEW)
glLoadIdentity()
```

At this initialization stage, the view coordinates, camera coordinates, and object coordinates are all the same. The viewer is then defined with a gluLookat command (see p. 2).

```
gluLookat( ... )
```

This multiplies the GL_MODELVIEW matrix by the appropriate matrix that transforms from world to viewer coordinates. In the first part of this lecture I said how this matrix was constructed out of a rotation and translation.

At this point, the object coordinate system and world coordinate system are identical. That is, if an object is defined e.g. by glVertex commands, then the object is placed at the origin in world coordinates. OpenGL automatically multiplies the vertices of the object by the GL_MODELVIEW matrix, and thereby transforms from object coordinates (which are the same as world coordinates in this case) to viewer coordinates.

Generally we want to position objects somewhere other than the origin in world coordinates and to give them an arbitrary orientation as well. To do so, we need to transform from object to world coordinates. Prior to defining the vertices of the object, we call glTranslate and glRotate which together define $\mathbf{M}_{world \leftarrow object}$. So, recalling that gluLookAt defines $\mathbf{M}_{viewer \leftarrow world}$ and multiplies it with the GL_MODELVIEW matrix (which should have been initialized to the identity), the following commands will compute the position of the vertex in camera coordinates as follows:

$$\mathbf{M}_{viewer \leftarrow world} \mathbf{M}_{world \leftarrow object} \mathbf{x}_{obj}$$

Here is the OpenGL sequence:

```
gluLookAt( ... )          // transform from world to viewer  coordinates

glRotate( ... )           // transform from dog to world coordinates
glTranslate( ... )        //  i.e.  translate and rotate the dog

glVertex(  )              //  a vertex of the dog object
```

What's happening here is that each of the first three of the above commands multiplies the GL_MODELVIEW matrix on the right by some new transformation matrix, giving

$$\mathbf{M}_{\texttt{GL_MODELVIEW}} = \mathbf{M}_{viewer \leftarrow world} \, \mathbf{R} \, \mathbf{T}.$$

Conceptually, the GL_MODELVIEW matrix is a composition of two maps. There is the map from object coordinates to world coordinates, and there is a map from world coordinates to viewer coordinates.

$$\mathbf{M}_{\texttt{GL_MODELVIEW}} = \mathbf{M}_{viewer \leftarrow world}\ \mathbf{M}_{world \leftarrow object}$$

But there is no guarentee that this is what you will get. For example, if you mistakenly write out two consecutive gluLookAt commands, then $\mathbf{M}_{\texttt{GL_MODELVIEW}}$ will be updated as follows:

$$\mathbf{M}_{\texttt{GL_MODELVIEW}} \leftarrow \mathbf{M}_{\texttt{GL_MODELVIEW}}\ \mathbf{M}_{viewer \leftarrow world}\ \mathbf{M}_{viewer \leftarrow world}$$

which is non-sense.

Finally, lets think about how to define multiple objects, for example, a dog and a house. You might think you could write:

```
gluLookAt( ... )         // transform from world to viewer  coordinates

glTranslate( ... )          // transform from dog to world coordinates
glRotate( ... )       //  i.e.  translate and rotate the dog
defineDog(  )            //  all vertices of the dog object

glTranslate( ... )          // transform from house to world coordinates
glRotate( ... )       //  i.e.  translate and rotate the house
defineHouse(  )          //  all vertices of the house object
```

However, this doesn't work since it would transform the house vertices from house coordinates to dog coordinates, when instead we want to transform the house vertices into world coordinates (unless of course we *wanted* to define the house with respect to the dog position, which is unlikely...)

To get around this problem, OpenGL has a GL_MODELVIEW stack. The stack allows you to remember (push) the state of the GL_MODELVIEW matrix and replace (pop) the GL_MODELVIEW matrix by the most recent remembered state. Here is an how the last example would work:

```
gluLookAt( ... )         // transform from world to viewer  coordinates

glPushMatrix()
glTranslate( ... )          // transform from dog to world coordinates
glRotate( ... )       //  i.e.  translate and rotate the dog
defineDog(  )            //  all vertices of the dog object
glPopMatrix()

glPushMatrix()
glTranslate( ... )          // transform from house to world coordinates
glRotate( ... )       //  i.e.  translate and rotate the house
defineHouse(  )          //  all vertices of the house object
glPopMatrix()
```

One final point: you'll note that in the above we first write translate and then rotate (**T R**), whereas when we transformed from world to viewer coordinates (bottom p. 2), we applied them in the opposite order (**R T**). As an exercise, ask yourself why.

Lecture 4

Up to now we have defined points in a 3D world coordinate space and we have considered how to transform into a 3D camera coordinate space. Our next step is to project the points into an image. Images are 2D, not 3D, and so we need to map 3D points to a 2D surface (a plane).

Orthographic projection

A first guess on how to do this is to just throw away the z coordinate and leave the x and y values as they are. This is called the *orthographic* projection in the z direction. We can write this as a mapping to the $Z = 0$ plane. This mapping can be represented by a 4×4 matrix:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

This is called an orthographic projection.

More generally, an *orthographic projection* is a mapping from 3D to a plane such that each point (x, y, z) is mapping in a direction that is parallel to the normal to the plane. The "view from above" and "view from the side" sketches shown later in this lecture can be thought of an orthographic projection in the y and x directions, respectively.

In class (see the slides), I presented a number of visual examples of orthographic projections, including the easy-to-think about x, y, and z orthographic projections. I also presented a projection onto the plane $x + y + z = 0$. This projection has a special name: an *isometric* projection. The name comes from the fact that the projected unit vectors (x, y, z axes of some object *e.g.* a cube) all have the same length in the image.

Parallel Projection

A slightly more general way to project the world onto the $z = 0$ plane is to project in direction $\mathbf{p} = (p_x, p_y, p_z)$. We assume $p_z \neq 0$ in order for this to make sense.

How do we project in direction $\mathbf{p}$? Consider any 3D point (x, y, z). Suppose this point projects to $(x^*, y^*, 0)$, i.e.

$$(x, y, z) - t(p_x, p_y, p_z) = (x^*, y^*, 0)$$

for some t. To compute x^*, y^*, we solve for t first:

$$z - tp_z = 0$$

so $t = \frac{z}{p_z}$. Thus,

$$x^* = x - (p_x/p_z)z$$
$$y^* = y - (p_y/p_z)z$$

In homogeneous coordinates, we can represent this projection as follows:

$$\begin{bmatrix} x^* \\ y^* \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -p_x/p_z & 0 \\ 0 & 1 & -p_y/p_z & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Note we really don't *need* homogeneous coordinates here. i.e. the fourth row and fourth column don't do anything interesting. But I want to be consistent since pretty much everything we do from here on is going to use homogeneous coordinates.

Note that orthographic projection is the special case of parallel projection in which $p_x = p_y = 0$. Any other case of parallel projection is called *oblique* projection. In class, I discussed briefly two common types of oblique projections, namely *cavalier* and *cabinet*. The latter in particular often are used in furniture illustrations e.g. instructions for assembling Ikea furniture.

Let's now turn to a model which more closely approximates the image formation that we experience with our eyes and with cameras.

Perspective Projection

The standard model of image formation that is used in computer graphics is that a 3D scene is projected towards a single point – called the *center of projection*. This center of projection is just the position of the camera.

The image is not defined at the projection point, but rather it is defined on a plane, called the *projection plane*. The projection plane is perpendicular to the camera z axis. For real cameras, the projection plane is the sensor plane which lies behind the camera lens. The same is true for eyes, although the projection surface on the back of your eye (the retina) isn't a plane but rather it is the spherically curved surface. In both of these cases, the image of the scene is upside down and backwards on the projection surface.

In computer graphics we define a projection plane to lie in front of the center of projection. (See illustrations on next page.) In camera coordinates, the center of projection is $(0, 0, 0)$ and the projection plane is $z = f$ where $f < 0$ since we are using right handed coordinates. A 3D point (x, y, z) is projected to (x^*, y^*, f). Using similar triangles, we observe that the projection should satisfy:

$$\frac{x}{z} = \frac{x^*}{f} \qquad\qquad \frac{y}{z} = \frac{y^*}{f}$$

and so the projection maps the 3D point (x, y, z) to a point on the projection plane,

$$(x, y, z) \rightarrow (f\ \frac{x}{z}, f\ \frac{y}{z}, f).$$

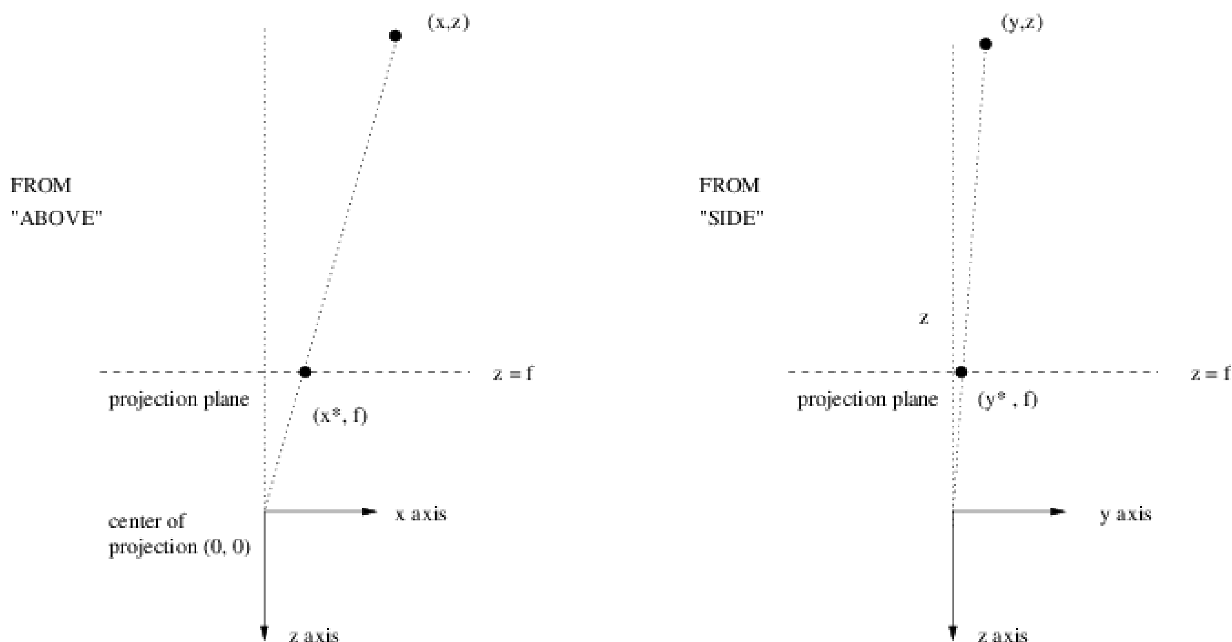
Using homogeneous coordinates to define a projection

The homogeneous coordinate representation allows us to write projection mapping. For any $z \neq 0$, we can re-write our projected points as:

$$(f\frac{x}{z}, f\frac{y}{z}, f, 1) \equiv (xf, yf, fz, z).$$

This allows us to represent the projection transformation by a 4×4 matrix, i.e. :

$$\begin{bmatrix} fx \\ fy \\ fz \\ z \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & f & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$



A few observations can be made. First, since we are now treating all 4D points (wx, wy, wz, w) as representing the same 3D point (x, y, z), we can multiply any 4×4 transformation matrix by a constant, without changing the transformation that is carried out by the matrix. For example, the above matrix can be divided by the constant f and written instead as:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{f} & 0 \end{bmatrix}$$

This matrix, like the one above, projects the 3D scene onto the plane $z = f$, with center of projection being the origin $(0, 0, 0)$.

A second observation (which I neglected to mention in class) is that, whereas the 4×4 translation, rotation, and scaling matrices were invertible (and hence of rank 4), the two projection matrices are not. They are of rank 3 since the third and fourth rows are linearly dependent.

Vanishing points

One well known concept from perspective geometry is that parallel lines in the scene intersect at a common point in the image plane. The most familiar example is two train tracks converging at the horizon. The point where the lines intersect is called a *vanishing point*.

How can we express vanishing points mathematically? Consider a family of lines of the form,

$$\{(x_0, y_0, z_0) + t(v_x, v_y, v_z)\}$$

where (v_x, v_y, v_z) is some non-zero constant vector. For any point (x_0, y_0, z_0), we get a line in the direction of this vector (v_x, v_y, v_z). So for a bunch of different points, (x_0, y_0, z_0), we get a bunch of parallel lines. What happens along each of the lines as we let $t \to \infty$?

We write points in homogeneous coordinates as:

$$\{(x_0 + tv_x, y_0 + tv_y, z_0 + tv_z, 1)\}$$

Now dividing by t and taking the limit gives

$$\lim_{t \to \infty} (\frac{x_0}{t} + v_x, \frac{y_0}{t} + v_y, \frac{z_0}{t} + v_z, \frac{1}{t}) = (v_x, v_y, v_z, 0)$$

so all the lines go to the same point at infinity. Thus, when we project the lines into the image, the lines all intersect at the image projection of the point at infinity.

What happens when we use perspective projection to project a point at infinity $(v_x, v_y, v_z, 0)$ onto a projection plane?

$$\begin{bmatrix} fv_x \\ fv_y \\ fv_z \\ v_z \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & f & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \\ 0 \end{bmatrix}$$

The projected point is called the *vanishing point* in the image. If $v_z \neq 0$, then the projected point written as 3D vector is $(\frac{v_x}{v_z} f, \frac{v_y}{v_z} f, f)$. This is clearly a finite point. However, if $v_z = 0$, then the projected point itself is a point at infinity.

If you took an art class in high school, you may have learned about *n point perspective*. You may remember this has something to do with vanishing points. (See the slides for example images.) With the above discussion, we can now define n-point perspective in terms of points at infinity.

N-point perspective refers to an image of a manmade scene or object that contains multiple sets of parallel lines. Typically the object or scene has a natural xyz coordinate system and the lines in the object/scene are aligned with these coordinate axes, but this is not a requirement. The requirement is just that the scene contains at least one set of parallel lines.

We say an image has n-point perspective if there are *n finite* vanishing points in the image projection plane. The 'finite vanishing point' condition is equivalent to saying that the parallel lines have a direction vector (v_x, v_y, v_z) such that $v_z \neq 0$. Parallel lines in directions $(1, 0, 0)$ and $(0, 1, 0)$, in particular, do *not* have this property. These vectors produce vanishing points at infinity. See slides for examples of 1–, 2–, and 3– point perspective.

View Volume (Frustum)

We have considered how, in perspective projection, points in a 3D scene project onto a 2D plane. We considered all 3D points. Note, however, that only part of the scene needs to be considered. For example, points behind the camera are not visible and hence need not be considered. In addition, many points in front of the camera also need not be considered. For example, typical cameras have an angular field of view of between 30 and 60 degrees only (depending on whether the lens is telephoto, regular, or wide angle). Let's next formalize this notion of field of view of an image and show how to express it using projective transformations and homogeneous coordinates.

To make an image of a scene, we want to consider only scene points that (1) project toward the origin, and (2) project to a finite rectangular *window* in the projection plane, and (3) lie over a given range of z values, i.e. "depths". This set of points is called the *view volume* or *view frustum*.[3]

[3] According to wikipedia, a "frustum" is the portion of a solid normally a cone or pyramid which lies between two parallel planes cutting the solid. Note the word is "frustum", not "frustrum".

The shape of the view volume is a truncated pyramid. (See the slides for illustrations.) We will only want to draw those parts of the scene that lie within this view volume. Those points that lie outside the view volume are not part of our picture. They are "clipped." I will say more about how clipping works next lecture.

Let's now define the view frustrum. Let xyz be camera coordinates. Define the sides of the view volume by four sloped planes,

$$\frac{x}{z} = \pm \tan(\frac{\theta_x}{2})$$
$$\frac{y}{z} = \pm \tan(\frac{\theta_y}{2})$$

where θ_x and θ_y define angular field of view in the x and y directions. We also restrict ourselves to a range of depths lying between some near and far z plane. Keep in mind that we care only about points where $z < 0$ i.e. the camera is looking in the $-z$ direction. Let near and far be the (positive) distances to the planes. Then

$$0 \leq \text{near} \leq -z \leq \text{far}$$

or if you prefer

$$-\text{far} \leq z \leq -\text{near} \leq 0.$$

In OpenGL, rather than defining such a view volume by the parameters $\{\theta_x, \theta_y, \text{near}, \text{far}\}$, the view volume is defined with parameters as follows:

```
gluPerspective(theta_y, aspect, near, far).
```

The third and fourth parameters do as expected. It is the first and second parameters that require some thought. The first parameter specifies θ_y in degrees. The second parameter specifies the *aspect ratio*

$$\text{aspect} \equiv \frac{\tan(\theta_x/2)}{\tan(\theta_y/2)}.$$

Note that this is *not* the ratio of the angles, but rather it is the ratio of x size to y size of the rectangle that is defined by any depth slice through the frustrum. (Typically – but not necessarily – the aspect ratio is defined to match that of the window on a display where the user knows the x and y size of the display in pixels.)

An equivalent way to define the above view volume is to choose a depth plane, say the near plane, and define a square on the plane which corresponds to the angles θ_x and θ_y. This square is also defined by the intersection of the four sloped planes with the near plane, so

$$x = \pm \text{near} \tan(\frac{\theta_x}{2})$$

$$y = \pm \text{near} \tan(\frac{\theta_y}{2})$$

The x values are the left and right edge of the square, and the y values here are the bottom and top of the square. (Of course, the far value is also needed to complete the definition of the view volume.)

OpenGL allows a slightly more general definition that this. One may define any rectangle in the near plane such that the sides of the rectangle are aligned with the x and y axes. One uses:

$$\texttt{glFrustum}(\texttt{left}, \texttt{right}, \texttt{bottom}, \texttt{top}, \texttt{near}, \texttt{far}).$$

Here the variables `left` and `right` refer to the limits of x on the near plane ($z = f_0 = -\texttt{near}$) and `bottom` and `top` refer to the limits of y on the near plane. Note that this is a more general definition than `gluPerspective`, which essentially required that `left = -right` and `bottom = -top`,

You may wonder why anyone would need the generality of the `glFrustum`. One nice application is to create 3D desktop displays such as stereo displays (like 3D movies). For stereo displays, you need to consider that there are two eyes looking at (through) the same image window, and so the position of the window differs in the coordinate systems of the two eyes. See the slides for an illustration. Another nice example is shown in this interesting video: `https://www.youtube.com/watch?v=Jd3-eiid-Uw`. Again, the basic idea is to consider the frame of your display monitor to be a window through which you view a 3D world. As you move, the position of the display changes with respect to your own position. If you could get a external camera or some other "motion capture" system to track your position when viewing the display, then the frustum could be defined by the relationship between the display frame and your (changing) viewing position. [I don't expect you to understand the mathematical details of this method from my brief summary here, or even from watching the video. I just want you to get the flavor.]

Lecture 5

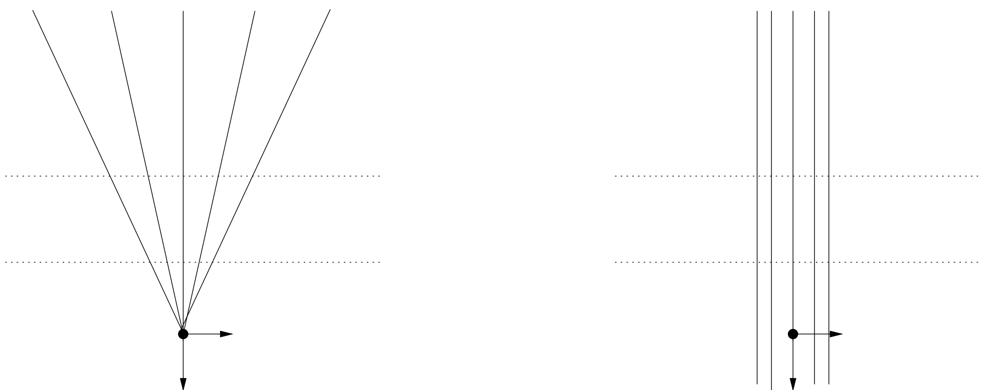
Projective transformation (from eye to clipping coordinates)

Last lecture we considered how to project 3D points to the 2D projection plane. The problem with doing this is that we would lose the information about the depth of the 3D points. We need to preserve depth information in order to know which surfaces are visible in an image, i.e. if surface A lies between the camera and surface B, then A is visible but B is not.

Specifically, recall the projection matrix we saw last class, where $f < 0$,

$$\begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & f & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

This matrix is non-invertible. You can easily see this mathematically by noting that the 3rd and 4th rows are linearly dependent. To keep information about the depth (z) ordering of points, we will choose a slightly different 4×4 transformation that it *is* invertible. This is called a *projective transformation*. The intuitive idea is shown in this figure. We are going to transform the scene by taking lines that pass through the camera position and make these lines parallel. (In the figure on the left, the lines stop at the origin – but this is just for illustration purposes.) Such a transformation will take lines that *don't* pass through the camera position and transform them too, but we don't care about those lines since they play no role in our image.



Define the near and far planes as before to be at $z = f_0 = -near$ and $z = f_1 = -far$, respectively. To get an invertible matrix, we change the third row of the above projection matrix:

$$\begin{bmatrix} f_0 & 0 & 0 & 0 \\ 0 & f_0 & 0 & 0 \\ 0 & 0 & \alpha & \beta \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

and require only that $\beta \neq 0$ which forces the matrix to be invertible since the four rows are linearly independent. We choose α and β such that

- the z value of points *on the near and far planes* ($z = -near$ and $z = -far$, respectively) is preserved under this transformation. That is, $z = f_0$ maps to $z = f_0$, and $z = f_1$ maps to $z = f_1$.
- the (x, y) values are preserved within the plane $z = f_0$ (and by the previous point, it follows that the (x, y, z) values in the plane $z = f_0$ are *all* preserved under the transformation.)

As I show in the Appendix, the values are $\alpha = f_0 + f_1$ and $\beta = -f_0 f_1$. I verified in the lecture slides that the following matrix has the properties just defined, but you should do it for yourself now as an exercise.

$$\begin{bmatrix} f_0 & 0 & 0 & 0 \\ 0 & f_0 & 0 & 0 \\ 0 & 0 & f_0 + f_1 & -f_0 f_1 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

This matrix maps a scene point from eye coordinates to a new coordinate system called *clipping coordinates*. The term 'clipping coordinates' refers to a 4D homogeneous coordinate representation of a point, although keep in mind that the 4D point represents a 3D point. We'll discuss the term "clipping" in more detail next lecture. For now, let's better understand what this transformation does.

As sketched above, this transformation maps lines that pass through the origin to lines that are parallel to the z direction. Let's examine the points that are outside the view volume in more detail. How are they transformed by the projective transformation above? This analysis turn out to be useful next lecture when we ask how to discard ("clip") points that are outside the view volume.

First, how are points at infinity transformed? These points are outside the (finite) view volume, but it is not clear where they map to. Applying the projective transform gives:

$$\begin{bmatrix} f_0 & 0 & 0 & 0 \\ 0 & f_0 & 0 & 0 \\ 0 & 0 & f_0 + f_1 & -f_0 f_1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix} = \begin{bmatrix} f_0 x \\ f_0 y \\ (f_0 + f_1) z \\ z \end{bmatrix} = \begin{bmatrix} f_0 \frac{x}{z} \\ f_1 \frac{y}{z} \\ f_0 + f_1 \\ 1 \end{bmatrix}$$

Thus, all points at infinity map to the plane $z = f_0 + f_1$, which is beyond the far plane. Keep in mind that f_0 and f_1 are both negative.

Next, where are points at $z = 0$ mapped to? Note that $z = 0$ is the plane containing the camera, since the context is that we are mapping *from* camera coordinates *to* clip coordinates.

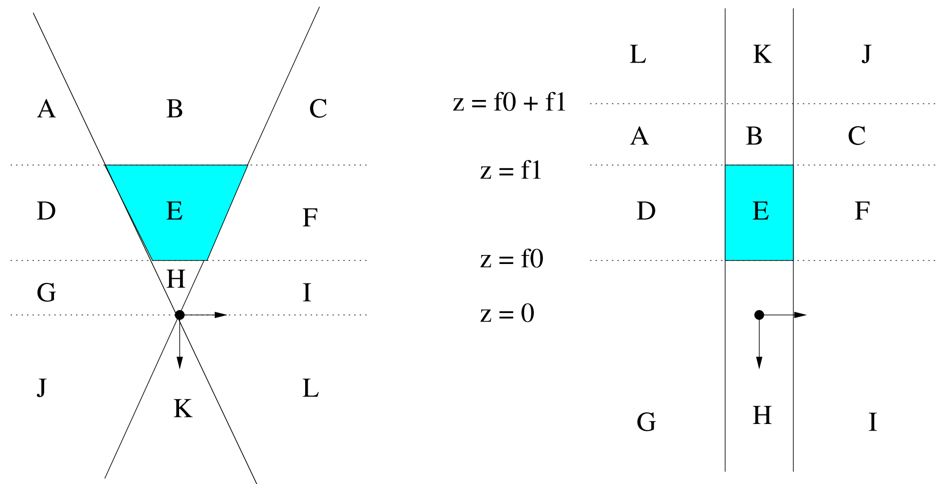
$$\begin{bmatrix} f_0 & 0 & 0 & 0 \\ 0 & f_0 & 0 & 0 \\ 0 & 0 & f_0 + f_1 & -f_0 f_1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} f_0 x \\ f_0 y \\ -f_0 f_1 \\ 0 \end{bmatrix} = \begin{bmatrix} x \\ y \\ -f_1 \\ 0 \end{bmatrix} = \begin{bmatrix} -x \\ -y \\ f_1 \\ 0 \end{bmatrix}$$

Thus, $z = 0$ is mapped to points at infinity.

For a general scene point, we have:

$$\begin{bmatrix} f_0 & 0 & 0 & 0 \\ 0 & f_0 & 0 & 0 \\ 0 & 0 & f_0 + f_1 & -f_0 f_1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} f_0 x \\ f_0 y \\ (f_0 + f_1) z - f_0 f_1 \\ z \end{bmatrix} = \begin{bmatrix} \frac{f_0}{z} x \\ \frac{f_0}{z} y \\ (f_0 + f_1) - \frac{f_0 f_1}{z} \\ 1 \end{bmatrix}$$

Let's compare the space in front of the camera ($z < 0$) to the space behind the camera ($z > 0$). For the space behind the camera, $\frac{f_0}{z} < 0$ since $f_0 < 0$ and $z > 0$ for these points. Thus this transformation changes the sign (and magnitude) of the x and y coordinates. Moreover, since $\frac{f_0 f_1}{z} > 0$, the half space $z > 0$ (JKL in figure below) is mapped to $z < f_0 + f_1$, that is, it maps to a space *beyond* the far plane. Combining the above two observations, we see that the regions labelled JKL on the left are mapped to the regions LKJ on the right.



Since we are only going to draw those parts of the scene that lie within the view volume (region E), you might think that it doesn't matter that the regions JKL behave so strangely under the projective transformation. However, *all* points gets transformed to clipping coordinates. In order to reason about which points can be discarded (or "clipped"), we need to keep all the regions in mind.

In particular, note that $(x, y, z, 1)$ maps to a 4D point (wx', wy', wz', w) such that $w = z$. (Verify this above!) Since points in the view volume always have $z < 0$, it follows that if we were to use this projection matrix above then we would have a negative w coordinate. OpenGL's avoids this by defining the $\mathbf{M}_{projective}$ matrix to be the negative of the above matrix, namely OpenGL uses:

$$\begin{bmatrix} -f_0 & 0 & 0 & 0 \\ 0 & -f_0 & 0 & 0 \\ 0 & 0 & -(f_0 + f_1) & f_0 f_1 \\ 0 & 0 & -1 & 0 \end{bmatrix} = \begin{bmatrix} \text{near} & 0 & 0 & 0 \\ 0 & \text{near} & 0 & 0 \\ 0 & 0 & \text{near} + \text{far} & \text{near} * \text{far} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Taking the negative of the matrix doesn't change the mapping. But the new matrix instead has the property that $w > 0$ and $z < 0$ for points in the view volume. As I'll mention briefly again below, this slightly simplifies the reasoning about whether a point *could* be in the view volume.

Normalized view volume

After performng the perspective transformation which maps the truncated pyramid frustum to a rectanguloid volume, it is common to transform once more, in order to normalize the rectanguloid volume to a cube centered at the origin.

$$[\text{left}, \text{right}] \times [\text{bottom}, \text{top}] \times [-\text{far}, -\text{near}] \quad \rightarrow \quad [-1,1] \times [-1,1] \times [-1,1],$$

This transformation is achieved by a translation, followed by a scaling, followed by a translation:

$$\mathbf{M}_{normalize} = \begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & 0 \\ 0 & \frac{2}{top-bottom} & 0 & 0 \\ 0 & 0 & \frac{-2}{far-near} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -left \\ 0 & 1 & 0 & -bottom \\ 0 & 0 & 1 & near \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The first translation lines up the left, bottom, and near planes with $x = 0, y = 0, z = 0$ respectively. The second transformation scales the x, y, z dimensions of the rectanguloid to a cube of width and height and depth of 2. The final translation shifts the cube so it is centered at the origin.

Notice the -2 factor instead of 2 in the (3,3) component of the scaling matrix. The z value is not merely scaled by 2, but it is also negated (i.e. reflection about the $z = 0$ plane). This flipping of the z axis switches from the usual right handed coordinate system to a left handed coordinate system. With the back of the hand facing toward you, x (thumb) is to the right, y (index) is up, and z (middle) points *away* from the camera. i.e. visible points that are farther from the camera now have a greater z values.

Clip Coordinates

The $\mathbf{M}_{normalize}$ transform does not affect the 4th coordinate (w) so if $w > 0$ prior to the normalization then it will be so after normalization as well. Do not confuse this with what happens to the sign of z. The normalization flips z (the factor -2 above) and maps it to $[-1, 1]$ so it can easily happen that the sign of z changes. But the sign of w does not change.

After normalization, the 3D points that lie within the view volume will all be normalized to the 4D points (wx, wy, wz, w) such that the x, y, z values all lie in $[-1, 1]$ and $w > 0$. Thus, these values all satisfy all of the following inequalities:

$$w > 0$$

$$-w \leq wx \leq w$$

$$-w \leq wy \leq w$$

$$-w \leq wz \leq w.$$

It is easy to check if these inequalities hold for a given point (wx, wy, wz, w) and hence if the point needs to be considered further or discarded (or "clipped"). That is the reaons such coordinates are referred to as *clip coordinates*.

[ASIDE: After the lecture, I was asked why we need to do the projective transform. It seems easy enough to do clipping in the original space, no? In particular, testing that z is between $-near$ and $-far$ is easy enough in the original space. So why make things so complicated? The answer is that, although the z condition is about equally easy to check in the original (camera) coordinates as in the clip coordinates, checking whether a point lies beyond the slanted side walls of the view volume is clearly more complicated in the camera coordinates than in clip coordinates. Bottom line: we don't *need* to do all these tests in clip coordinates, but it is simple to do.]

Summary (thus far) of the OpenGL pipeline

The last few lectures we have developed a sequence of transformations:

$$\mathbf{M}_{normalize}\mathbf{M}_{projective}\mathbf{M}_{camera\leftarrow world}\mathbf{M}_{world\leftarrow object}$$

where the transformations are applied from right to left.

In OpenGL, the transformation defined by the product $\mathbf{M}_{normalize}\mathbf{M}_{projective}$ is represented by a 4×4 `GL_PROJECTION` matrix. This matrix is defined by the commands `glFrustrum` or alternatively `gluPerspective` which I introduced last lecture. Note that *near* and *far* only play a role in $\mathbf{M}_{projective}$ whereas, as we saw last page, *top, bottom, left, right* are used for $\mathbf{M}_{normalize}$

The product $\mathbf{M}_{camera\leftarrow world}\mathbf{M}_{world\leftarrow object}$ is represented by the 4×4 `GL_MODELVIEW` matrix. The matrix $\mathbf{M}_{camera\leftarrow world}$ is specified by the OpenGL command `gluLookAt`. The matrix $\mathbf{M}_{world\leftarrow object}$ is specified by a sequence of `glRotate`, `glTranslate`, and `glScale` commands which are used to put objects into a position and orientation in the world coordinate system. These transformations are also used to position and orient and scale the parts of an object relative to each other.

Perspective division, normalized device coordinates

In order to explicitly represent the 4-tuple (wx, wy, wz, w) as a 3-tuple (x, y, z), one needs to divide the first three components of (wx, wy, wz, w) by the fourth component i.e. the w value. This step is called *perspective division*. It transforms from 4D clip coordinates (wx, wy, wz, w) to a 3D normalized view coordinates (x, y, z), also known as *normalized device coordinates* (NDC). Note that NDC is in left handed coordinates, with z increasing away from the camera.

In OpenGL, perspective division step occurs only after it has been determined that the point indeed lies within the view volume. That is, perspective division occurs after clipping. This ordering is not necessary, but it's just how OpenGL has done it historically.

Addendum: more on projective transformations

Projective transformation of a plane

When we use the term "polygon" we have in mind a set of vertices that all lie in a plane. It should be intuitive that a polygon remains a polygon under a modelview transformation since this transformation only involves rotation, translation, and scaling. It is less intuitive, however, whether our projective transformation also maps polygons to polygons (planes to planes). So, let us show why it does.

Consider any 4×4 invertible matrix $\mathbf{M}$ and consider the equation of a plane in $\Re^3$,

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

which can be written as

$$[a, b, c, d] \ [x, y, z, 1]^T = 0.$$

We are assuming the matrix $\mathbf{M}$ is invertible, so we can insert $\mathbf{M}^{-1}\mathbf{M}$ as follows:

$$[a, b, c, d] \ \mathbf{M}^{-1}\mathbf{M} \ [x, y, z, 1]^T = 0.$$

Define

$$[w'x', w'y', w'z', w'] = \mathbf{M} \ [x, y, z, 1]^T$$

where we have assumed $w' \neq 0$. Similarly, define

$$[a', b', c', d'] \equiv [a, b, c, d] \ \mathbf{M}^{-1}.$$

Then,

$$[a', b', c', d'] \ [w'x', w'y', w'z', w']^T = 0$$

which is the equation of a plane in (x', y', z'). Thus, $\mathbf{M}$ maps planes to planes.

Projective transformation of surface normal

What about the normal to a plane? Does a projective transformation map the normal of a plane to the normal of the transformed plane? It turns out that it does *not*. Why not?

Consider projective transformation matrix $\mathbf{M}$. Let $\mathbf{e} = (e_x, e_y, e_z)$ be a vector that is perpendicular to the surface normal vector $\mathbf{n} = (n_x, n_y, n_z)$. For example, $\mathbf{e}$ could be defined by some edge of the polygon, i.e. the vector difference $\mathbf{p}_1 - \mathbf{p}_0$ of two vertices . By definition of "surface normal", the vectors $\mathbf{n}$ and $\mathbf{e}$ must satisfy:

$$n_x e_x + n_y e_y + n_z e_z = \mathbf{n}^T \mathbf{e} = 0.$$

Write $\mathbf{e}$ and $\mathbf{n}$ as direction vectors $(e_x, e_y, e_z, 0)$ and $(n_x, n_y, n_z, 0)$. When the scene is transformed to clip coordinates, the normal $\mathbf{n}$ and vector $\mathbf{e}$ are transformed to $\mathbf{M}\ \mathbf{n}$ and $\mathbf{M}\ \mathbf{e}$, respectively. Note there is no reason why the projective transformation *should* preserve angles (e.g. 90 degree angles) and so there is no reason why $\mathbf{Mn}$ should remain perpendicular to $\mathbf{Me}$. So there is no obvious reason why $\mathbf{Mn}$ *should* be the normal to the corresponding polygon in the clip coordinates.

What *is* the normal vector to the polygon in clip coordinates? The normal must be perpendicular to any transformed edge $\mathbf{Me}$, so

$$0 = \mathbf{n}^T \mathbf{e} = \mathbf{n}^T \mathbf{M}^{-1} \mathbf{M} \mathbf{e} = ((\mathbf{M}^{-1})^T \mathbf{n})^T (\mathbf{M}\mathbf{e})$$

Thus, the vector $(\mathbf{M}^{-1})^T \mathbf{n}$ is perpendicular to all the transformed edges $\mathbf{M}\ \mathbf{e}$ of the polygon. Thus $(\mathbf{M}^{-1})^T \mathbf{n}$ must be the normal to the polygon in clip coordinates. Note that in general

$$(\mathbf{M}^{-1})^T \mathbf{n} \neq \mathbf{M}\mathbf{n}$$

and indeed you need special circumstances for this to hold e.g. $\mathbf{M}$ could be a rotation matrix.

Appendix

We first take a point (x, y, f_0) that lies on the near plane. Then,

$$\begin{bmatrix} f_0 & 0 & 0 & 0 \\ 0 & f_0 & 0 & 0 \\ 0 & 0 & \alpha & \beta \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ f_0 \\ 1 \end{bmatrix} = \begin{bmatrix} f_0 x \\ f_0 y \\ f_0\, \alpha + \beta \\ f_0 \end{bmatrix} = \begin{bmatrix} x \\ y \\ \alpha + \frac{\beta}{f_0} \\ 1 \end{bmatrix}$$

Since we require the transformation to preserve the z values for points on the f_0 plane, we must have:

$$\alpha + \frac{\beta}{f_0} = f_0.$$

Note that the x and y values will be preserved for these points!

What about points on the $z = f_1$ plane?

$$\begin{bmatrix} f_0 & 0 & 0 & 0 \\ 0 & f_0 & 0 & 0 \\ 0 & 0 & \alpha & \beta \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ f_1 \\ 1 \end{bmatrix} = \begin{bmatrix} f_0 x \\ f_0 y \\ f_1\, \alpha + \beta \\ f_1 \end{bmatrix} = \begin{bmatrix} \frac{f_0}{f_1} x \\ \frac{f_0}{f_1} y \\ \alpha + \frac{\beta}{f_1} \\ 1 \end{bmatrix}$$

Unlike for $z = f_0$ plane, points on the $z = f_1$ plane undergo a change in their (x, y) values. In particular, since $\frac{f_0}{f_1} < 1$ the values of (x, y) are scaled down.[4]

Also notice that since we are also requiring that these points stay in the $z = f_1$ plane, we have

$$\alpha + \frac{\beta}{f_1} = f_1.$$

The two cases ($z = f_0, f_1$) give us two linear equations in two unknowns (α and β). Solving, we get $\alpha = f_0 + f_1$ and $\beta = -f_0 f_1$, and this defines the following transformation:

$$\begin{bmatrix} f_0 & 0 & 0 & 0 \\ 0 & f_0 & 0 & 0 \\ 0 & 0 & f_0 + f_1 & -f_0 f_1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Verify for yourself that it maps points on the plane $z = f_0$ to points on $z = f_0$, and that it maps points on $z = f_1$ to $z = f_1$.

Lecture 6

Last lecture I introduced the view volume and mentioned how it is used to limit the set of points, lines, objects, etc that need to be drawn in an image. We might have a very complicated model of a scene, for example, an architectural model of many buildings, each of which contain many rooms.

[4]This reflects the familiar projective phenomenon that objects that are further away from the camera appear smaller in the image.

But at a given time, the view volume only requires us to consider a small part of the scene. In the next few lectures, we will discuss methods for discarding parts of the scene that we know won't appear in the image. These methods are called *clipping* or *culling*. (The two terms are sometimes used interchangably but there are distinctions which I'll mention as we go.)

Line Clipping

If we just want to know if a vertex lies in the view volume or not, then we apply the inequalities that we discussed last lecture. There's no more to be said. Today we'll look at the problem of clipping a *line*. This problem is more difficult because we need to do more than understand whether the endpoints are in the view volume; we have all the points along the line as well to consider.

There are three cases to think about this. One is if both end points of the line belong to the view volume. In this case, the whole line does too. A second case if the entire line lies outside the view volume. In this case, the line can be discarded or *culled*. The third case is that part of the line lies outside the view volume and part lies in it. In this case, we only want to keep the part that lies within. We *clip* off the parts of the line that lie outside the view volume. Imagine we take scissors and cut off parts of the line that poke outside the view volume. In the discussion below, I won't bother using different terminology for the second and third cases. I'll refer to them both as *clipping*.

As I mentioned at the end of last lecture, vertex clipping is done in clip coordinates (wx, wy, wz, w) by requiring that all the inequalities hold:

$$w > 0$$

$$-w \leq wx \leq w$$

$$-w \leq wy \leq w$$

$$-w \leq wz \leq w.$$

Indeed the same is true for line clipping and for clipping of other objects as well (triangle, etc). However, for the purposes of simplicity, we will follow tradition and *explain* a basic clipping algorithm in normalized device coordinates (x, y, z), so

$$-1 \leq x \leq 1$$

$$-1 \leq y \leq 1$$

$$-1 \leq z \leq 1.$$

Cohen-Sutherland

Several line clipping algorithms have been proposed over the years. I'll present one of the earliest which is from 1965 and was invented by Cohen and Sutherland[5]

[5] Ivan Sutherland, in particular, has made many enormous contributions to computer science http://en.wikipedia.org/wiki/Ivan_Sutherland.

We also start by considering the 2D case since it captures the main ideas and it is easier to illustrate. Suppose we have a line segment that is defined by two endpoints (x_0, y_0) and (x_1, y_1). We only want to draw the part of the line segment that lies within the window $[-1, 1] \times [-1, 1]$. How can we "clip" the line to this window? One idea is to check whether the line intersects each of the four edges of the square. A *parametric representation* of the line segment is

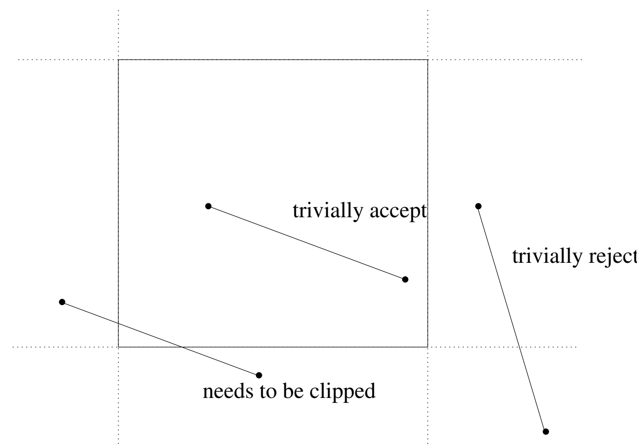
$$x(t) = x_0 + t(x_1 - x_0)$$

$$y(t) = y_0 + t(y_1 - y_0),$$

where $t \in [0, 1]$. To see if the line segment intersects an edge of the square, we could set $x(t)$ and $y(t)$ to the values at the edges of the square, e.g. we could check the intersection with the edge $x = -1$ (assuming $x_0 \neq x_1$), and solve for t which requires a division. If we find that $t \in [0, 1]$, then the line segment would intersect the edge and hence would need to be clipped. To find the clipping point, we substitute this value of t back into $(x(t), y(t))$ which requires a multiplication.

Although having to compute several divisions or multiplications per line segment might not seem like a problem (since computers these days are very fast), it could slow us down unnecessarily if we have many line segments. Let's look at a classical method (by Cohen and Sutherland) for how to clip line segments in a more efficient way.

To motivate, note that if x_0 and x_1 are both greater than 1 or both less than -1, or if y_0 and y_1 are both greater than 1 or less than -1 then the line segment from (x_0,y_0) to (x_1,y_1) cannot intersect the square. This is called *trivially rejecting* the line segment. It is possible to *trivially accept* the line segment as well: if x_0 and x_1 are both in $[-1, 1]$ and y_0 and y_1 are both in $[-1, 1]$, the line segment lies entirely within the square and does not need to be clipped.



Cohen and Sutherland's method is a way to organize the test for trivial reject/accept. Let (x, y) be a point in the plane, namely an endpoint of a line segment. Define a 4 bit binary string - called an "outcode" - to represent (x, y) relative to the clipping square.

$$
\begin{aligned}
b_3 &:= (y > 1) \\
b_2 &:= (y < -1) \\
b_1 &:= (x > 1) \\
b_0 &:= (x < -1)
\end{aligned}
$$

Note the outcode $b_3\ b_2\ b_1\ b_0$ denotes the position in the order "top, bottom, right, left."

The 3×3 grid of regions showed in the sketch on the previous page have a corresponding 3×3 grid of outcodes:

1001	1000	1010
0001	0000	0010
0101	0100	0110

We "trivially accept" a line segment if the *bitwise or* of the outcodes of the endpoints (x_0, y_0) and (x_1, y_1) is 0000. We "trivially reject" a line segment if the *bitwise and* of the two outcodes of the two points is other than 0000.

If we can neither trivially reject nor trivially accept the line segment, then we need to carry out further computation to determine whether the line overlaps the square and, if so, to clip it to the square. We sequentially clip the line segment according to the bits $b_3\ b_2\ b_1\ b_0$. If we cannot trivially reject or accept, then at least one of these bits must have the value 0 for one endpoint and 1 for the other endpoint, which means that the line segment crosses the boundary edge that corresponds to that bit. We proceed arbitrarily from b_3 to b_0 and examine the bits, clipping the line segment to the edge for that bit. Each such clip requires computing t and the new (x, y) endpoint.

The example below on the left shows that four clips might be needed using this method. The initial outcodes are 1001 and 0110. At each clip, exactly one of the 1 bits is flipped to a 0. For the example on the right, the outcodes are 1010 for the upper and 0100 for the lower vertex. After the first clip, the outcode of the upper vertex switches from 1010 to 0010. After the second clip, the bottom vertex switches from 0100 to 0010 (note the flip in the b_1 bit). Then, the line segment can be trivially rejected since the b_1 bit is 1 for both endpoints.

The Cohen Sutherland method can be applied in 3D to clip a line segment to a 3D view volume in normalized projection coordinates. One uses six bits instead of four, i.e. one needs the two additional bits

$$\begin{aligned} b_5 &:= (z > 1) \\ b_4 &:= (z < -1) \end{aligned}$$

A few technical issues are worth noting. First, recall that in OpenGL one doesn't clip in normalized device coordinates but rather one clips in clip coordinates (wx, wy, wz, w). If $w > 0$, then assigning the bitcodes is done as follows:

$$\begin{aligned} b_5 &:= (wz > w) \\ b_4 &:= (wz < -w) \\ b_3 &:= (wy > w) \\ b_2 &:= (wy < -w) \\ b_1 &:= (wx > w) \\ b_0 &:= (wx < -w) \end{aligned}$$

What about if $w < 0$? Think about it, and then see the Exercises.

Another issue arises when using clip coordinates for clipping. While the tests for outcodes are straightforward, it is not obvious how to do the clipping itself. Recall the parametric equation of the line from page 2 which you need to solve on page 2. In clipping coordinates, you don't have the x or y values but rather you have the wx or wy values. To solve for the intersection with the boundary of the view volume, you might think that you need to divide by the w coordinate, which essentially takes you to normalized device coordinates. It turns out you don't need to do this division. You can find the intersection in clip coordinates. Think about it, and see the Exercises.

Windows and viewports

We are almost done with our transformations. The last step concerns mapping to pixels on the screen i.e. in pixel space. That is, we need to talk about *screen coordinates*, also known as *display coordinates*.

A few points to mention before we discuss these coordinates. First, the display coordinates of a point are computed by mapping the normalized device coordinates (x, y, z) to a pixel position. However, only the (x, y) components are used in this mapping. The z component matters, as we'll see in the next few lectures when we discuss which points are visible. But for the discussion here, the z values don't matter.

Second, there is some inevitable confusion about terminology here, so heads up! The term *window* will be used in a number of different ways. The most familiar usage for you is the rectangular area of the screen where your image will be drawn. We'll call this the *screen window* or *display window*. This is the same usage as you are familiar with from other applications: a web browser, an X terminal, MS Office, etc. You can resize this window, drag it around, etc. Because this is most familar, let's consider it first.

The OpenGL glut commands

```
glutInitWindowPosition(x,y)
glutInitWindowSize(width,height)
```

specify this display window where your image will be drawn.[6] Typically, you can drag this window around. One often defines code to allow a user to resize the window (using the mouse), which is why the function has `Init` as part of its name. Note that this is not core OpenGL, but rather it is `glut`.

Note that a "display window" is different from the "viewing window" which we discussed in the previous two lectures, namely a window defined by `(left, right, bottom, top)` boundaries in the plane z = `-near`. We have to use the term window in both cases and keep in mind they are not the same thing.

OpenGL distinguishes a display window from a viewport. A *viewport* is a 2D region *within* a display window. At any given time, OpenGL draws images within the current viewport. You can define a viewport using

```
glViewport(left,bottom,width,height).
```

The parameters `left` and `bottom` are offsets relative to the bottom left corner of the display window. Of course, they are different values from the parameters with the same name that are used to define the display window in a `glFrustum` call.

The default for the viewport (i.e. no call) is the entire display window and is equivalent to

```
glViewport(0,0,width,height).
```

where width and height are the parameters of the display window.

Why would you want a viewport that is different from the default? You might want to define multiple non-overlapping viewports within the display window. For example each viewport might show the same scene from different viewer positions. That is what we will do in Assignment 1.

To map the normalized device coordinates coordinates to pixel coordinates on the display, OpenGL performs a *window-to-viewport* transformation. I emphasize: the term "window" here does *not* refer to the display window. Rather, it refers to the $(x, y) = [-1, 1] \times [-1, 1]$ range of the normalized device coordinates, which corresponded to the viewer's "window" on the 3D world as specified by the `glFrustum()` call.) The *window-to-viewport* mapping involves a scaling and translation. See the Exercises.

Scan conversion (rasterization)

By mapping to a viewport (or more generally, to a display window), we have finally arrived at the pixel representation. There will be alot to say about pixels later in the course – and we will need to use a more general term *fragments* at that time. For now, let's just stick to pixels and deal with a few basic problems.

[6] This is only a recommendation. You have no guarentee that your window ends up exactly there.

One basic prolem is how to represent a continuous position or set of positions – a point, a line segment, a curve – on a rectangular pixel grid. This problem is known as *rasterization* or *scan conversion*.

The term "scan" comes from the physical mechanism by which old CRT (cathode ray tube) televisions and computer screens used to display an image, namely they scanned the image from left to right, one row at a time. The term "scan conversion" should be interpreted as converting a continuous (or floating point) description of object image so that it is defined on a pixel grid (raster) – so that it can be scanned.

Scan converting a line

Consider a line segment joining two positions (x_0, y_0) and (x_1, y_1) in the viewport. The slope of the line is $m = \frac{y_1 - y_0}{x_1 - x_0}$. The line may be represented as

$$y = y_0 + m(x - x_0).$$

If the absolute value of the slope is less than 1, then we draw the line by sampling one pixel per column (one pixel per x value). Although the equation of the line (and in particular, the slope) doesn't change if we swap the two points, let's just assume we have labelled the two points such that $x_0 \leq x_1$. We draw the line segment as follows.

```
m = (y1 - y0)/(x1 - x0)
y = y0

for x = round(x0) to round(x1)
   writepixel(x,  Round(y), rgbValue)
   y = y + m
```

Notice that this method still requires a floating point operation for each y assignment since `m` is a float. In fact, there is a more clever way to draw the line which do not require floating point operations, which takes advantage of the fact that the x and y coordinates are integers. This algorithm[7] was introduced by Bresenham in the mid 1960s.

The above algorithm assumes that we write one pixel on the line for each x value. This makes sense if $|m| < 1$. However, if $|m| > 1$ then it can lead to gaps where no pixel is written in some rows. For example, if $m = 4$ then we would only write a pixel every fourth row! We can avoid these gaps by changing the algorithm so that we loop over the y variable. We assume (or relabel the points so) that $y_0 \leq y_1$. The line can be written as $x = (y - y_0)/m + x_0$. It is easy to see that the line may be drawn as follows:

```
x = x0
y = y0;
for y = y0 to y1 {
```

[7]I am not covering the details because the tricks there (although clever and subtle) are not so interesting when you first see them and they don't generalize to anything else we are doing in the course (and take some time to explain properly). If you are interested, see `http://www.cim.mcgill.ca/~langer/557/Bresenham.pdf`

```
    writepixel(Round(x),  y).   //  RGB color to be determined elsewhere...
    x  = x  +  1/m;
}
```

Scan converting a triangle (more generally, a polygon)

Suppose we want to scan convert a simple geometric shape such as a triangle. Obviously we can scan convert each of the edges using the method described above. But suppose we want to fill in the interior of the shape too. In the case of a triangle, this is relatively simple to do. Below is a sketch of the algorithm. For more complicated polygons, the two steps within the loop require a bit of work to do properly. For example, a data structure needs to be used that keeps a list of the edges, sorted by position with the current row. I'm omitting the details for brevity's sake. Please see the sketches drawn in the slides so you have a basic understanding of what is required here. We will return to this problem later in the course when we will say *much* more about what it means to fill a polygon.

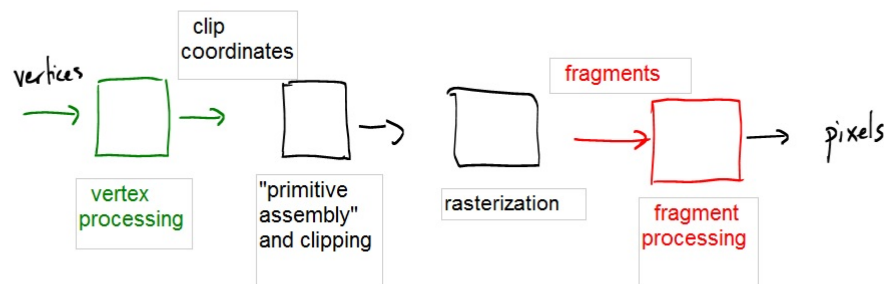
```
examine vertices of the polygon and find ymin and ymax
//  if they don't have integer coordinates, then round them

for y = ymin to ymax
    compute intersection of polygon edges with row y
    fill in between adjacent pairs of edge intersections
```

Lecture 7

At the begin of this lecture I discuss the standard graphics pipeline. This finishes up Part 1 of the course, covered in lectures 1-6, and also provides a segue to Part 2 which concerns issues of visibility and scene and object modelling.

Graphics pipeline

We have spent the last several lectures discussing how to map positions of object points to positions of pixels in the image. This mapping involved a sequence of matrix transformations, as well as other operations such as clipping, rasterization, depth testing, and many more. Here I will sketch out the basic stages of the pipeline.



Vertex processing

Here each vertex is processed individually. For now, you should think of the vertex processor as transforming the coordinates of the vertex from object to clip coordinates, since this is what we have been mostly talking about the past several lectures. It turns out there is more to it that that, for example, the color of the vertex and other properties of a vertex can be determined here as well.

In classical OpenGL, there are a limited (fixed) set of functions for the programmer to use at this stage – basically the transformations covered in lectures 1-6. In modern OpenGL, there is much more flexibility. The programmer implements this stage explicitly by writing programs called *vertex shaders*. This includes explicitly writing the transformations we have seen, as well as other interesting transformations on vertices and surface normals.

For example, suppose you wanted to make an animation of a water surface with complicated waves. Or suppose you wanted to animate a person's face as the person speaks. In classic OpenGL, the changes in the mesh would be done on the CPU. For example, you would define data structures for representing your surface object (say a "mesh" of triangles), and you would write code for the CPU to manipulate the triangles of this object so that they have a certain motion over time (and maybe changing colors).

In modern OpenGL, these vertex manipulations would be done on the graphics card (the GPU). You would write instructions for manipulating the surface. THis would be a program called a vertex shader. The advantage of using the GPU is that it has many parallel processors. They all run the same instructions (the code of the vertex shader) in parallel over multiple vertices. The CPU is then free to do other things.

Primitive Assembly and Clipping

In order to clip a line segment or triangle or polygon, these primitive must be created. Loosely, think of an object of type 'line segment' or type 'triangle'. The object would have pointers to its constituent vertices.

After the vertices have been transformed individually to clip coordinates by the vertex processor (shader), the "geometric primitives" – e.g. line segment objects, or triangle objects – must then be assembled. Think Cohen-Sutherland here: you can't run the clipping algorithm unless the two vertices are considered as a line segment. (Note that a different clipping algorithm is used to clip a line vs a triangle vs a general polygon. I omitted those details in the clipping lecture.)

As a concrete example, think of the following. The `glBegin` specifies what the 'primitive assembler' need to assemble (a line) and which vertices to use. The `glVertex()` calls send the vertices to the vertex shader who maps them to clip coordinates.

```
glBegin(GL_LINES)
   glVertex()
   glVertex()
glEnd()
```

Both in classical OpenGL and modern OpenGL, these two stages – primitive assembly and clipping – are done "under the hood", and indeed are sometimes considered as grouped into one stage. As a programmer, you don't have to think about it. You define the geometric primitives and you let the graphics card take care of assembling them at this stage.

Rasterization

Here we say the processing is done by a "rasterizer". For each primitive (line, triangle, etc), the rasterizer defines a set of potential pixels that this primitive might contribute to. (I say "potential" because the primitive is not necessarily visible in the image.) These potential pixels generally have more information associated with them than just (x,y) values, for example, they may have depth and color information. Such potential pixels are called *fragments*. So the output of the rasterizer is a set of fragments.

Both in classical OpenGL and modern OpenGL, this stage is also hidden. The programmer has no access to the rasterizer.

Fragment Processing

The next stage is to process the fragments. The job the "fragment processor" is a big one. Given all the fragments, the fragment processor computes the image RGB value at each pixel. This involves not just deciding on the intensities. It also involves deciding which surfaces are visible at each pixel.

In classical OpenGL, there are a limited (fixed) number of functions for the programmer to use at this stage. In modern OpenGL, there is much more flexibility and the programmer implements this stage explicitly by writing programs called *fragment shaders*. Much of the second half of the course will deal with this stage. We won't be writing fragment shaders, but we will be discussing the problems that these shaders solve and the classical algorithms for solving them.

We are now ready to continue to part 2 of the course which concerns visibility and geometry modelling. We begin with the hidden surface removal problem.

Hidden surface removal

Suppose we have a view volume that contains a set of polygons, for example, triangles, that are pieced together to make up an object. We would like to know which points on these polygons are visible in the image. We will examine several methods. Some of these methods examine each of the objects of the scene and decide, *for each object*, at which pixels are the objects visible. These are called "object order" methods. Others examine, *for each pixel*, which of the objects can be visible at that pixel. These are called "image order" methods.

Let's begin by looking at a few object order methods that are used by OpenGL. I'll then take a brief step back and discuss how these methods fit into the "OpenGL pipeline". After that, I'll return to the hidden surface problem and discuss some methods that are *not* used by OpenGL but that are heavily used in other graphics systems. These can be object or image order.

Back face culling

In many cases a polygon is part of a surface that bounds a solid object, e.g. the square faces that bound a cube. We only want to draw the "front" part of the object, namely the part facing towards the camera. Thus for each polygon we would like to test if it faces toward the eye or away from the eye and, in the latter case, we discard the polygon rather than draw it. Discarding back faces is called *back face culling*[8].

In the above description, we relied on having a solid object to define front and back faces. But back face culling is more general than this. For any polygon, we can define the front and back face of the polygon. Back face culling just means that we only draw the polygon if the viewer is on the side of the polygon that corresponds to the front face. This is a more general notion since it applies to polygons that don't belong to solid objects.

In OpenGL, the default for the front face is that the vertices are ordered in a counter-clockwise direction as seen by the viewer. That is, given a polygon and a viewer position, the viewer sees the front face if the viewer sees the vertices ordered counter clockwise and the viewer sees the back face if the viewer sees the vertices ordered clockwise.

[ASIDE: OpenGL has commands that can reverse this definition. The default front face is counterclockwise, but `glFrontFace(GL_CW)` and `glFrontFace(GL_CCW)` can be used to toggle this definition. If you want both sides of a polygon to be rendered, then there is a state variable `GL_FRONT_AND BACK` that you can set.]

Conceptually, the front face has an outward facing normal and the back face has an inward facing normal. However, these concepts of outward and inward being associated with front and back faces, respectively, is only in your head. It is not part of OpenGL. In OpenGL, one can define surface normals explicitly (and we will do so later in the course when we discuss shading) but these explicitly defined surface normals are not constrained to correspond to front faces, and the explicit surface normals have nothing to do with how back face culling is implemented. Back face culling is

[8] to "cull" basically means to "remove"

implemented only by examining whether a polygon's vertices are counterclockwise, as seen by the viewer.

Finally, we don't *have to* do back face culling to draw the scene correctly – at least, the solid objects. As long as we do hidden surface removal (see above), back faces of a solid object automatically will not appear in the image but rather will be hidden by front faces which are closer to the camera. The reason for doing back face culling is that it is a cheap way to speed up the computation.

Back face culling is a general technique and there is no single stage of the graphics pipeline where it has to occur. As shown in the slides, it can be done in eye coordinates: in this case the decision to cull or not cull a polygon depends on the sign of the dot product of two vectors: the vector from the eye to a vertex on the polygon, and the normal pointing out of the front face. If that dot product is positive then the polygon is back facing and can be culled; otherwise, it is kept (not culled).

In OpenGL, back face culling is done at the rasterization phase, that is, after the polygon has passed the clipping stage and so we know the polygon lies in the view frustum. The vertices of the polygon are now represented in normalized view coordinates, not eye coordinates. To decide if a surface polygon is a front or back face, the rasterizer checks if the outward normal vector of the polygon *in normalized device coordinates* has a negative or positive z component, respectively. (Recall that viewing in normalized device coordinates is orthographic. To get the sign of the test correct, recall that normalized device coordinates are left handed coordinates i.e. $\hat{\mathbf{z}}$ points away from the camera.) If the polygon is culled, this means that the rasterizer doesn't generate any fragments for that polygon.

Depth buffer method

Let's now turn to a second object order method for deciding which points on a polygon are visible in the image. Here we need to introduce the idea of a *buffer*. In computer graphics, a buffer is just a 2D matrix that holds values. The buffer holding the RGB values of the image[9] is referred to as the *frame buffer* or *image buffer*. Many graphics systems also have a buffer that holds the z values (depth) of the surface point that is visible at a pixel. This is known as the *z buffer* or *depth buffer*. In OpenGL, the z buffer holds values in $[0, 1]$ where 0 corresponds to the near plane and 1 corresponds to the far plane.

Recall I mentioned last lecture that there is a window to viewport mapping that takes the xy in $[-1, 1] \times [-1, 1]$ of normalized device coordinates and maps it to viewport coordinates. We can tack on the depth values to this transformation and map z from $[-1, 1]$ to some range of values. In OpenGL, the z mapping is from $[-1, 1]$ to $[0, 1]$. So in screen coordinates, depth has values in $[0, 1]$. This mapping is achieved in the expected way, namely a translation and a scaling. (You now should see why we used f_0 and f_1 back in lecture 5.) We referred to the xy coordinates as viewport coordinates last lecture. We also sometimes call them *screen coordinates*. With the depth buffer in mind, we can think of screen coordinates as having a depth too.

Here is the depth buffer algorithm for calculating visible surfaces. Note that it is an object order method: the polygons are in the outer loop and pixels are in the inner loop.

[9] Or RGBA – see later in the course

```
for each pixel (x,y)   // initialization
   z(x,y) = 1
   RGB(x,y) = background color
for each polygon
   for each pixel in the image projection of the polygon
        z := Ax + By + C
         // equation of polygon's plane *in screen coordinates*
        if z < z(x,y)
            compute RGB(x,y)
            z(x,y) :=  z
```

A few points to note. First, you might expect that floats are used for z values in order to make precise decisions about which surface is in front of which. This is not the case, however. Rather OpenGL uses fixed point depth representation, typically 24 bits (but sometimes 32). This can lead to errors when multiple objects fall into the same depth bin. However, 2^{24} is alot of bins so this is unlikely to happen (though see below).

To elaborate of why floating point isn't used: floating point allows you to represent extremely small numbers and extremely large numbers, with the bin size being roughly proportional to the size of the number. But that last condition makes no sense for representing depth values in graphics where the objects can be anywhere between the near and far plane. Moreover, you don't typically deal with extremely tiny or extremly large depths. So floating point makes little sense for depth. [And besides, fixed point arithmetic is generally faster.]

A second point to note is that we can now see why the choice of `near` and `far` is important. You might have been thinking up to now that it is best to choose your `near` to be very close to 0 and your `far` to be infinity. The problem with doing so is that you only get 2^{24} depth bins. If near is too close and far is too far, then "aliasing" can easily occur – namely two objects can mapped to the same depth bin for some (x,y). In that case, when the algorithm compares the depth of a polygon at that pixel to the value stored in the depth buffer, it can potentially make errors in deciding which surface is closest.

Finally, where in the pipeline is the depth buffer algorithm run? The simplest way to think about it for now is that it is done by the rasterizer, rather than the fragment shader. For each polygon, the rasterizer generates a set of fragments and compares the depth of each fragment to the current depth value at the associated pixel in the depth buffer. If the depth of the fragment is greater than the closest depth at that position, then the fragment is discarded.

Painter's algorithm (Depth sort)

Here I give an overview of a second object order method, which draws polygons in terms of *decreasing* distance i.e. draw farthest ones first. This is called the painter's algorithm because it is similar to what many painters do, namely paint closer objects *on top of* farther ones. The method goes like this:

1. Sort polygons according to each polygon's *furthest* z coordinate.

   However, in the case that two polygons have overlapping z range, we need to be careful about how we place them in the ordered list. This is discussed below and you should consult the slides for figures.

2. Draw the polygons in the order defined by the list, namely from back to front (farthest to nearest).

Suppose we have two polygons P and Q such that furthest vertex of Q is further than the further vertex of P. It could still happen that Q could cover up part of P or even all of P in the image – and in that case we do not want to draw Q before P. One thing to try is to swap the ordering of P and Q. This doesn't necessarily solve the problem, though since it could create problems with other polygons.

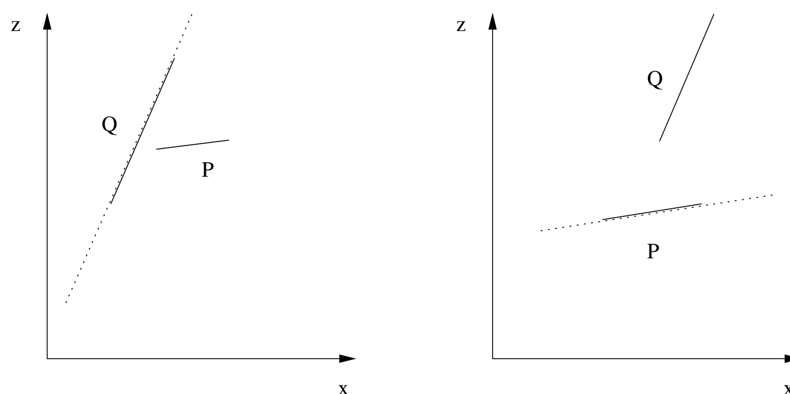
I am not presenting the full algorithm here, since there are alot of details and, in the end, the Painter's algorithm has flaws that we will address next lecture (when we discuss BST trees). But I still want to give you a flavour of the issues here, so let me say a bit more.

What are some conditions to check in order we can be sure its ok to draw Q before P ? If any of the following conditions is true, then we are ok.

- the z range of P and Q do not overlap, i.e. all of Q's vertices are further than all of P's vertices
- the x range of P and Q do not overlap
- the y range of P and Q do not overlap
- all vertices of Q are on the far side of P's plane (requires checking each vertex of Q against P's plane)
- all vertices of P are on the near side of Q's plane (requires checking each vertex of P against Q's plane)

The last two conditions are subtle. For both, we need to check whether a set of points on one polygon lies in front of or behind another polygon, in the sense of being in front of or behind the plane defined by that polygon. It is a simple exercise to do this, using basic linear algebra.

Note that these last two conditions are not the same. For example, below I show a 2D version of the conditions. On the left, all the vertices of P lie in front of Q's plane, whereas it is not true that all vertices of Q lie behind P's plane. On the right, all vertices of Q lie behind P's plane, whereas it is not the case that all vertices of P lie in front of Q's plane.



What if none of the five conditions holds? In class, I said that the solution is to cut (say) P's plane using Q. However, this is not guarenteed to work. For example, it doesn't necessarily solve the 'highway over bridge' problem of slide 33 in the lecture. e.g. Planes P and Q in that slide could be parallel in 3D and hence not intersect.

Ray casting method

Let's now turn to image order methods. The first image order method, called *ray casting*, is similar to the depth buffer method but it inverts the order of the two loops. It considers each pixel in the image window and shoots (casts) a corresponding ray in the z direction. It then computes the intersection of the ray with all the polygons in the scene and figures out which polygon is closest along the ray.

```
for each pixel (x,y)
   z_min = 1  #  initialized to max value i.e. max[-1,1]
   RGB(x,y) = backgroundRGB
   for each polygon
      if (x,y) lies in image projection of polygon // see item below
        z =  Ax + By + C
            // equation of polygon's plane in screen coordinates
        if z < z_min
           z_min := z
           pixel(x,y).poly = polygon  //  point to polygon
   compute RGB(x,y)   // more later
```

A few observations:

- The ray casting method doesn't require a z buffer, since we are dealing with one pixel at a time. The scalar variable `z_min` plays the role of the z buffer for that single pixel.
- We choose the color of a pixel (x, y) only once in this method. We do so by maintaining a pointer to the polygon that is closest. We only choose the color once we have examined all polygons.
- We need a method to check if a pixel (x,y) lies with the image projection of a polygon. This is known as the "point in polygon" test. I am not planning to present algorithms for solving this problem. If you are interested, please see `http://erich.realtimerendering.com/ptinpoly/`.

The ray casting method implicitly assumed that the scene either was being viewed under orthographic projection, or it already had been transformed into screen coordinates (via normalized device coordinates and a transform to viewport). That is, we assumed z was a function of image coordinates (x, y) of the form $z = Ax + By + C$. In particular, we were assumed we were ray casting in the z direction.

The idea of ray casting is more general than this. For example, suppose we have a polygon that lies in a plane

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

Consider two points (x_0, y_0, z_0) and (x_1, y_1, z_1). Does the ray that starts at (x_0, y_0, z_0) and that passes through (x_1, y_1, z_1) intersect that polygon? For example, (x_0, y_0, z_0) might be a viewer position and (x_1, y_1, z_1) might be a point in the projection plane.

We do this ray cast in two steps. The first step is to intersect the ray with the plane. We substitute three parametric equations of the ray

$$(x(t), y(t), z(t)) = (x_0, y_0, z_0) + ((x_1, y_1, z_1) - (x_0, y_0, z_0))\ t$$

into the equation of the plane and solve for t. This gives us the point of intersection of the ray with the plane. (Note if $t < 0$ then the ray doesn't intersect the plane, and if the solution gives a division by zero then the ray is parallel to the plane and either doesn't intersect it or lies within it.)

The second step is to decide if this intersection point lies within the polygon. At first glance, this seems a bit awkward to decide since we might be tempted to write the vertices and intersection points in terms of a 2D coordinate system for this plane. A much simpler approach is to project the intersection point and polygon orthographically onto one of the three canonical planes i.e. set either $x = 0$, or $y = 0$, or $z = 0$, and then solve the "point in polygon" problem within that canonical plane. Note that the point is in the polygon in the original plane if and only if its projection is in the projection of the polygon in the canonical plane. The only warning is that you need to ensure that the projection doesn't collapse the polygon into a line.[10]

We can do ray casting with quadric surfaces, just as we did with polygons. We substitute the three parametric equations of the ray $\{(x(t), y(t), z(t)) : t > 0\}$ into the equation for the quadric surface $\mathbf{x^T Q x = 0}$. This yields a second order "quadratic equation" in the variable t, namely

$$\alpha t^2 + \beta t + \gamma = 0.$$

This equation yields two solutions for t, which may be either real or complex. If this equation has no real solution, then the ray does not intersect the quadric. If it has two identical solutions, then the ray is tangent to the surface. If the equation has two real solutions but only one is positive[11], then this one positive solution defines the point on the quadric that is visible along the ray. If the equation has two positive real roots, then the smaller one is used. For example, the quadric might be a sphere, and the ray might pass twice through the sphere – once on the way in, and once on the way out. We would choose the first of these two solutions since we are interested in the "front face".

[10]For example, if the original polygon lay in the $z = 5$ plane, and we project in the x or y direction, then the projected polygon would collapse to a line. This is no good. So, instead for this polygon we must project in z direction.

[11]If a solution is negative, then this solution does not lie on the ray, since the ray goes in one direction away from the point in the positive t direction.

Lecture 8

Last lecture we looked at several methods for solving the hidden surface removal problem. Many such methods boil down to the problem of finding, for each pixel, what is the closest surface in the view volume that is 'visible' at that pixel. Mathematically this amounts to defining a ray from the viewer through that pixel out into the scene and finding the first intersection.

The ray casting approach I presented last class was very inefficient since, for each pixel, it blinding tested every polygon in the scene. We would like to solve this problem more efficiently. Today we'll look at three approaches which use clever *data structures.* These methods were developed in the early 1980s.

We'll begin with two methods that use spatial partitions. The first method is a generalization of the painter's algorithm from last class.

Binary Space Partition (BSP) tree

The painters algorithm sorted polygon surfaces by their furthest vertices, and had to deal with several special cases which arose when the z extents of the surfaces overlapped.

One limitation of the painter's algorithm is that if the camera is moving, then we will need to recompute the depth ordering for each camera position and orientation. To see this, consider an extreme case where the camera moves in a circle, and the view direction always points to the center of the circle!

An alternative way to solve the painter's problem is to organize the spatial relationships between the polygons in a way that it the ordering of the polygons from back to front can be determined quickly for any viewing position and direction. The basic idea is this[12]: consider an arbitary plane in the scene, if the camera is on one side of this plane, then any polygons that are on the same side of that plane as the camera cannot be occluded (blocked) by polygons that are on the opposite side of the plane. Thus, if we draw polygons on the far side of the plane first, then these far polygons will never occlude polygons that are on the near side of the plane. That is, the near polygons are drawn after the far ones, just like in the painter algorithm.

The method is reminiscent of binary search trees that you saw in COMP 250 and 251. Recall how you build a binary search tree. You take some element from your collection, and then partition the set of remaining elements into two sets. The selected element is placed at the root of the tree. The elements less than the selected one would go in the left subtree (recursively contructed) and the greater than elements go in the right subtree (recursively constructed).

There are many ways to use a binary search tree. You can search for a particular element. Or you can print out all the elements in order. Or you can print them out in backwards order. (I'm sure you know how to do this so I won't review it.) The analogy to the BSP data structure that I'll describe next is that you want to examine the elements in reverse order. The order in our case is the distance to the surface.

The binary space partition tree which represents a partition of 3D space. There are two aspects here. One is how to construct the tree. The other is how to use it to solve the visiblity problem.

[12][Fuchs et al, 1980]

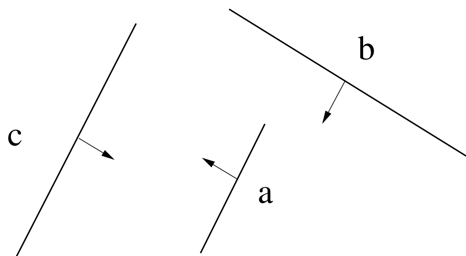
Constructing the BSP tree

Suppose we have a list of polygons. Each polygon is assumed to have a surface normal, so the normal points out of the surface and defines the front side of the polygon. We construct a *binary space partition tree* (BSP tree) as follows. Note that this is a pre-processing step. There is no viewer defined.

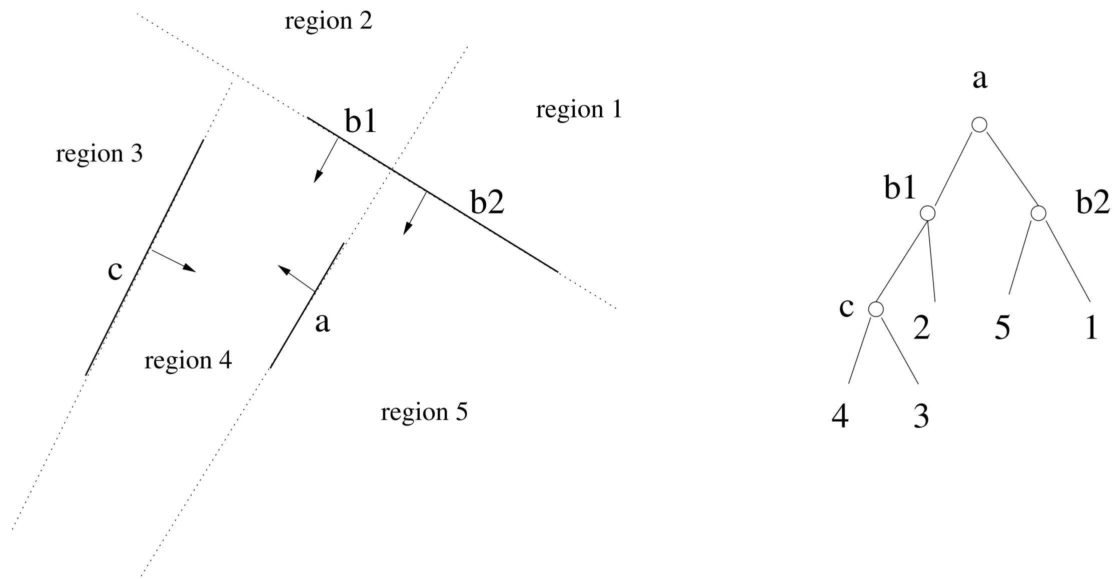
```
makeBSPtree(list of polygons){
   if list is empty
      return(NULL)
   else {
      select and remove a polygon P from list
      backlist := NULL;
      frontlist := NULL;
      for each polygon Q in list
         if all vertices of Q are in front of plane of P
            add Q to frontlist
         else if all vertices of Q are behind plane of P
            add Q to backlist
         else  //  plane P splits Q
            split Q into two polygons and add them to
            frontlist and backlist, respectively
       return   combine( makeBSPtree(frontlist),  P,
                         makeBSPtree(backlist) )
   }
}
```

The relations "front" and "back" refer to the front and back face of polygon P, as defined last class. They do not depend on the viewer's position, but rather only depend on the ordering of the vertices. *We use the convention that the polygons Q in front of plane P belong to the left child and the polygons behind plane P belong to the right child.*

Here is an example. What is the 2D binary space partition tree (BSP tree) defined by the following set of line segments ? Let's suppose that when constructing the BSP tree, we choose edges in alphabetical order (a,b,c).



Here is the solution.



Hidden surface removal using the BSP tree

How do you use a BSP tree to solve the visible surface problem ? As I mentioned earlier, the idea is similar to the painter's method: draw from back to front, relative to the viewpoint. We start at the root of the BSP tree and draw the polygons that lie on the far side of the root polygon's plane, then draw the root, then draw the polygons that lie on the near side of the root polygon's plane. Don't confuse "front" and "back" face with near and far sides of the polygon. Only the latter depend on the viewer position.

```
displayBSPtree( tree, viewer){
   if (tree != NULL)
      if (viewer is on the front side of plane defined by root){
         displayBSPtree(backchild, viewer)
         displayPolygon(root)
         displayBSPtree(frontchild, viewer)
      }
      else{  //  viewer is behind the root note
         displayBSPtree(frontchild,viewer)
         displayPolygon(root)  //  back faced culled,
                                   so not necesssary
         displayBSPtree(backchild,viewer)
      }
}
```

In the above example, suppose the camera was in region 5. We would first examine the root node `a` and see that the viewer is on the back side of polygon `a`, so it needs to draw all polygons on the opposite side (the front side, which is the left subtree. Next, examine the root of that left subtree `b1` and note that the view lies in on the front side of `b1` and hence needs to draw all polygons behind `b1` first, i.e. the right subtree, which is empty in this case. Then draw `b1`, and then draw

the left subtree of b1. This brings us to node c. The viewer is on the front side of c. We thus need to draw its right subtree (region 3 – empty), draw c., and then draw c's left subtree (region 4 – empty).

Now we traverse back up the tree, draw a, and then examine the right subtree of a. We examine b2, draw its right subtree (region 1 – empty), then draw b2, then draw its left subtree (region 5 – empty). Thus, the order in which we draw the polygons is b1, c, a, b2. Similarly, the order in which we traverse the regions is 2,3,4,1,5.

Observations:

- In COMP 250, you learned about in-order, pre-order, and post-order tree traversals. But here the traversal need not be any one of these. The traversal order depends on where (which region) the viewer is.
- We hope depth of the tree grows roughly like $\log n$, where n is the number of polygons. There is no guarentee on this, though.
- You are free to cull back faces if you want to, or not.
- The great advantage of the BSP method over the depth buffer method comes if you will want to make lots of pictures of a scene from many different viewpoints, i.e. move the camera through the scene (translate and rotate). You can use the same BSP tree over and over again, as long as the polygons in the scene aren't moving from frame to frame. If there is motion in the scene, then this is not a good method to use, however. You would be better off using the depth buffer method (like OpenGL).
- We did not discuss the special case that the viewer lies exactly on the plane defined by a node. This special case occurs, then we could move the viewer by a tiny amount off the polygon.

Uniform spatial partitions

The next method also partitions the scene into regions, but it uses a regular grid which is independent of the particular scene. Consider a cube that bounds the scene. Suppose we partition this bounding cube into disjoint volumes, namely each of the x,y,z axis directions is partitioned into N intervals which would define a cubic lattice of N^3 cells. Here we'll consider casting just just one ray through this bounding volume. We allow its direction to be arbitrary.

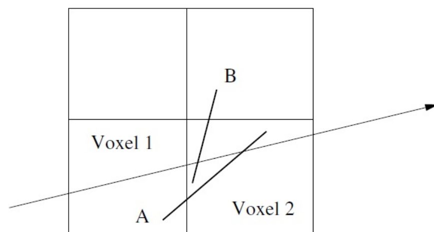
The voxels partition space into disjoint sets of points [13] But the voxels need not partition the set of surfaces into disjoint sets, since a surface might belong to more than one voxel. Here is an algorithm for ray casting using this simple "3D uniform voxel grid" data structure. First we initialize our t and closest polygon p.

```
t =  infinity
p = NULL
```

[13] Here we ignore the fact that voxels can share faces, so the word "partition" is being used a bit loosely here.)

```
current_voxel := voxel containing the starting point of ray
while (t == infinity){
    for each surface in current voxel{
        t_intersect = distance to surface along ray    // infinite if no intersection
        if (t_intersect < t) and (intersection point belongs to current voxel)
            t = t_intersect
            p = surface
    }
    current voxel = next voxel hit by the ray
}
```

One subtlety in this algorithm is the test condition that the intersection point belongs to current voxel. Without this condition, the algorithm would be incorrect. Consider the example shown below. *(In fact, it is basically the same example shown twice. I need to fix that figure.)* Suppose we are currently examining voxel 1. Surface A belongs to voxels 1 and 2. The ray intersects surface A in voxel 2. This intersection would be discovered when the current voxel was 1. At that time, t would be set to a finite value. The `while` loop would thus terminate after examining surfaces that belong to voxel 1. However, surface A would in fact *not* be visible along the cast ray, since there is a surface B which is closer. But surface B would only be examined when the current voxel is 2. The algorithm thus needs to ignore the intersection with surface A at the time it is in voxel 1 (since the intersection itself does not occur in voxel 1).



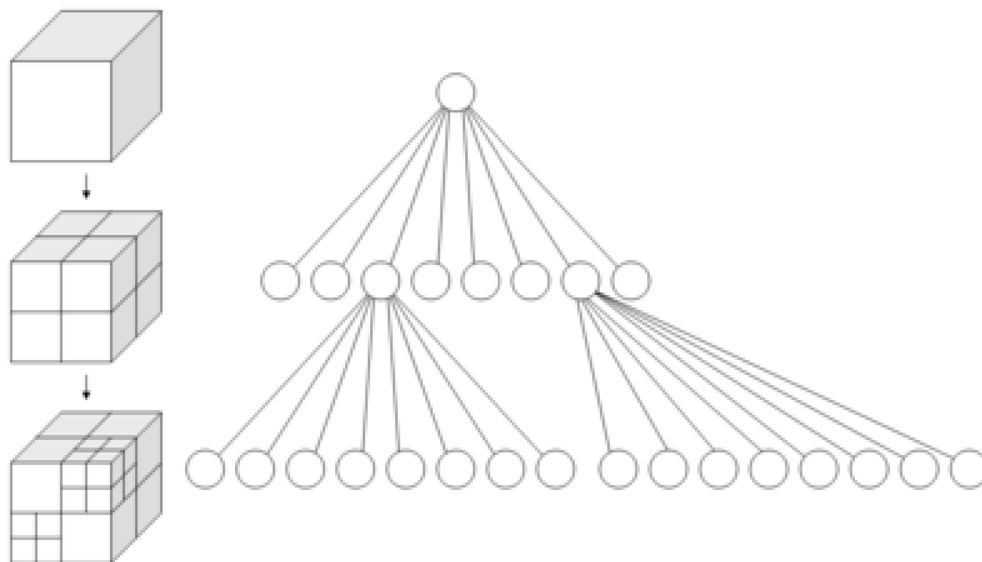
How many voxels should one use? If we use N^3 voxels, how big should N be? If N is very large, then the memory requirements will be large. There will there will be relatively few surfaces intersecting each voxel, and each surface will belong to a large number of voxels. There will be $O(N)$ voxels along each ray which would result in many checks of empty voxels. This would be inefficient. On the other hand, if we use a relatively small N, then there would tend to be some voxels that have many surfaces and so this would lead to testing for intersections of many surfaces and performance would end up being similar to the original ray casting approach. The way to improve the situation is to use non-uniform spatial partitions. Let's turn to a classic one next.

Octrees

Start with a cube. Define an *octree* data structure for partitioning this cube. The cube is partitioned into eight subcubes (voxels) of equal volume, by dividing each of the xyz dimensions in two equal

parts. Each of these subcubes can itself be partitioned into eight equal subcubes, etc. However, we only partition a cube if at least one surface intersects the cube.

Each non-leaf node of an octree has eight children, corresponding to these eight subcubes. A node at depth d is of width 2^{-d} times some constant which is the width of the original cube. See Figure below, which was taken from wikipedia.



At each *leaf* of the octree, store a list of surfaces that intersect the subcube (voxel) defined by that leaf. Surfaces are referenced only at the leaves.

There is some freedom in how one defines the octree, namely how one decides when to stop partitioning a voxel. An octree can be defined to some pre-chosen maximal tree depth, or up to a maximal number of surfaces per leaf, or some other criterion.

How are octrees used in ray casting? The basic idea is to cast the ray and examine the voxels/cells through which the ray passes. As in uniform ray casting, we examine the voxels in front-to-back order. The algorithm for ray casting using an octree is the same as the one for the uniform spatial subdivision above. The only difference is the implementation of the line

```
current voxel := next voxel hit by the ray
```

This requires that we traverse the tree in a particular manner. (See Exercises.)

As in all ray casting methods, the octree method searches from near voxels to far voxel. It is particularly efficient because the search for the closest object tends to terminate quickly, since we have sorted the objects into groups along the ray. There are extra cost one must pay though: the octree needs to be constructed in advance, and the algorithm for traversing the tree is more complex than the algorithm for the uniform space partition.

Octrees are *not* used by OpenGL. They are used (heavily!) by a system called RADIANCE `http://radsite.lbl.gov/radiance/`.

Hierarchical Bounding Volumes

Let's assume that we've clipped the scene already so that all the objects are restricted to lie in the view volume, which obviously does not contain the viewer itself.

Suppose you have a surface with a complex shape that is described by many polygons or other surfaces. For example, a chair might consist of hundreds of polygons that precisely describe its shape. It would be expensive to check if the ray intersects each of these polygons. One quick method to examine whether a ray intersects such a surface is to consider a much simpler volumetric shape (a cube, a sphere, a cylinder) whose volume strictly contains (bounds) the surface. If the ray intersects the surface then it must intersect the bounding volume. We use the contrapositive statement. If the ray *doesn't* intersect the bounding volume, then it *doesn't* intersect the surface. So, to check if a ray intersects the surface, we first *quickly* check if the ray intersects the bounding volume. If it doesn't, then we don't have to check any of the polygons that belongs to the surface. But if the ray does intersect the bounding surface, then it may or may not intersect the surface. We don't know, and we need to check further.

It is common to take this a step further and consider a *hierarchy* of bounding volumes. Consider a *tree* whose internal nodes correspond to bounding volumes, such that the bounding volume of a (non-root) internal node is contained in the bounding volume of its parent node. The bounding volumes could be defined any way you like (e.g. by rectanguloids, spheres, cylinders, or whatever you want). The key is that it is easy to compute the intersection between a ray and a bounding volume.

At each leaf of the tree, rather than having a bounding volume, we have a list of objects/surfaces which are entirely contained in the bounding volume defined by their parent node. It is also common to define the tree so that each leaf consists of just one surface.

Although the bounding volume of a child is contained in the bounding volume of a parent, there is no containment relation for siblings. For example, we might have an internal node whose bounding volume contains a person that is sitting in a chair. The node might have two children, namely a bounding volume for the person and a bounding volume for the chair. These two bounding volumes would overlap, in the case that the person is sitting in the chair.

How do we use this bounding volume hierarchy for efficient ray casting? Let's keep the discussion as general as we can, and consider casting a ray out from some point in a scene. We wish to find the closest object in the scene that is intersected by the ray. To do so, we do a depth first search of the bounding volume tree and try to find the closest surface. We check the bounding volume of the root node first and, if the ray intersects this volume, then we check each of the children, and proceed recursively. The key idea is that if a ray *does not* interest some bounding volume (an internal node), then we *do not* need to check the children of this node.

For some images illustrating bounding volumes (and octrees), see the illustrations in my lectures slides. These illustrations are taken from:
`http://www.cs.princeton.edu/courses/archive/spring14/cos426/lectures/12-ray.pdf`

Below is an algorithm for the above "hierarchical bounding volume" tree traversal.[14] First initialize global variables.

```
p =   NULL     //  Pointer to closest surface
t = infinity   //  Distance to closest surface found so far
```

Then define a recursive procedure that modifies these global variables. We call it with `traverseHBV(root,t)`.

```
traverseHBV( ray, node){
   if (node is a leaf)
      for each surface in leaf
           cast ray i.e. compute t_intersect
            //  infinity if ray doesn't intersect surface
           if (t_intersect < t)
               t := t_intersect
               p := surface  // point to surface
   else // node is bounding volume
       for each child of node
           traverseHBV(ray, child)
}
```

The above algorithm does a depth first search. For each node, it "expands" the node whenever there is a potential that this node contains a closer surface than the current closest surface. One inefficiency of this algorithm is that it could spend its time searching through a complex bounding volume subtree when in fact none of the surfaces in that bounding volume are visible. You would somehow like to design the algorithm to try to avoid this possibility. The basic idea is to be more clever in how you choose which child to recursively call next. See the Exercises.

[14] Rubin and Whitted, SIGGRAPH 1980)

Lecture 9

It is often natural to model objects as consisting of parts, and the parts themselves as consisting of subparts, etc. Think of a bicycle. It has 2 wheels, a frame, a seat, handle bars, etc. Each wheel consists of a tire, spokes, hub, and rim. The frame consists of several cylindrical parts. The seat consists of a frame and cushion and mount, etc. Another example is a person's body. The body consists of a head, a torso, and limbs.

It is useful to think of these hierarchies as having a tree structure.

A person's body might be a root node. Its two children might be upper body and lower body. Each of these "parts" would consist of subparts and hence child nodes. We can imagine a set of draw commands which define a call tree correspdonding the the drawing of parts and subparts. In the lecture, I sketched out such a call tree for the parts of the human body.

When drawing a body or any hierarchical object in OpenGL, one also needs to keep in mind a different kind of tree that describes the coordinate systems of the various objects and parts. The root of the tree might be the coordinate system of some object (or perhaps the world coordinate system). Then any parent-child pair in the tree represents the coordinate systems of two objects and the edge represents a mapping from one coordinate system to the other.

To draw all the parts of all the objects in a scene, we essentially need to traverse this tree as well. It is common to to keep track of the `GL_MODELVIEW` matrix, by using a stack. OpenGL provides `glPushMatrix()` and `glPopMatrix()` commands for doing so. Before we descend from parent to child, we push the current `GL_MODELVIEW` matrix onto the stack. When we return to the parent, we pop the stack to recall the `GL_MODELVIEW` matrix at the parent. I sketched out how to do that in the lecture slides as well.

In the rest of this lecture, I'll briefly describe two notions of object hierarchies that have been very useful and commonly used in computer graphics, namely fractals and L-systems.

Fractals

We want to make pictures of many kinds of objects. Some objects such as spheres and cylinders have smooth surfaces. However other objects in the world do not have smooth surfaces, and indeed what makes many objects visually interesting and characteristic is that they are *not* smooth. For example, a pile of mud, a broccoli, a rocky mountain, a sponge, a cloud, a lung or kidney, etc are certainly not smooth objects. Such objects have been modelled as *fractals*.

What are fractals? Fractals are rough (non-smooth) objects, so rough that it is impossible to define their length/ area/ volume in the usual way. A famous example is the coastline of some rocky island, such as England. How would you measure the length of the coastline? One idea is to use a measuring instrument and to count how many "steps" you need to take with this instrument to go once around the object. The length is step size multiplied by the number of steps. When we are dealing with a smooth curve, the length of the curve is the limit of step size times numsteps as stepsize goes to zero. You know this well from Calculus.

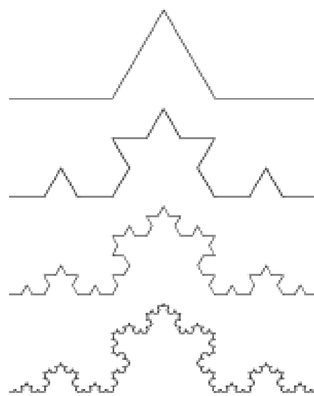
The problem with non-smooth objects is that the limit mentioned above doesn't exist. For example, take the case of a rocky coastline and start with 1 km steps. The total length of the coastline would be not be the same as if you used 1 m steps. The reason is that there would be indentations and protrusions in the coastline that occur at a scale of 1 m and that you would "step

over" with your 1 km instrument. But the same problem arises if you were to take 1 cm steps. There would be little protrusions and indentations that the 1 m stick would step over but the 1 cm stick would not. For a smooth curve, the total length asymptotes as the step size becomes small because, "in the small" the coastline is approximately a straight curve. This is just not the case for rocky coastlines though.

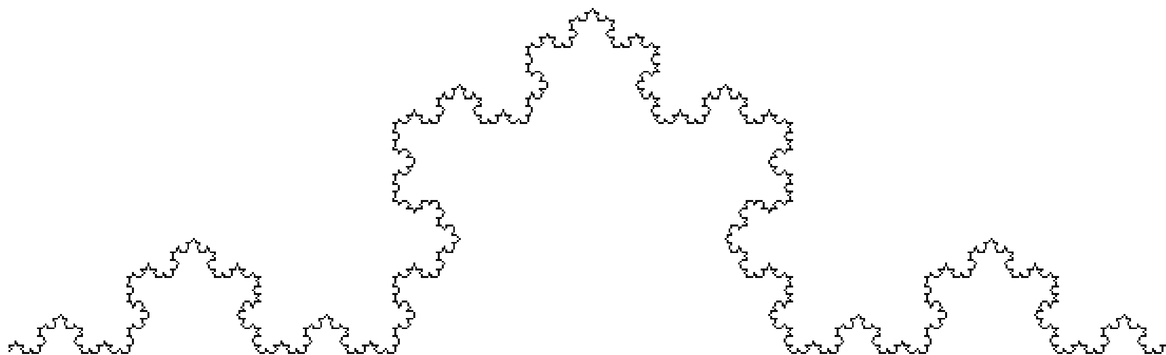
Fractals have been studied by mathematicians for many decades. They became much more popular with the work of Benoit Mandelbrot who pointed out how often real world objects and phenomena have fractal-like behavior. Mandelbrot introduced both new determininstic (non-random) fractals e.g. the Mandelbrot set, and also discussed random fractals like Brownian motion. Here I will touch on some of the basics of these fascinating objects and give some examples.

Koch Curve

Consider a line segment of length 1. Suppose replace it with four line segments of length 1/3 each, as in the figure below. Then we replace each of these line segments of length 1/3 each, by four line segments of length length $\frac{1}{3^2}$ each. This gives a total length of $(\frac{4}{3})^2$. If we repeat this recursively n times, we get a curve of total length $(\frac{4}{3})^n$ composed out of 4^n line segments, each of length $\frac{1}{3^n}$. The figure below shows the construction after 1,2,3,4 steps.



We expand the drawing (by rescaling by a factor of 3) to see the reconstruction after 5 steps.



The length of the curve goes to ∞ as $n \to \infty$. The limiting curve is called the *Koch Curve*.

The Koch curve is said to be *self-similar*. If you compare the whole Koch curve to the part of the Koch curve that is generated by any single one of the four line segments of length $\frac{1}{3}$ (at step 1), then you get the same curve except that it is scaled by a factor of $\frac{1}{3}$ (and then translated and perhaps rotated).

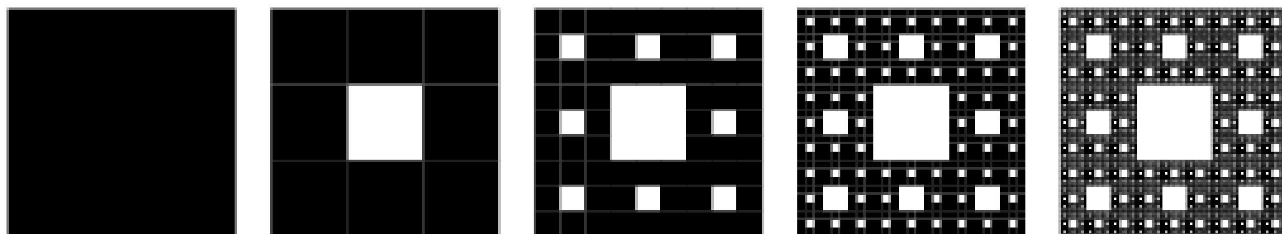
We can write a recursive function for drawing a Koch curve in OpenGL as follows:

```
def koch(i):
    if i == 0
        drawline()
    else if i > 0
        glPushMatrix()
        glScalef(1.0/3,1.0/3,1.0/3)
        koch(i-1)
        glTranslatef(1.0, 0.0, 0.0)
        glRotatef(60, 0.0, 0.0, 1.0)
        koch(i-1)
        glTranslatef(1.0, 0.0, 0.0)
        glRotatef(-60, 0.0, 0.0, 1.0)
        glRotatef(-60, 0.0, 0.0, 1.0)
        koch(i-1)
        glTranslatef(1.0, 0.0, 0.0)
        glRotatef(60, 0.0, 0.0, 1.0)
        koch(i-1)
        glPopMatrix()
```

Sierpinski carpet

Here's another interesting example. Begin with a unit square, and partition it into nine subsquares, i.e. a 3×3 grid of squares, each of area $\frac{1}{9}$. We then delete thecentral subsquare, leaving a remaining area of $\frac{8}{9}$. For each of these eight remaining subsquares, we again partition it into nine sub-squares and delete the central one. The total area remaining is now $(\frac{8}{9})^2$. We proceed recursively n times, and we are left with a total area of $(\frac{8}{9})^n$. This area goes to 0 as $n \rightarrow \infty$.

Notice there are lots of points that are never deleted. For example, the boundary of any of the subsquares is never deleted since we only delete interiors. Each subsquare at step $n \geq 1$ has perimeter (boundary length) $4(\frac{1}{3})^n$ and there are 8^{n-1} such subsquares. Thus the total perimeter (length) of the boundary of the deleted subsquares is $8^{n-1}4(\frac{1}{3})^n$ which goes to ∞ as $n \rightarrow \infty$.

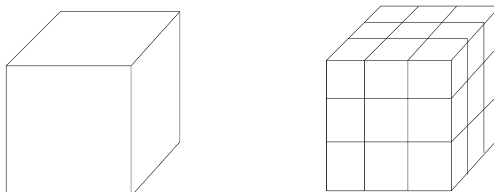


The limiting shape is called the *Sierpinski carpet*.

Both the Koch Curve and Sierpinski carpet are infinitely "long" (if we try to measure them in 1D, as curves) but have infinitesimal "area" (if we try to measure them in 2D, as areas). Both of these seem to have dimension somewhere between 1 and 2. Intuitively, the Sierpinski carpet consists of more points than the Koch curve, but it is not yet clear how to quantify this.

Sierpinski Cube

The Sierpinski cube is the 3D version of the Sierpinski carpet. We partition a unit cube, rather than a square. Partition the unit cube into 27 subcubes ($3 \times 3 \times 3$) each of volume $\frac{1}{27}$. **[Here I give a slightly different version than the one in the lecture.]** In this version, I delete only the interior of the central subcube. This leaves a total volume $\frac{26}{27}$. For each of these remaining 26 subcubes, again partition it into 27 subcubes (now each of volume $(\frac{1}{27})^2$) and delete the central subcube from each of these. The total remaining volume is now $(\frac{26}{27})^2$. If we repeat this n times, the remaining volume is $(\frac{26}{27})^n$, which goes to 0 as $n \to \infty$.



Fractal dimension

What do we mean by dimension anyhow? One simple idea is that we are referring to the rate at which the size of a set grows as we *scale* it. Suppose we are talking about subsets of $\Re^3$. We are familiar with talking about points (0D), curves (1D), surfaces (2D), and volumes (3D). But what do we mean by nD ?

If we take a line segment of length 1, and we scale it by a factor, say $S = 2$, then get a new line segment such that we require $C = 2$ copies of the original line segment to cover the newly scaled line segment. What about area? Suppose we have a unit square. If we scale it by $S = 2$, then get a new square of area S^2 and so we need $C = S^2$ of the original squares to cover the new square. The same idea holds for volume. If we have a unit cube and we scale it by $S = 2$, then we get a new cube of volume S^3, which means that we need $C = S^3$ unit cubes to cover this new cube.

In general, if have an object that is of unit size and is of dimension D, and we scale this object by $S > 1$, then we need how many copies to cover the newly scaled object? We need

$$C = S^D$$

objects of unit size (measured in that dimension). Taking the log of both sides, we get

$$D \equiv \frac{\log C}{\log S}$$

Verify for yourself that this definition holds for a line segment, square, and cube. Now, let's use this observation as a *definition* of dimension and apply it to the objects introduced above.

What is the dimension of the Koch curve? If one starts with a Koch curve, and makes $C = 4$ copies, one can piece them together to make a scaled Koch curve where the scale factor is $S = 3$. Thus, according to the above definition, the dimension of the Koch curve is $D = \frac{\log 4}{\log 3} \approx 1.26$. Note that the dimension of the Koch curve is strictly between 1 and 2 which is consistent with the idea that its dimension is greater than that of a line segment and less than that of a square.

What about the Sierpinski carpet? If we make $C = 8$ copies of the Sierpinskicarpet described above, then we can piece them into a 3×3 grid (with the center one missing) and this gives a scaled Sierpinski carpet such that the scale factor is $S = 3$. Thus, the dimension of the Sierpinski carpet $D = \frac{\log 8}{\log 3} \approx 1.88$. Note that the dimension of the Sierpinski carpet is greater than the dimension of the Koch curve, and that both have dimension strictly between 1 and 2.

For the 3D version of the Sierpinski carpet, we can take $C = 26$ copies and make a 3D Carpet that is a scale version where the scale factor is $s = 3$. Thus the dimension is $D = \frac{\log 26}{\log 3} \approx 2.96$. Notice that this is very close to 3, but still less than 3.

Random fractals e.g. Midpoint displacement method

Just as a sphere and and a square are too symmetric to be for modelling smooth objects, the objects we have just defined are too symmetric to be used to model natural complicated objects like coastlines, rock faces, broccoli, and clouds. To model these, we need to destroy the symmetry. We can do so by adding random perturbations. Here are some basic ideas of how to do so.

Suppose we take a 1-d function $f(x)$ with endpoints $f(0) = a$ and $f(1) = b$. We can make a line joining these two endpoints. The midpoint has $f(\frac{1}{2}) = (a + b)/2$. If we add a random amount to this midpoint value, we get $f(\frac{1}{2}) = \frac{a+b}{2} + \delta$, where δ is chosen from a suitable probability density function. For example, it might be normal (Gaussian) with mean zero and standard deviation σ.

We then repeat this "midpoint displacement" for the two line segments from $(0, f(0))$ to $(\frac{1}{2}, f(\frac{1}{2}))$ and from $(\frac{1}{2}, f(\frac{1}{2}))$ to $(1, f(1))$. We partition each line segment into two halves, giving us midpoints $x = 1/4$ and $x = 3/4$. We then add a random perturbation δ to each, chosen from a suitable probability density function. And so on... At step n, we have 2^n line segments. We can repeat this recursively, adding more and more detail as we wish.

How should we choose probability density functions to generate the δ values? There is no single answer to this. It depends on how rough we wish the curve to be. If we perturb only by a little, then the curve will be close to smooth. If we perturb by alot, then the curve will be very rough.

On the next page is Python code for midpoint displacement algorithm. It takes two parameters. One is a `roughness` parameter which is a number between 0 and 1. The other is the standard deviation of the displacement, which has a normal distribution – see Python `numpy.random.normal`. (If you don't know what a normal distribution is, you probably want to look it up.) At each level of the recursion the standard deviation is reduced by some factor, namely the `roughness` parameter. If you use a roughness parameter of $\frac{1}{\sqrt{2}}$ then the curve you get has fractal-like properties, namely statistical self similarity. (To state this claim more carefully and to prove it is true requires mathematics beyond this course.)

```
def  midpointDisplacement(a, roughness, stdev):
   size = len(a)
   newStd = stdev * roughness
   if (size <= 2):
       return a
   else:
       middle = size / 2
       a[middle] = (a[0] + a[size-1])/2 +random.normal(0, newStd)
       a[0:middle+1]  = midpointDisplacement(a[0:middle+1], roughness, newStd)
       a[middle:size] = midpointDisplacement(a[middle:size], roughness, newStd)
   return a
```

The slides show several examples of curves generated with different roughnesses. [At the time of this writing, I am still trying to decide whether to give you something on this material in A2 or A3. I may edit this later once this issue has been settled.]

L-systems

Earlier this lecture I gave recursive code for drawing a Koch curve. Let's write the code in a slightly different way, where we describe how to generate the Koch curve by replacing each line with a four smaller lines in a particular arrangement.

Let [and] denote `glPushMatrix` and `glPopMatrix`, respectively. Let **L** refer to the drawing of a unit line in the x direction, and let **T** denote a unit translation in the x direction. Let **S** be scaling by a factor $\frac{1}{3}$. Let **R** be a rotation by 60 degrees counterclockwise and let $\mathbf{R}'$ be a rotation 60 degrees clockwise. The first step of approximating a Koch curve is just to draw a line of length 1, so the string is just **L**. In the second step, the string L is replaced as follows: In the third step, each occurence of **L** in the second step is replaced (recursively) by the same substitution, etc.

$$\mathbf{K} \rightarrow \ [\ \mathbf{S\ K\ T\ R\ K\ T\ R'\ R'\ K\ T\ R\ K}\ ]$$

Note that this recursion, as written, doesn't have a base case. Later this lecture we will add notation to indicate a base case.

This idea of recursively replacing symbols with strings will be familiar to most of you. It is reminiscent of formal grammars that are used to define programming languages e.g. parse trees. Here we will not be interested in *parsing* such strings, however. Rather we will be interested only in the simpler problem of *generating* such strings.

This idea of generating a shape by recursive substitution was developed by Astrid Lindemayer, and the method has come to be known as L-systems. Lindemayer was interested in growth and form of biological structures, in particular, plants. The idea is that the parts of a plant (stems, leaves, flowers, branches, etc) at any given time is described by a string that has been generated by a sequence of "productions". As the plant grows over time, some parts grow, some parts create new parts, some parts die. To describe such growth formally, some symbols in the string are replaced by new symbols, and other symbols disappear.

L-systems began to be used in computer graphics in the 1980's. Download the book "The Algorithmic Beauty of Plants" by Przemyslaw Prusinkiewicz[15] and Astrid Lindemayer. The book was published in 1990 and is now out of print. It is available online at `http://algorithmicbotany.org/papers/#abop` Have a look through some of the examples, in particular, Chapter 1 (p. 25) of that book.

For those of you who are familiar with formal grammars: an L-system is defined by a set of symbols (alphabet) $\mathcal{A}$ which includes a special symbol $a \in \mathcal{A}$ called an *axiom* or starting symbol. There is also a set of *production rules*

$$P : \mathcal{A} \rightarrow \mathcal{A}^*$$

where $\mathcal{A}^*$ is the set of strings made up of symbols from $\mathcal{A}$ which may include the empty string. A specific production $p \in P$ is written

$$p : a \rightarrow \chi$$

where $\chi \in \mathcal{A}^*$. If no production is explicitly specified for a symbol a, then we assume the identity production

$$p : a \rightarrow a \ .$$

While L-systems are similar to formal grammars, there is a key difference: with L-systems the productions are applied to each symbol in the string *in parallel*. With formal grammars, only one production is applied at each step (and the ordering in which productions are applied is often important for the analysis). Also, we use L-systems to generate sequences, not to analyze (parse) them.

Example: plant

Here is an example for drawing a cartoon plant. This example was modified slightly from the example given in the book Algorithm Beauty of Plants (p. 25, Fig. 1.24d). The recursion can be written:

$$\mathbf{P} \rightarrow \ [\ \mathbf{S\ D\ T}\ [\ \mathbf{R\ P}\ ]\ \mathbf{D\ T}\ [\ \mathbf{R'\ P}\ ]\ \mathbf{R\ P}\ ]$$

and the base case is:

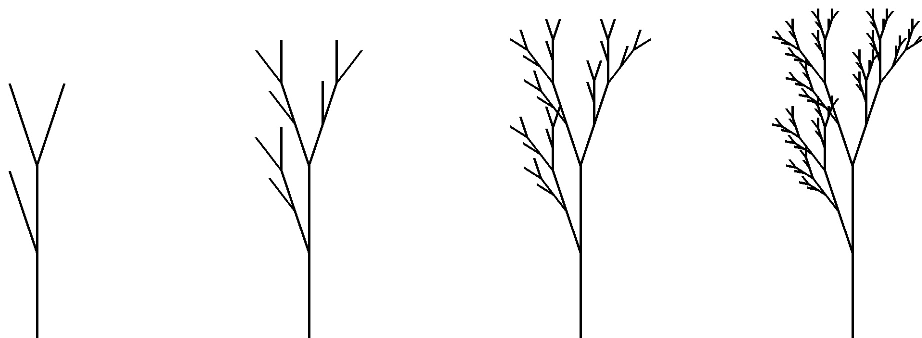
$$\mathbf{P} \rightarrow \mathbf{D}$$

Here

- **S** is a scaling by 0.5 in x and y
- **R** is a counterclockwise rotation by 30 degrees
- **T** is a unit length translation in y

Shown below are four stages of these productions. I have not bothered to include the zeroth stage, in which there is a single vertical line.

[15] Professor Prusinkiewicz is at the University of Calgary and here is his lab website `http://algorithmicbotany.org/`



Parametric L-systems

In the above examples, the symbols had parameters associated with them. For example, the rotations had a particular angle, the translation have a direction, scaling has a scalar. Since the symbols are just functions calls, these parameters would act just as function parameters. In the context of OpenGL, this is just obvious – we're talking about the parameters of the rotate, translate, and scaling functions. Another example of a parameter is the level for the recursion.

You can add the parameters in the obvious way, e.g $\mathbf{S}_x(\frac{1}{3})$ or $\mathbf{R}_z(30)$. Indeed the only reason I didn't include that notation originaly was to keep it simple!

Here is how you could write the conditions of the recursion more explicitly:

$$p_1 \; : \; \mathbf{P}(n) : n > 1 \rightarrow \; [\; \mathbf{S}\; \mathbf{D}\; \mathbf{T}\; [\; \mathbf{R}\; \mathbf{P}(n-1)\;]\; \mathbf{D}\; \mathbf{T}\; [\; \mathbf{R}'\; \mathbf{P}(n-1)\;]\; \mathbf{R}\; \mathbf{P}(n-1)\;]$$
$$p_2 \; : \; \mathbf{P}(n) : n = 0 \rightarrow \mathbf{D}$$

Open L-systems

Another powerful technique is to allow the productions to depend on global variables. This allows plants to interact with each other and with the environment. For example, suppose we consider the roots of a plant growing into the ground. We could define a 3D array `water(x,y,z)` that represents the amount of water at each point in the soil. As the roots grow, the plant absorbs water from the soil. At each time step, a root growth production could occur if there is a sufficient amount of water present in the grid cell closest to the tip of the root. Moreover, as the root grows, it could decrease the amount of water in that grid cell.

Another idea is to allow a branches to grow only if they receive a sufficient amount of light from the sky. At each time step, you could cast a set of rays toward the sky and count the percentage of rays that do not intersect branches or leaves above. If the number of rays that "see the sky" is sufficiently large, then you could allow the growth production to occur; otherwise, not.

Finally, in the lecture, I also mentioned probabilistic L-systems where productions could occur or not, based on the outcome of some random event (e.g. a coin flip). See the book Algorithmic Beauty of Plants for examples of this, and many more examples too.

Today I will give an introduction to the topic of smooth curves and surfaces in $\Re^3$.

Cubic Curves (and splines)

Up to now, we have mostly considered simple linear shapes – lines, polygons — and second order shape, i.e. quadrics. (Last lecture was an obvious exception, where we considered fractals.) Today we consider smooth parametric curves and surfaces that are defined by third order polynomials. Such curves and surfaces are useful for two reasons. First, they can be used to define the paths of points in an animation. These paths can be the trajectory of an object, or they can be the paths of the camera. Second, smooth curves can be used to define shapes, not just the shapes of 1D objects (curves) but also shapes of surfaces as we'll see in the second part of this lecture.

Many times in this course we have defined a line (or ray) using a parametric curve. Suppose we have two points $\mathbf{p}(0), \mathbf{p}(1)$. Then we define a line that passes between them by

$$\begin{aligned}\mathbf{p}(t) &= \mathbf{p}(0) + t\,(\mathbf{p}(1) - \mathbf{p}(0)) \\ &= (1-t)\,\mathbf{p}(0) + t\,\mathbf{p}(1)\end{aligned}$$

Suppose we want to define a 3D curve that passes through three points $\mathbf{p}(0), \mathbf{p}(1), \mathbf{p}(2) \in \Re^3$. This is relatively easy. We can always fit a circle to any three points, as long as they are not collinear. We can define a parametric equation for a circle but this requires sines and cosines and they are expensive to compute. It turns out to be easier to go to a slightly more general model, the cubic curve.

Let's define a 3D parametric curve

$$\mathbf{p}(t) = (x(t), y(t), z(t))$$

using functions $x(t), y(t), z(t)$ that are polynomials of degree 3, namely

$$\begin{aligned}x(t) &= a_x t^3 + b_x t^2 + c_x t + d_x \\ y(t) &= a_y t^3 + b_y t^2 + c_y t + d_y \\ z(t) &= a_z t^3 + b_z t^2 + c_z t + d_z\end{aligned}$$

Since we have 12 coefficients in total, we might guess that we can specify a cubic curve by chosing four points in $\Re^3$ (since $4 \times 3 = 12$). This is indeed the case, as we see below.

The technique to fitting cubic curves boils down to the following idea. Consider the case $x(t)$ which we rewrite

$$x(t) = \begin{bmatrix} a_x & b_x & c_x & d_x \end{bmatrix} \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix}. \quad (1)$$

We want to find coefficients a_x, b_x, c_x, d_x so that the function takes given values, say $x(0), x(1), x(2), x(3)$ at $t = 0, 1, 2, 3$. We can do so by substituting in for t and grouping into four columns.

$$\begin{bmatrix} x(0) & x(1) & x(2) & x(3) \end{bmatrix} = \begin{bmatrix} a_x & b_x & c_x & d_x \end{bmatrix} \begin{bmatrix} 0 & 1 & 8 & 27 \\ 0 & 1 & 4 & 9 \\ 0 & 1 & 2 & 3 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

We can then find the coefficients a_x, b_x, c_x, d_x by right multiplying by the inverse of the 4×4 matrix on the right side. We will use that inverse matrix again below. We call it **B**.

$$\mathbf{B} \equiv \begin{bmatrix} 0 & 1 & 8 & 27 \\ 0 & 1 & 4 & 9 \\ 0 & 1 & 2 & 3 \\ 1 & 1 & 1 & 1 \end{bmatrix}^{-1}$$

We fit a curve $\mathbf{p}(t)$ to four points in $\Re^3$ using the same technique. Define $\mathbf{p}(t)$ to be a column vector:

$$\mathbf{p}(t) = \begin{bmatrix} x(t) \\ y(t) \\ z(t) \end{bmatrix} = \begin{bmatrix} a_x & b_x & c_x & d_x \\ a_y & b_y & c_y & d_y \\ a_z & b_z & c_z & d_z \end{bmatrix} \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix} \quad (2)$$

Let $\mathbf{p}(0)$, $\mathbf{p}(1)$, $\mathbf{p}(2)$, $\mathbf{p}(3)$ be the four given values specified on the desired curve at $t = 0, 1, 2, 3$. We write these four points as a 3×4 matrix which is called a *geometry matrix*, or matrix of *control points*.

$$\mathbf{G} \equiv \begin{bmatrix} \mathbf{p}(0) & \mathbf{p}(1) & \mathbf{p}(2) & \mathbf{p}(3) \end{bmatrix}$$

Now substitute the four values $t = 0, 1, 2, 3$ into Eq. (2) and group the four substituted vectors into a four column matrix on both the left and right side:

$$\begin{bmatrix} \mathbf{p}(0) & \mathbf{p}(1) & \mathbf{p}(2) & \mathbf{p}(3) \end{bmatrix} = \begin{bmatrix} a_x & b_x & c_x & d_x \\ a_y & b_y & c_y & d_y \\ a_z & b_z & c_z & d_z \end{bmatrix} \begin{bmatrix} 0 & 1 & 8 & 27 \\ 0 & 1 & 4 & 9 \\ 0 & 1 & 2 & 3 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

We then solve for the a, b, c, d coefficients by right-multiplying by matrix **B** just as we did on the previous page. i.e.

$$\mathbf{G}\,\mathbf{B} = \begin{bmatrix} a_x & b_x & c_x & d_x \\ a_y & b_y & c_y & d_y \\ a_z & b_z & c_z & d_z \end{bmatrix}$$

We rewrite Eq. (2) as:

$$\mathbf{p}(t) = \begin{bmatrix} x(t) \\ y(t) \\ z(t) \end{bmatrix} = \mathbf{G}\,\mathbf{B} \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix}.$$

The product

$$\mathbf{B} \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix}$$

is a 4×1 column vector whose rows (single elements) are *blending functions*. These blending functions are polynomials of degree 3 in t which define the weights of each of the four control points

$\mathbf{p}(0), \mathbf{p}(1), \mathbf{p}(2), \mathbf{p}(3)$ to the curve point $\mathbf{p}(t)$. That is, for any t, $\mathbf{p}(t)$ is a linear combination of the four control points, and the blending functions specify the weights of the combinations.

Suppose we would like to make a curve that passes through n points instead of 4 points. How could we do this? One way would be to take points 1-4 and fit a cubic curve to these points. We could then take points 4-7 and fit a cubic curve to them and piece the two curves together, etc for points 7-10, 10-13, etc. The problem is that the curve might have a kink in it at points 4, 7, 10, etc that is, it might change direction suddenly. To avoid this problem, we need to model explicitly the derivative of the curve and make sure the derive is continuous at the points where we join the curve pieces together.

Hermite Curves (and splines)

Instead of using four points, we use two points $\mathbf{p}(0)$ and $\mathbf{p}(1)$ and we specify the tangent vector at each of these two points. By *tangent vector*, I mean the derivative with respect to the parameter t:

$$\mathbf{p}'(t) \equiv \frac{d\mathbf{p}(t)}{dt} \equiv lim_{\delta t \to 0} \frac{\mathbf{p}(t+\delta t) - \mathbf{p}(t)}{\delta t}$$

If t were interpreted as time, then the tangent vector would be the 3D velocity of a point following the curve. This interpretation is relevent, for example, in animations if we are defining the path of a camera or the path of a vertex.

Taking the derivative of Eq. (2) gives

$$\mathbf{p}'(t) = \begin{bmatrix} x'(t) \\ y'(t) \\ z'(t) \end{bmatrix} = \begin{bmatrix} a_x & b_x & c_x & d_x \\ a_y & b_y & c_y & d_y \\ a_z & b_z & c_z & d_z \end{bmatrix} \begin{bmatrix} 3t^2 \\ 2t \\ 1 \\ 0 \end{bmatrix}$$

Substituting for $t = 0$ and $t = 1$ for $\mathbf{p}(t)$ and $\mathbf{p}'(t)$ gives us a 3×4 Hermite geometry matrix:

$$\begin{aligned} \mathbf{G}_{Hermite} &\equiv \begin{bmatrix} \mathbf{p}(0) & \mathbf{p}(1) & \mathbf{p}'(0) & \mathbf{p}'(1) \end{bmatrix} \\ &= \begin{bmatrix} a_x & b_x & c_x & d_x \\ a_y & b_y & c_y & d_y \\ a_z & b_z & c_z & d_z \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 3 \\ 0 & 1 & 0 & 2 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix}. \end{aligned}$$

Of course, you should realize here that we are just taking the 1D problem,

$$\begin{bmatrix} x(0) & x(1) & x'(0) & x'(1) \end{bmatrix} = \begin{bmatrix} a_x & b_x & c_x & d_x \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 3 \\ 0 & 1 & 0 & 2 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

and similarly for $y(t)$ and $z(t)$ and then stacking the three solutions. That is, there is no *interaction* of x, y, z in the solution.

As before, we can compute the coefficient matrix by right multiplying by the inverse of the 4×4 matrix at the right end of the above equation. Define this inverse to be

$$\mathbf{B}_{Hermite} \equiv \begin{bmatrix} 0 & 1 & 0 & 3 \\ 0 & 1 & 0 & 2 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix}^{-1}$$

and define the 3×4 *geometry matrix* (or *control points*)

$$\mathbf{G}_{Hermite} \equiv \begin{bmatrix} \mathbf{p}(0) & \mathbf{p}(1) & \mathbf{p}'(0) & \mathbf{p}'(1) \end{bmatrix}.$$

Then we can write

$$\mathbf{p}(t) \;=\; \mathbf{G}_{Hermite}\,\mathbf{B}_{Hermite} \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix}. \tag{3}$$

The 4×1 column vector

$$\mathbf{B}_{Hermite} \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix}$$

contains four *Hermite blending functions*. These four functions are each third order polynomials in t, and define the contribution of the four control points $\mathbf{p}(0), \mathbf{p}(1), \mathbf{p}'(0), \mathbf{p}'(1)$, respectively, to the curve $\mathbf{p}(t)$. That is, $\mathbf{p}(t)$ is a linear combination of $\mathbf{p}(0), \mathbf{p}(1), \mathbf{p}'(0), \mathbf{p}'(1)$ and the weights of this combination are the blending coefficients.

Now suppose we want to draw a more complicated curve in which we have n points $\mathbf{p}(0)$ to $\mathbf{p}(n-1)$ and we have tangents $\mathbf{p}'(0)$ to $\mathbf{p}'(n-1)$ defined at these points as well. We would like to fit a curve (say Hermite) between each pair of points. We can be sure that the curve passes through the points and that the tangents are continuous, simply by making the endpoint and tangent of one curve segment (say, from $t = k-1$ to $t = k$) be the same as the beginning point and tangent of the next curve segment (from $t = k$ to $t = k+1$). This defines a *Hermite (cubic) spline*.

Bézier Curves

A slightly different method for defining a curve is to specify the tangent vectors in terms of points that are not on the curve. Given $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$, define a curve $\mathbf{q}(\mathrm{t})$ where $t \in [0, 1]$. Let respectively, and define the control points $\mathbf{p}_1$ and $\mathbf{p}_2$ such that

$$\begin{aligned} \mathbf{q}(0) &= \mathbf{p}_0 \\ \mathbf{q}(1) &= \mathbf{p}_3 \\ \mathbf{q}'(0) &= \beta(\mathbf{p}_1 - \mathbf{p}_0) \\ \mathbf{q}'(1) &= \beta(\mathbf{p}_3 - \mathbf{p}_2) \end{aligned}$$

It is convenient to use $\beta = 3$ for reasons we will mention below, which gives:

$$\begin{aligned} \mathbf{G}_{Hermite} &= \begin{bmatrix} \mathbf{q}(0) & \mathbf{q}(1) & \mathbf{q}'(0) & \mathbf{q}'(1) \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{p}_0 & \mathbf{p}_1 & \mathbf{p}_2 & \mathbf{p}_3 \end{bmatrix} \begin{bmatrix} 1 & 0 & -3 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & -3 \\ 0 & 1 & 0 & 3 \end{bmatrix}. \end{aligned}$$

Thus,

$$\mathbf{G}_{Hermite}\,\mathbf{B}_{Hermite} = \mathbf{G} \begin{bmatrix} 1 & 0 & -3 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & -3 \\ 0 & 1 & 0 & 3 \end{bmatrix} \mathbf{B}_{Hermite}$$

and so if we define

$$\mathbf{B}_{Bezier} \equiv \begin{bmatrix} 1 & 0 & -3 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & -3 \\ 0 & 1 & 0 & 3 \end{bmatrix} \mathbf{B}_{Hermite}$$

we get

$$\mathbf{q}(t) = \begin{bmatrix} x(t) \\ y(t) \\ z(t) \end{bmatrix} = \mathbf{G}\,\mathbf{B}_{Bezier} \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix}.$$

It turns out that

$$\mathbf{B}_{Bezier} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

The blending functions[16] for Bézier curve are particularly important:

$$\mathbf{B}_{Bezier} \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix} = \begin{bmatrix} (1-t)^3 \\ 3t(1-t)^2 \\ 3t^2(1-t) \\ t^3 \end{bmatrix}$$

The key property of these functions is each is bounded by $[0,1]$ over the interval $t \in [0,1]$, and the four functions add up to 1 for each t. Thus, for all t, the point $\mathbf{q}(t)$ is a "convex combination"[17] of the points $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \cdots, \mathbf{p}_n$. It is therefore contained within the tetrahedron defined by these four

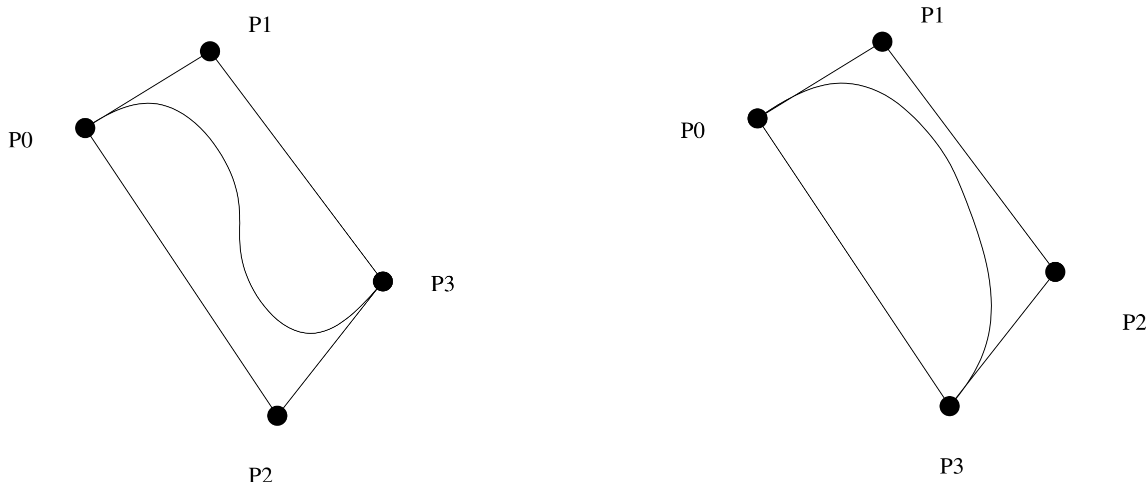
[16] They are called Bernstein polynomials of degree 3. The Bernstein polynomials of degree n are

$$B_{i,n} = \frac{n!}{i!(n-i)!} t^i (1-t)^{n-i}.$$

[17] Given some points $\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_n \in \Re^3$ a convex combination of these points is a point of the form $\alpha_1 \mathbf{p}_1 + \alpha_2 \mathbf{p}_2 + \cdots + \alpha_n \mathbf{p}_n$ where $1 \geq \alpha_i \geq 0$ and $\alpha_1 + \alpha_2 + \cdots + \alpha_n = 1$.

points. This is useful for clipping: when we project the curve into the image plane, the projection of the four P points defines a convex polygon in the image plane that must contain the curve!

The following are two examples of Bézier curves. Note the positions of $\mathbf{p}_2$ and $\mathbf{p}_3$ are swapped.



Catmull-Rom splines

With Bezier splines, half the points serve as serve as anchors through which the spline passes, and half serve as controls which essentially define the tangent vectors. A slightly different approach is to make the spline pass through all the points and to use the points themselves to define the tangents. One example for doing this is the Catmull-Rom spline.

Suppose again we have a sequence of points $\mathbf{p}_0, \cdots \mathbf{p}_{n-1}$ and we want to fit a spline that passes through all of them. For any $\mathbf{p}_i$ and $\mathbf{p}_{i+1}$, define the tangent vectors $\mathbf{p}'_i$ and $\mathbf{p}'_{i+1}$ to be

$$\mathbf{p}'_i = \frac{1}{2}(\mathbf{p}_{i+1} - \mathbf{p}_{i-1})$$

$$\mathbf{p}'_{i+1} = \frac{1}{2}(\mathbf{p}_{i+2} - \mathbf{p}_i) \quad .$$

As an exercise, derive the geometry and blending matrices.

Bicubic Surfaces

We next turn from curves to surfaces. We wish to define a two parameter surface:

$$\mathbf{p}(s,t) = (x(s,t), y(s,t), z(s,t))$$

which maps

$$\mathbf{p} : \Re^2 \to \Re^3$$

and is such that each $x(s,t), y(s,t)$, and $z(s,t)$ is a polynomial in s,t of (maximum) degree 3, and $\mathbf{p}(s,t)$ passes through a given set of 3D data points (see below). The surface $\mathbf{p}$(s,t) is a *parametric*

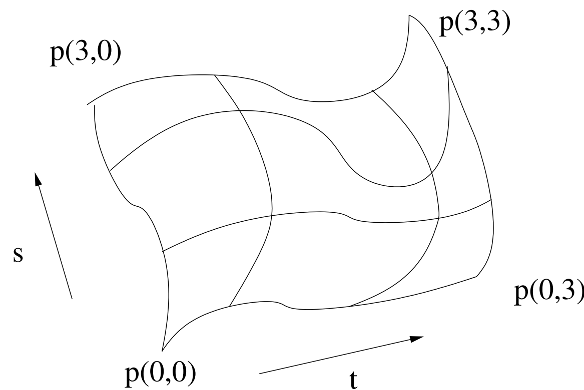
bicubic surface with parameters s and t, in the following sense. For any fixed s, the curve $\mathbf{p}(s,t)$ is a cubic function of t and, for any fixed t, the curve $\mathbf{p}(s,t)$ is a cubic function of s.

The construction of this surface builds on the construction for cubic curves. For any four distinct points in $\Re^3$, we can define a cubic curve that passes through these four points. We use four sets of four points, giving us four cubic curves.

We fit a *bicubic surface* such that it passes through a 4×4 array of arbitrary 3D points

$$\{\mathbf{p}(s,t) : s,t \in \{0,1,2,3\}.$$

These sixteen points are the intersections of the eight curves shown in the following sketch. The eight curves correspond to four values $(0,1,2,3)$ for each of the s,t variables that define the surface. Only the four corner points of the surface patch are labelled in the figure.



To define the surface, we need to specify three functions, $x(s,t)$, $y(s,t)$, and $z(s,t)$. I'll present the construction for $x(s,t)$ only.

For each of the four values of $s \in \{0,1,2,3\}$, we define a cubic curve $x(s,t)$ over $t \in \Re$ by applying the solution from the beginning of the lecture. This gives:

$$x(s,t) = \begin{bmatrix} x(s,0) & x(s,1) & x(s,2) & x(s,3) \end{bmatrix} \mathbf{B} \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix}. \tag{4}$$

Next, take these x component of the four curves $s = 0,1,2,3$ and stack them. This gives the four horizontally oriented curves in the above sketch.

$$\begin{bmatrix} x(0,t) \\ x(1,t) \\ x(2,t) \\ x(3,t) \end{bmatrix} = \begin{bmatrix} x(0,0) & x(0,1) & x(0,2) & x(0,3) \\ x(1,0) & x(1,1) & x(1,2) & x(1,3) \\ x(2,0) & x(2,1) & x(2,2) & x(2,3) \\ x(3,0) & x(3,1) & x(3,2) & x(3,3) \end{bmatrix} \mathbf{B} \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix} \tag{5}$$

The left hand side of Eq. (5) is interesting. For any t, it is a four-tuple. To fit a cubic to this four-tuple (for fixed t), we apply the solution again, but now the curve parameter is s. The illustration above shows the s curves for the particular case of $t \in \{0,1,2,3\}$. These are the four vertically oriented curves in the above sketch.

To obtain the s curves, we need to deal with a minor subtlety in notation. The four-tuple on the left side of (5) is written a column vector whereas previously we wrote our four-tuple control points as row vectors. To use the same form as Eq. 4 and fit a cubic to the four points on the left side of (5), we use the transpose of the left side of (5), namely

$$\begin{bmatrix} x(0,t) & x(1,t) & x(2,t) & x(3,t) \end{bmatrix}$$

and fit a cubic of parameter $s \in \Re$

$$x(s,t) = \begin{bmatrix} x(0,t) & x(1,t) & x(2,t) & x(3,t) \end{bmatrix} \mathbf{B} \begin{bmatrix} s^3 \\ s^2 \\ s \\ 1 \end{bmatrix}. \tag{6}$$

We then take the transpose again (of both sides now) which of course this does nothing to the left side since it is a scalar or 1×1 matrix.

$$x(s,t) = \begin{bmatrix} s^3 & s^2 & s & 1 \end{bmatrix} \mathbf{B}^T \begin{bmatrix} x(0,t) \\ x(1,t) \\ x(2,t) \\ x(3,t) \end{bmatrix} \tag{7}$$

Finally, substitute (5) into the right side of (7), we get:

$$x(s,t) \;=\; \begin{bmatrix} s^3 & s^2 & s & 1 \end{bmatrix} \mathbf{B}^T \begin{bmatrix} x(0,0) & x(0,1) & x(0,2) & x(0,3) \\ x(1,0) & x(1,1) & x(1,2) & x(1,3) \\ x(2,0) & x(2,1) & x(2,2) & x(2,3) \\ x(3,0) & x(3,1) & x(3,2) & x(3,3) \end{bmatrix} \mathbf{B} \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix}.$$

This is our parametric bicubic function. The 4×4 $x(*,*)$ matrix is a geometrix matrix which we can call $\mathbf{G}_x$ and so

$$x(s,t) \;=\; \begin{bmatrix} s^3 & s^2 & s & 1 \end{bmatrix} \mathbf{B}^T \, \mathbf{G}_x \, \mathbf{B} \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix}. \tag{8}$$

We use the same formula to define $y(s,t)$ and $z(s,t)$ via geometry matrices $\mathbf{G}_y$ and $\mathbf{G}_z$.

Surface Normals

As we will see later in the course, it is often very useful to know the surface normal for any (s,t). How could we define this surface normal? Once we have the geometry matrice $\mathbf{G}_x, \mathbf{G}_y, \mathbf{G}_z$, we can compute tangent vectors to the surfaces for any (s,t), namely

$$\frac{\partial}{\partial t} x(s,t) = \begin{bmatrix} s^3 & s^2 & s & 1 \end{bmatrix} \mathbf{B}^T \, \mathbf{G}_x \, \mathbf{B} \begin{bmatrix} 3t^2 \\ 2t \\ 1 \\ 0 \end{bmatrix}.$$

and

$$\frac{\partial}{\partial s} x(s,t) = \begin{bmatrix} 3s^2 & 2s & 1 & 0 \end{bmatrix} \mathbf{B}^T \ \mathbf{G}_x \ \mathbf{B} \begin{bmatrix} t^3 \\ t^2 \\ t \\ 1 \end{bmatrix}$$

The same form is used for the partial derivatives of $y(s,t)$ and $z(s,t)$. Thus, for any (s,t) we have a way to exactly compute:

$$\frac{\partial}{\partial t} \begin{bmatrix} x(s,t) \\ y(s,t) \\ z(s,t) \end{bmatrix} \quad \text{and} \quad \frac{\partial}{\partial s} \begin{bmatrix} x(s,t) \\ y(s,t) \\ z(s,t) \end{bmatrix}.$$

From Calculus III (and intuition ?), these two partial derivatives are both tangent to the surface (and are not identical). SInce the surface normal must be perpendicular to both of these tangent vectors, the surface normal must be parallel to the cross product of these two partial derivatives.

$$\frac{\partial}{\partial s}\mathbf{p}(s,t) \times \frac{\partial}{\partial t}\mathbf{p}(s,t)$$

We will use this cross product method for defining the surface normal in a later lecture, when we discuss bump mapping, although there we won't be assuming necessarily that $\mathbf{p}(s,t)$ is a bicubic.

Meshes

You all know what vertices and edges are in the context of graphs. In computer graphics, a *mesh* is a specify type of graph. Vertices are points in R^n where $n = 2$ or 3. Edges are line segments. Polygons are cycles in the graph of length greater than 2, specifically, minimal cycles with non-repeating vertices (except the first and last). Polygons (also known as faces) are assumed to be planar.

To keep things simple, we'll assume that each edge of our mesh belongs to at most two triangles. There may be edges that belong to only one triangle. Such edges are said to be on the boundary of the surface. Restricting ourselves to edges that belong to at most two triangles means that we can talk about one surface. If an edge were to belong to more than two triangles, then we would be talking about multiple surfaces that are stuck together. (This is allowed in general, but we're not dealing with it.) Technically, one uses the term *manifold mesh* for the cases we are considering. (The term "manifold" is not something you are typically exposed to as an undergraduate. You would learn about it if you studied differential geometry.)

We'll typically be talking about meshes made out triangles today, although in general one might like to have meshes made out of quads (each polygon has four edges) or be more general and have a mix of triangles, quads, etc.

Polyhedra

A *polyhedron* is a polygonal mesh that is the boundary of some 3D volume. A *regular polyhedron* is a polyhedron whose faces are congruent i.e. identical except for a rotation and translation. There exist many regular polyhedra. For example, you may be familiar with the convex regular polyhedra (called the Platonic solids): tetrahedron, cube, octahedron, dodecahedron, icosahedron.

Some polyhedra such as a torus or a coffee cup model have *holes*. For the coffee cup, the hole is in the handle. For a polyhedron with no *holes* the number of vertices, faces, and edges satisfies Euler's formula:

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

Verify this formula for a tetrahedron (pyramid) and cube. For objects with one hole,

$$V + F - E = 0$$

and for objects with n holes,

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

ASIDE: the number of holes is called the *genus*.

Data structures for polygons

A typical way to represent a polygon mesh is to use tables. (The term table is very general here. It could be an array, or a hashtable, etc.)

- *vertex table*: each entry points to a point in $\Re^3$, $v_i = (x_i, y_i, z_i)$
- *edge table*: each entry points to a pair of vertices, $e_j = (v_k, v_l)$
- *polygon table (face table)*: each entry points to a list of vertices that define the polygon.

This is sometimes called a *polygon soup* representation of a mesh, since it says nothing about which polygons are connected to which.

It would be nice to have a richer representation of the mesh, since this would allow you to answer certain questions about connectivity. For example, suppose you are given a face and you wished to know which other faces are connected to that face. The above tables don't allow you to do that efficiently. If, however, you were to augment say the edge table so that it had a list of the faces that the edge belonged to, then you could answer this query. You could examine each edge of the given face (via the list of vertices of that face, stored in the face table) and then lookup in the edge table which other face that edge belongs to.

Similarly, you might also wish to represent the set of polygons that a vertex belongs to. This could be useful, for example, in an animation in which you are moving the vertex and you want to know which polygons are affected. For each vertex in the vertex table, we could have a list of pointers to the polygon (or edge) table.

There is a tradeoff, of course. The more information that is made explicit, the easier it is to reason about the connectivity of the mesh and the neighborhood relationships, but the larger is the representation.

[ASIDE: In OpenGL, surface meshes are typically represented as a polygon soup, but there are a few specialized mesh structures you can use as well, called strips or fans, for example, `GL_TRIANGLE_STRIP` or `GL_TRIANGLE_FAN`. This allows you to use slightly less memory.]

Parameterizing the points on a mesh surface

A mesh is a 2D surface in R^3. How to parameterize this surface ? Specifically, we know the vertices and edges of the mesh, but what about the points inside the faces? How do we refernence these positions? These points lie on a plane that contains the face. We will show how to reference points on the face by a linear combination of the vertices of the face.

Before doing so, we ask a more basic question. A triangle (v_1, v_2, v_3) is a 2D surface in R^3. How to parameterize points in the triangle ? Assume v_1, v_2, v_3 are linearly independent (so the three points don't lie on a line). Consider set of points:

$$\{av_1 + bv_2 + cv_3\}$$

where a, b, c are real numbers. This set spans R^3. If we restrict this set to

$$a + b + c = 1$$

then we get a plane in R^3 that contains the three vertices v_1, v_2, v_3, namely (a, b, c) is $(1, 0, 0), (0, 1, 0), (0, 0, 1)$ respectively. We can further restrict to points within the triangle by requiring

$$a = 1 - (b + c)$$

and

$$0 <= a, b, c <= 1.$$

These are called the "convex combinations" of v_1, v_2, v_3. Note that this constraint can be derived by saying we are writing a vertex in the triangle as

$$v = v_1 + b(v_2 - v_1) + c(v_3 - v_1)$$

and grouping the terms that involve v_1.

[ASIDE: In mathematics, a triangle in 3D mesh is called a "2-simplex" and a, b, c are called "barycentric coordinates of points in the triangle.]

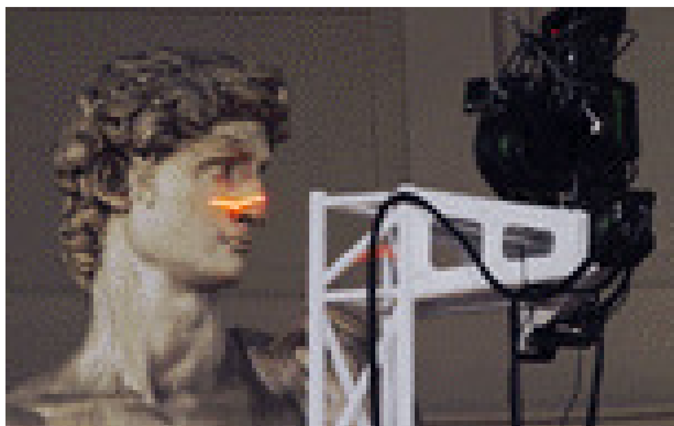
Finally, suppose a mesh has V vertices and F triangles (faces). Consider R^V, with one dimension for each vertex. For each triangle in the mesh, we have a corresponding 2-simplex in R^V. These F simplices define a mesh in R^V, whose points are in 1-1 correspondence with the mesh in R^3.

For example, suppose that v_1, v_2, v_3 is a triangle in the mesh. Then a point near the center of the triangle could be paramterized by say $(.3, .4, .3, 0, 0, 0, 0, \dots) \in \Re^V$.

Where do mesh models come from ?

You can build mesh models by hand. For example, the original Utah teapot that is provided in OpenGL was designed by a computer graphics researcher named Martin Newell in the mid-1970's. See story: `http://www.sjbaker.org/wiki/index.php?title=The_History_of_The_Teapot`

These days, one typically uses a 3D imaging device which yields a set of vertices (x, y, z). Often these devices come with software that also deliver a mesh in a standard format. See `http://graphics.stanford.edu/data/3Dscanrep/` for several recent large polygonal meshes that were popular to use in computer graphics research about 10 years ago, and a discussion of how they were obtained, for example, the "Stanford bunny" (shown below) and "Michelangelo's David".



Range scanning devices can be based on light (laser, photography) or lower frequency waves (radar). The models mentioned above were obtained using laser range finders. The scanning devices often use computer vision techniques. In a nutshell, these techniques combine images taken by two cameras, and use the slight differences in the image to infer the depth at each point. The basic principle is similar to how your brain combines the images seen by your left and right eyes to perceive depth. Typically these system also project a pattern onto the surface (using a laser or conventional light projector) to make the matching task easier. Details are omitted since this is more a topic for a computer vision course than a computer graphics course.

What's important is that the scanning devices typically define a depth map $z(x, y)$ on a regular grid of image coordinates (x, y). For each 3D position of the scanner/camera, you get a set of points $(x, y, z(x, y))$ in 3D space. There is a natural triangulation on these points, namely it is a height function defined over an (x, y) grid. (See leftmost figure in the "terrain" figures below.)

For solid surfaces, and especially surfaces with holes, one needs to make many such scans from different positions of the scanner i.e. you need to scan the back and front. This requires piecing together the scans. It is a non-trivial computational problem. Details are again omitted (since it is more computer vision than graphics).

The models require alot of data! For example, if we were to represent an object by having a vertex spaced every 1 mm (say), then we would need 1,000 vertices for a 1 meter curve on the surface, and so we would need about one million vertices for a 1 m^2 surface area. Several of the models in the Stanford database are of that size. The model of the entire David statue, which was scanned at about .25 mm resolution, has about two billion triangles! Many different scans were needed especially to capture the details within the concavities and these scans needed to be pieced together.

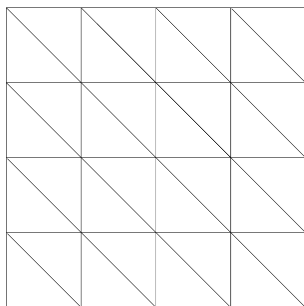
If you are interested in learning more, see the slides for links to a few examples of current state of the art, including the light stage work by Paul Debevec.

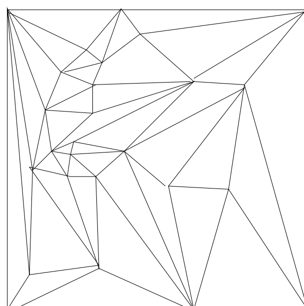
Terrains

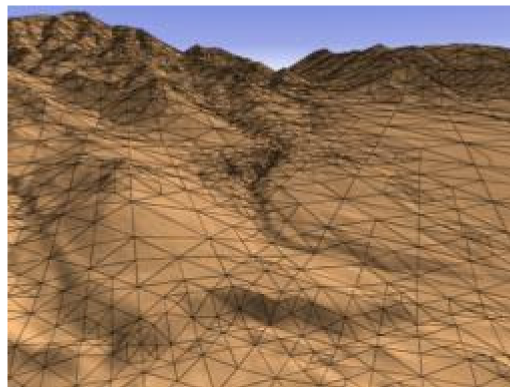
A *terrain* (or height field) is a polygonal mesh such that one coordinate (say z) is a function of the other two coordinates (say (x, y)) i.e. there is at most one z value for each x, y value. The points on the terrain can be written $(x, y, z(x, y))$.

Terrains can be defined over a *regular* polygonal mesh in the (x, y) plane e.g. integer coordinates, or over a *non-regular* polygonal mesh in the (x, y) plane. (See below.) Terrains are popular and useful, since all scenes have a ground and in most natural scenes the ground is non-planar. See the example [18] below on the right of a mesh, which was texture mapped and rendered. (Later in the course we will say what we mean by "texture map".)

Note that scanning devices mention above often compute terrain meshes even though the objects being scanned are not themselves terrains, since they have front sides and back sides and holes. This is why multiple scans of each object are needed and the terrain meshes must be combined into non-terrain meshes.


regular (x, y)


non-regular (x, y)


Example

[18] The image was taken without permission from Peter Lindstrom's Georgia Tech web page.

Level of Detail (LOD)

We use the term *resolution* to refer to the size of polygons in the model, for example, the length of the shortest edge. As we discussed last class in the context of fractals, if we limit ourselves to a given smallest edge size (say 1 mm) then there may be details in the surfaces geometry at a smaller size that we ignore when modelling the object.

The resolution at which an object should be displayed in some image might be much coarser then the resolution of the model in the database. For example, if a mudpile or rockface were viewed from up close then each triangle in the model might project to multiple pixels and so the highest (finest) resolution of the mesh would be needed to draw the object accurately. But if the same object were seen from a great distance, then each triangle might occupy only a small fraction of a pixel and so the fine resolution of the model would not be needed. One often refers to the *level of detail* (LOD) of a model when describing the resolution that is being represented.

One approach to having multiple levels of detail is to *precompute* several models that differ in resolution. For example, a low resolution model might have 10^2 polygons, a medium resolution model might have 10^4 polygons, and a high resolution model might have 10^6 polygons, etc.

How do you choose which model to use at any time? One idea is to consider, when drawing an object, roughly how many pixels the object would occupy in the image (using a bounding box for the object, say) and then decide which resolution model to use based on this number of pixels.

For example, suppose you wanted to use the Stanford model of Michelangelo's David in an image, such that whole David is viewed from a distance of 20 meters and occupies only a small fraction of the image. Obviously it would be wasteful to process every one of the two billion polygons in the model. Instead, you would use a lower resolution model.

A second approach is to represent the model using a data structure that allows you to vary the number of polygons by simplifying the polygonal mesh in a near-continuous way (see below). For example, you could let the resolution of the model vary across the object in a *viewpoint dependent* way. In an outdoor scene, the part of the ground that is close to the camera would need a higher LOD, whereas for the part that is further from the camera a lower LOD would suffice. Thus, we could use a coarse resolution for distant parts of the scene and a fine resolution for nearby parts of the scene. For the example of the terrain the previous page, the surfaces in the upper part of the image which are way off in the distance would not need to be represented as finely (in world coordinates i.e. $\Re^3$) as the polygons at the bottom of the image which are relatively close to the camera.

Mesh Simplification

How do we obtain models with different level of detail ? For example, suppose we are given a triangular mesh having a large number of triangles. From this high LOD mesh, we would like to compute a lower level of detail mesh. This process is called *mesh simplification.*

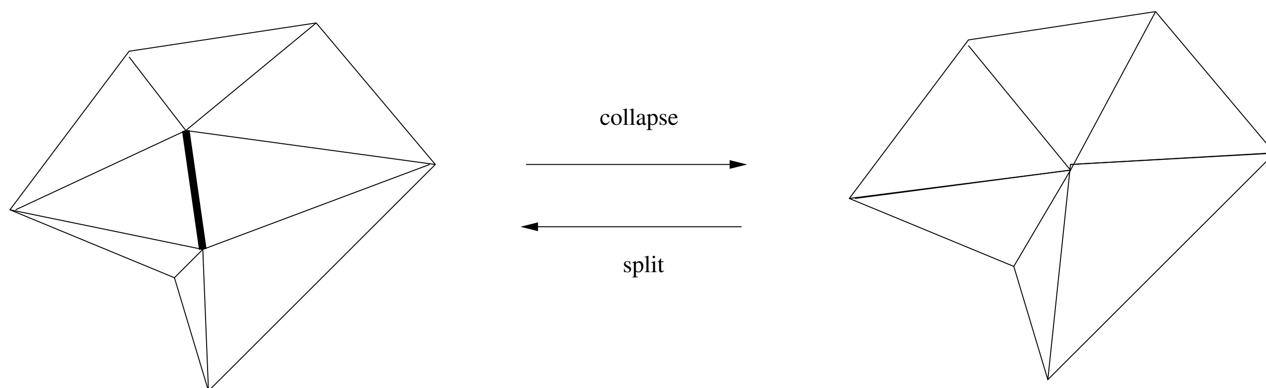
Edge collapse and vertex split

Suppose we are given a triangulated mesh with V vertices, and we wish to simpify it to mesh having $V - 1$ vertices. As shown in the example below, we can collapse one of the edges (defined by a pair of vertices) to a single vertex. There are two ways to do this. One way is to delete one of the

vertices of this edge. The other is to replace both vertices of the edge by one new vertex. Either way, a new triangulation must be computed (locally) to take account of the fact that vertices have been deleted or replaced.

Notice that when you do an edge collapse, you lose two triangles, one vertex, and three edges, and so $V + F - E$ is unchanged.

Also notice that an edge collapse is invertible. To go the other direction, you can split a vertex into two (essentially adding a new vertex) and join the two vertices by an edge. The vertex you are splitting may belong to several edges, and so when you split the vertex you need to decide whether these edges remain with the vertex or whether they instead join the new vertex.



Subtle issues can arise. For example, faces can be flipped. In the first example below, the boldface edge is collapsed and all edges that are connected to the upper vertex are deleted and/or replaced by an edge to the lower vertex. Consider one such new edge, namely the dotted one shown on the right. This dotted edge in fact belongs to *two* triangles: one whose normal points out of the page, and a second whose normal points into the page. The latter is thus flipped. This is not good. For example, if the V vertex surface were a terrain then the $V - 1$ vertex surface should also be a terrain.

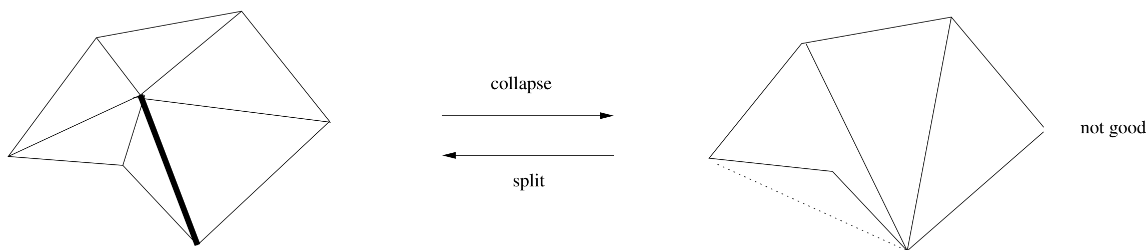
In the second example below, a different edge is collapsed and no such problem arises.

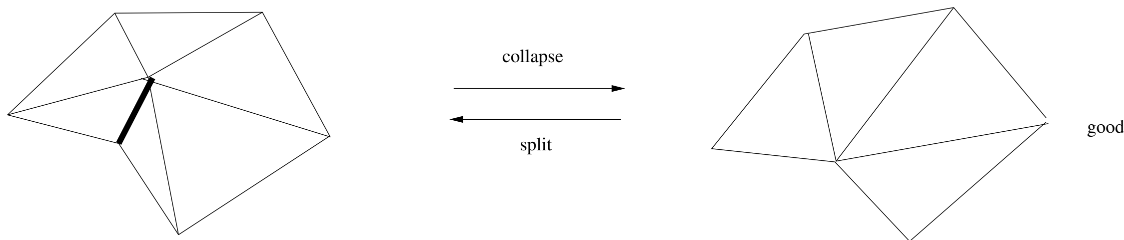
You can organize edge collapses using a forest of binary trees. The leaves of the trees are the vertices of the original mesh. Each edge collapse creates a sibling relationship between two nodes in the tree, such that the parent of these siblings is the vertex that the pair of vertices collapses to.

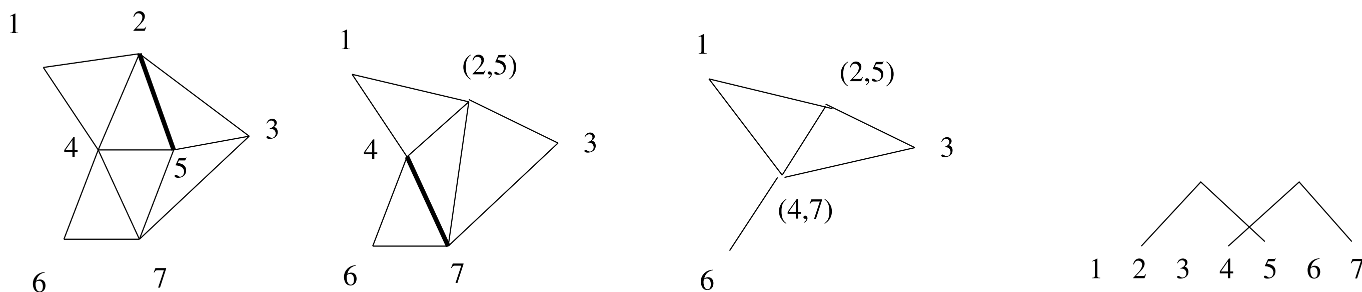
Start with the entire high LOD model, such that each vertex is its own tree (a trivial tree consisting of a root only). Then, if two vertices are connected by an edge and you choose[19] to collapse this edge, then a parent node is defined for these two vertices and the trees for these two vertices are merged. The representative vertex for the new parent node is either one of its children vertices, or some new vertex that replaces the two children vertices. This information must be stored at the parent vertex, so that it is possible later to *split* the parent vertex, that is, to recover the children vertices (and the edge between them). As many edges can be collapsed as desired, i.e. one can simplify the mesh from V vertices down to any V' vertices, where $V' < V$.

The example below shows the basic idea. Two edge collapses are shown. Although the rightmost "triangulation" doesn't appear to be a triangulation, you must keep in mind that only part of the mesh is shown. In particular, verticles 1, 3, 6 are each connected to other vertices not shown here.

[19] I am not specifying how you decide which edge to collapse.







Such a data structure can be used to choose the level of detail at render time. One can begin with a greatly simplified mesh, whose vertices are the roots of the trees in the forest. Once can then expand the mesh by "splitting" (or subdividing) vertices until the resolution of each part of the mesh is fine enough based on some criterion. [ASIDE: mesh simplification and LOD choice is a highly technical area in computer graphics, and my goal here is just to give you a feel for the sort of problems people solve. For an example, see `http://research.microsoft.com/en-us/um/people/hoppe/proj/pm/`]

Subdivision surfaces

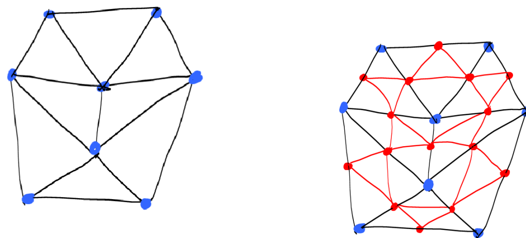
Last class we discussed how to interpolate between a sequence (or grid) of vertices, in order to define a smooth curve (or surface). We use cubics (and bicubics). There are many very nice methods such as those for modelling curves and surfaces in this way. However, there are limitations, for example, they have strong restrictions on the sampling of the points. Bicubics typically require that one piece together 4×4 grids. That was fine for Martin Newell to make the Utah teapot. But it typically non-trivial to make a large number of 4×4 grids from the scans that one obtains of objects such human bodies.

A different set of techniques, known as surface subdivision, were invented long ago which relaxed

these requirements. I discussed one of these in class, namely Loop subdivision (which was named after Charles Loop, not `for/while` Loop). This scheme assumes the surface has been triangulated.

The general idea of subdivision is to make a smooth mesh with many vertices from a coarse mesh that has relatively few vertices. One iteratively "subdivides" the mesh, introducing more (smaller) triangles, such that the vertices of the triangles define a "smoothed" version of the surface at the higher level.

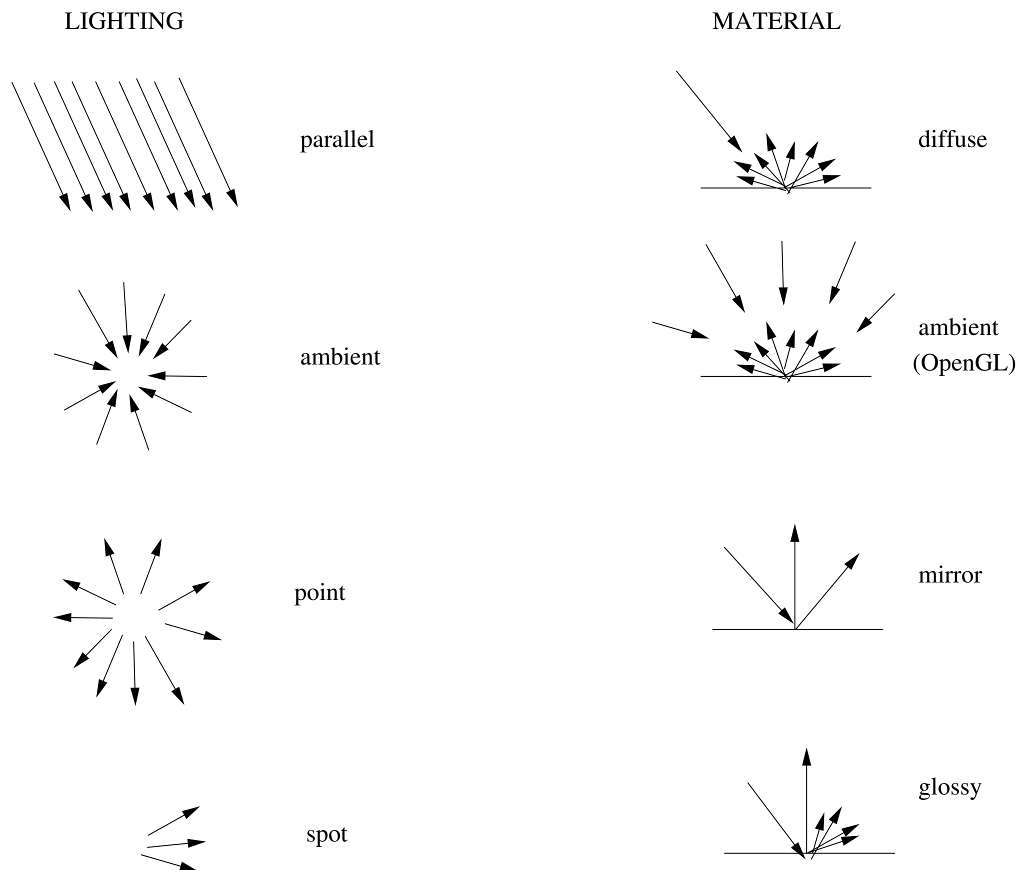
In Loop subdivision, every triangle is broken down into 4 subtriangles. There is a unique way to do this. Given a mesh (such as on the left below), insert a new vertex on the midpoint of each edge and then make three new edges inside each face, namely joining the three new vertices (see red vertices and edges in figure on the right).



To make the new surface smoother than the original one, the vertices need to be moved. For the new (red) vertices that we added, there are four natural vertices to consider when choosing the position of the new vertex. Each new (red) vertex sits on an old (black) edge, and there are the two old (blue) vertices that define this edge. There are also two old (blue) vertices opposite this edge in the face of the two old triangles to which this edge belongs. Loop lets the weights be each $\frac{3}{8}$ for the adjacent vertices, and he lets the weights be each $\frac{1}{4}$ for the opposite vertices.

We also move the original vertices. Each of the original vertices is connected to k other vertices in a ring. (Hence we sometimes need to know which are the vertices that v is connected to be an edge.) We want to give the original vertex'es position some weight. We also want to give some weight (say β) to all the other vertices that surround it. If we give each of the surrounding vertices a weight β, then the total weight is $k\beta$ which means the weight of the original vertex is $1 - n\beta$. What is β ? There is no strict rule. One simple scheme is to let the original point keep about half its weight ($1 - k\beta = \frac{1}{2}$) so $\beta = \frac{2}{k}$.

We now move to third part of the course where we will be concerned mostly with what intensity values to put at each pixel in an image. We will begin with a few simple models of lighting and surface reflectance. I discussed these qualitatively, and then gave a more detailed quantitative description of the model.



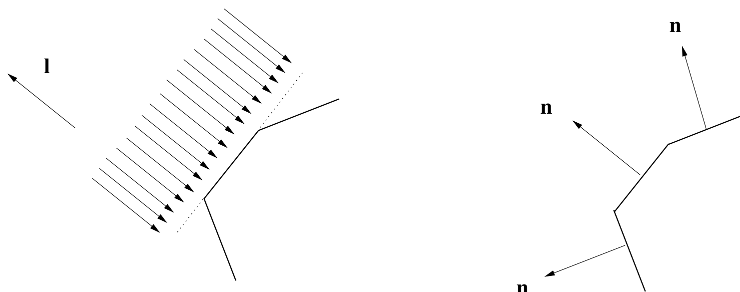
Light sources and illumination

You know what light sources are you probably have not thought about how to model them. Think about the sun vs. a lamp vs. a spotlight vs. the sky. All of these are light sources, but each illuminates the scene in its own way.

Sunlight (Parallel source, "point source at infinity")

Typical lighting models account for the directionality of lighting, namely that light travels along rays. One simple model of directional lighting is the *parallel source*. According to this model, any light ray that arrives at a surface point $\mathbf{x}$ comes from a single direction. We typically say $\mathbf{l}$ is the direction *to* the source and often it is a unit vector. We can think of this as a "sunny day" model. Since the sun is far away, all rays from the sun that reach the earth can be treated as parallel to each other.

The amount of light arriving *per unit surface area* at a point $\mathbf{x}$ and from direction $\mathbf{l}$ depends on the surface normal $\mathbf{n}(\mathbf{x})$. To understand this, think of the parallel rays from the source as having a certain density. The more intense the light source, the greater the density of the rays. The number of rays that reach a unit area patch of surface is proportional to $\mathbf{n}(\mathbf{x}) \cdot \mathbf{l}$, namely to the cosine of the angle between the surface normal and the light source. To understand this, consider the sketch below. The parallel vectors indicate the rays arriving at the surface. There are three surfaces (polygons) of identical size. These are indicated by 1D cross-sections only, but you should imagine they are 2D surface patches with the same area. For the two surfaces that are tilted away from the light source, the number of rays "caught" by the surface is lower than the number caught by the surface that faces the light source directly. The number of rays caught depends on $\mathbf{n}(\mathbf{x}) \cdot \mathbf{l}$.



lightbulb (local point source)

The parallel source is a good model for sunlight but it is not a good model for sources that are positioned *in* the scene. For such lights it is common to use a *local point light source* model. According to this model, light is emitted from a point $\mathbf{x}_l \in \Re^3$ and radiates outwards from that point. I'll first describe a model in which light radiates uniformly out in all directions. Then I'll describe a model in which the directional intensity is not uniform (spotlight). *The order of presentation here differs from in the slides, but covers the same main ideas.*

According to the point source model, the amount of light reaching a surface of unit size depends on three factors:

- the distance between the light source and the surface; If the surface is moved further away from the source, then it will receive less light and hence will become darker. The reason that distant surfaces receive less light is that the light radiating from a source is spread out over a sphere whose area increases with distance squared.
- the fraction of light that the source emits in the direction of the surface: Some sources emit light in certain directions more than others. Car headlights, for example, are designed to throw light on the road ahead, rather than off to the side.
- the surface normal $\mathbf{n}(\mathbf{x})$: If the surface normal points directly at the light source then it will receive more light than if it is oriented obliquely to the source. The reasoning is the same as we argued above for the parallel source;

Consider a point light source located at position $\mathbf{x_l}$ in the scene. A scalar function $I_l(\mathbf{x})$ captures the dependence of the strength of the illumination (the density of rays) at $\mathbf{x}$ as a function of its

position relative to the source. This includes both a distance and direction effect. The distance effect is naturally modelled as a function of distance r between $\mathbf{x}$ and $\mathbf{x}_l$,

$$r = \| \mathbf{x} - \mathbf{x}_l \| .$$

What is this function? As light radiates from a point source, its energy is spread out on a sphere which has area proportional to r^2. The physics suggests that the intensity should fall off as $\frac{1}{r^2}$, but this assumes the source is a single point. Real sources are not points. Rather, they have a finite volume. This suggests a more general model of the strength of the source as a function of scene position, namely

$$I_l(\mathbf{x}) = \frac{1}{ar^2 + br + c}$$

where r is now the distance to some (arbitrary) point within the source volume. Have constants $c > 0$ (and $a, b > 0$ too) prevents the intensity from going to infinity as $r \rightarrow 0$. OpenGL allows such a general falloff function as we will see.

The above function only specifies the strength of the source, but does not specify the direction of the source. To model the directional effect, we define a unit vector $\mathbf{l}(\mathbf{x})$ at each $\mathbf{x}$ which indicates the direction from $\mathbf{x}$ to the point light source $\mathbf{x}_l$:

$$\mathbf{l}(\mathbf{x}) = \frac{\mathbf{x}_l - \mathbf{x}}{\| \mathbf{x}_l - \mathbf{x} \|}.$$

We can then weight this unit vector (direction) with the distance effect:

$$I_l(\mathbf{x})\ \mathbf{l}(\mathbf{x}) = \frac{1}{(ar^2 + br + c)} \frac{\mathbf{x}_l - \mathbf{x}}{\| \mathbf{x}_l - \mathbf{x} \|}$$

which specifies the illumination at any point $\mathbf{x}$.

Spotlight

Many lights in a scene do not illuminate equally in all directions but rather are designed to send most of their illumination in particular directions. A spotlight or car headlight are two examples. Even a typical desklamp is designed to have non-uniform directional properties – the light is supposed to illuminate the desk, rather than shining directly into one's eyes.

To capture this directionality effect, we can modify the scalar $I_l(\mathbf{x})$ of the point source by consider the direction here as well. For example, suppose the light source at $\mathbf{x}_l$ sends out its peak illumination in direction $\mathbf{l_{peak}}$. One common and simple model for the strength of the source as function of direction (relative ot the peak direction) is as follows:

$$(\mathbf{l_{peak}} \cdot \frac{\mathbf{x} - \mathbf{x_l}}{\| \mathbf{x} - \mathbf{x_l} \|})^{\text{spread}}$$

where the dot product decreases away from the peak direction, and spread controls how fast the falloff is with direction. A laser beam would have a very high spread, whereas a spread close to 0 would have no directional effect. Note that this is *not* a physics-based model. Rather, it is just a simple qualitative model that is meant to capture the fact that some sources are more directional than others.

We can combine this directional effect with the distance effect discussed earlier to have a model of a directional local source such as a spotlight:

$$I_l(\mathbf{x}) \; \mathbf{l}(\mathbf{x}) \;=\; \frac{1}{(ar^2+br+c)} \;\; \frac{\mathbf{x}_l - \mathbf{x}}{\parallel \mathbf{x}_l - \mathbf{x} \parallel} \;\; (\mathbf{l_{peak}} \cdot \frac{\mathbf{x} - \mathbf{x_l}}{\parallel \mathbf{x} - \mathbf{x_l} \parallel})^{\text{spread}}$$

In this model, $\mathbf{l(x)}$ is a unit vector defines at position $\mathbf{x}$ which denotes the direction from which light is coming. The scalar $I_l(\mathbf{x})$ captures the strength of this source. [ASIDE: for those of you with more advanced calculus background, the above is a *vector field*, namely a vector defined at each position $\mathbf{x}$ in space.]

Surface reflectance

The above discussion was concerned with how much light *arrives* at a surface point $\mathbf{x}$ per unit surface area. To model the intensity of light *reflected* from a surface point $\mathbf{x}$, we must specify not just the lighting, but also the surface material. Let's look at the classic models.

Diffuse (matte, Lambertian)

The first model is diffuse (or matte or *Lambertian*[20]) reflectance. This model states that the intensity of light reflected from a surface point $\mathbf{x}$ does not depend on the position from which the point is viewed. This will more clear by the end of the lecture.

For a parallel light source in direction $\mathbf{l}$, the intensity of light reflected from a point $\mathbf{x}$ on a Lambertian surface can be modelled as

$$I_{diffuse}(\mathbf{x}) = I_l \; k_d(\mathbf{x}) \; max(\mathbf{n} \cdot \mathbf{l}, 0)$$

where I_l is the intensity of the source and k_d is the fraction of light reflected by the diffuse component. The max operation is needed because surfaces that face away from the light do not receive direct illumination. The cosine of the angle is negative for such surfaces.

For a local point light source (see top of this page), the intensity of light reflected from a point $\mathbf{x}$ on a Lambertian surface can be modelled as

$$I_{diffuse}(\mathbf{x}) = I_l(\mathbf{x}) \; k_d(\mathbf{x}) \; max(\mathbf{n}(\mathbf{x}) \cdot \mathbf{l}(\mathbf{x}), 0)$$

Ambient light

In real scenes, surfaces that face away from a point light source or a parallel light source still receive some illumination, although this is only indirectly via reflections from other surfaces in the scene. We do not want surfaces that face away from a light source to appear black in our image. To avoid this, we add on a component called the "ambient" component.

$$I_{ambient}(\mathbf{x}) \;=\; I_l k_{ambient}(x).$$

I'll mention more about this later when I discuss OpenGL and in an upcoming lecture (on shadows and interreflections).

[20] Johann Lambert lived in the 18th century.

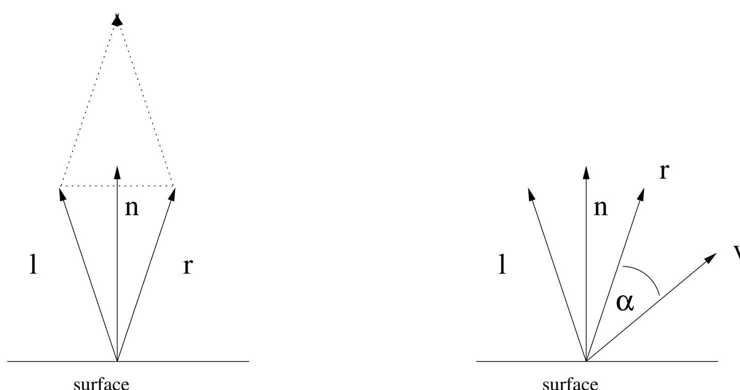
Mirror reflectance

A mirror surface can be modelled as follows. Assume that light source is in direction $\mathbf{l}$ which is a unit vector. To keep the notation simple, we do not distinguish whether the light source is parallel or a point here, and we just write $\mathbf{l}$ rather than $\mathbf{l}(\mathbf{x})$. The light that is reflected from the mirror obeys the rule: "the angle of incidence equals the angle of reflection". Call the unit vector that is the direction of reflection $\mathbf{r}$. The vectors $\mathbf{n}$, $\mathbf{l}$, $\mathbf{r}$ all lie in a plane. Using simple geometry as illustrated in the figure below on the left, one can see that

$$\mathbf{l} + \mathbf{r} = 2(\mathbf{n} \cdot \mathbf{l})\ \mathbf{n}.$$

Thus,

$$\mathbf{r} = 2(\mathbf{n} \cdot \mathbf{l})\mathbf{n} - \mathbf{l}$$



Glossy/shiny/specular surfaces - the Phong and Blinn-Phong models

In between Lambertian surfaces and mirror surfaces are those that we would call (more or less) shiny. Examples include polished surfaces such as floors and apples, oily surfaces such as skin, and plastic surfaces. Such surfaces do not obey the Lambertian reflectance model because the intensity of the light reflected from a surface point depends on where the camera is placed, i.e. from where the surface point is viewed. An example is the "highlight" on an apple or floor tile. As you move, the location of the highlight slides along the surface.

Shiny surfaces are not perfect mirrors. Rather, the light reflected from these surfaces is concentrated in a *cone* of directions containing the mirror direction $\mathbf{r}$. The reflected light will be brightest if it is viewed along the ray in the mirror reflection direction. The intensity will fall off continuously if it viewed along other rays that begin at $\mathbf{x}$ but have slightly different reflection direction.

One popular model of shiny surfaces that roughly captures the above phenomena is the *Phong lighting model*. Phong's model takes the Lambertian model and adds a shiny component, also called *specular* component. For a given surface point, the intensity of the specular component depends on how "close" the mirror reflection direction $\mathbf{r}$ is to the viewing direction $\mathbf{v}$, that is, the direction from $\mathbf{x}$ to the camera. (Notice that, for the first time, we need to specify explicitly where the camera is. The unit vector $\mathbf{v}$ points from the surface point to the camera.) See the figure above right.

Phong's model considers angular distance between the unit vectors **r** and **v**. It says that the intensity of the specular component of the light reflected in direction **v** is proportional to the cosine of the angle between **r** and **v** and raises this value to some power, which we can think of as the "shininess". I write the glossy component of the Phong lighting model:

$$I_{specular}(\mathbf{x}) \ = \ I_s \ k_s \ (\mathbf{r} \cdot \mathbf{v})^e.$$

where

- I_s is a scalar that describes the intensity of the light source. In terms of the physics, this should be the same scalar as was used in the diffuse reflection component model, since the lighting is not affected when you swap from a diffuse to a shiny surface! [ASIDE: As we will see later, OpenGL allows you to use different lighting properties for the diffuse versus glossy components. This makes no physical sense but graphics people aren't bothered by it.]
- k_s is proportional to the fraction of light reflected in the glossy/specular manner;
- the exponent e is the sharpness of the specular component. e.g. If e is large (100 or more) then the surface will be more mirror-like, whereas if e is small, then the specular reflection will be spread out over a wide range of directions. Note that this is a similar mathematical model to what is used for spotlights. Again, it is *not* a physics-based model, but rather just a qualitative model that describes a directional effect.

Note that each of the above $I_s, k_s, \mathbf{r}$, and $\mathbf{v}$ can depend on $\mathbf{x}$.

Consider a point $\mathbf{x}$ on the surface such that

$$\mathbf{v}(\mathbf{x}) \approx \mathbf{r}(\mathbf{x}),$$

that is, the mirror reflection direction is about the same as the direction to the camera. According to the Phong model, such points produce a peak intensity value since $\mathbf{v} \cdot \mathbf{r} \approx 1$. Such intensity peaks are called *highlights*. You can find highlights on many shiny surfaces, such as polished fruit (apples), ceramic pots, plastic toys, etc.

OpenGL uses a slightly different model of specular reflection. Define the *halfway* vector at a surface point:

$$\mathbf{H} = \frac{\mathbf{l} + \mathbf{v}}{\mid \mathbf{l} + \mathbf{v} \mid}$$

where $\mathbf{l}$ is the direction of the lighting at this point. If $\mathbf{H} = \mathbf{n}$, then $\mathbf{r}$ will be the mirror reflection direction and the peak of the highlight should appear at the surface point. As $\mathbf{n}$ deviates from $\mathbf{H}$, the mirror reflection direction $\mathbf{r}$ will deviate more from the viewer direction $\mathbf{v}$. The Blinn-Phong is similar to the Phong model and can be written,

$$I_{specular}(\mathbf{x}) \ = \ I_s \ k_s \ (\mathbf{H} \cdot \mathbf{n})^{\text{shininess}}.$$

See Exercises for the advantage of Blinn-Phong.

Lighting and Material in OpenGL

In the lecture slides, I gave an overview of how light sources and materials are defined in OpenGL. In a nutshell, you can define light sources which can be parallel, or uniform local sources, or spotlights. You define a light with

```
glLightf( light, parameterName, parameter )
```

and the parameter names of the light are:

```
GL_AMBIENT                      GL_SPOT_DIRECTION
GL_DIFFUSE                      GL_SPOT_EXPONENT
GL_SPECULAR                     GL_SPOT_CUTOFF
GL_POSITION                     GL_ CONSTANT_ATTENUATION
                                GL_LINEAR_ATTENUATION
                                GL_QUADRATIC_ATTENUATION
```

OpenGL allows you to define the RGB color of the light as values between 0 and 1. Weirdly, you can define different colors for the ambient, diffuse, and specular components. I say it is weirdly because lights should be independent of the surfaces they illuminate. It is the surfaces that are diffuse or specular, not the lights. Anyhow, what about the surfaces?

Surfaces are defined by:

```
glMaterialfv( face, parameterName, parameters )
```

where `face` specifies whether the front or back (or both) are drawn, and the parameter names are as show below and ___ is a RGB or RGBA tuple. (I'll explain A later in the course).

```
glMaterial(GL_FRONT, GL_AMBIENT, ___ )
glMaterial(GL_FRONT, GL_DIFFUSE, ___ )
glMaterial(GL_FRONT, GL_SPECULAR, ___ )
glMaterial(GL_FRONT, GL_SHININESS, ___)
```

We can relate the above to the models presented earlier. In the first part of the lecture, we talked about $I(\mathbf{x})$ but really we had one of these functions for each of RGB. In OpenGL, the RGB values are determined by :

$$I(\mathbf{x}) = I_{diffuse}(\mathbf{x}) + I_{specular}(\mathbf{x}) + I_{ambient}(\mathbf{x})$$

where the range is between 0 and 1.

OpenGL does not allow for mirror surfaces, in the general sense I discussed above, although it does provide something more specific, namely environment mapping. I'll describe what this is in an upcoming lecture.

Finally, in the slides, I briefly discussed material modelling beyond OpenGL. I mentioned BRDF's (bidirectional reflection distribution functions), subsurface scattering, etc. Do have a look, though I don't expect you to know this in any detail since I only showed pictures but didn't go into the models.

Shading polygons

At the end of lecture 6, I sketched out how to rasterize a polygon. One of the steps there was to choose the colors at each point inside the polygon. The most straightforward way to fill a polygon is to consider, for each pixel (x, y) in that polygon, what is the surface point (x, y, z) in the scene that projects to that pixel. If we knew the surface normal at this scene point and the lighting and the material of the surface at this scene point, then we could choose the intensity based on the models we discussed above. While this approach sounds straightforward, there are subtleties that arise. There are also several shortcuts that we can take to make the choice of colors more efficient.

Surface normals on polygons: OpenGL

In OpenGL, if you define a polygon but you don't specify the surface normal, then OpenGL will automatically compute the normal by fitting a plane and taking the normal to the plane. It is very useful, however, to explicitly define the surface normals. This can be done at each vertex, for example:

```
glBegin(GL_QUAD)
glNormal3f( _, _, _ )
glVertex3f( _, _, _ )
glNormal3f( _, _, _ )
glVertex3f( _, _, _ )
glNormal3f( _, _, _ )
glVertex3f( _, _, _ )
glNormal3f( _, _, _ )
glVertex3f( _, _, _ )
glEnd()
```

OpenGL would then compute a color at each vertex based on the surface normal and a lighting and material model using what I discussed above. Let's now turn to the question of how to "fill" a polygon, namely choose the colors of pixels in the interior.

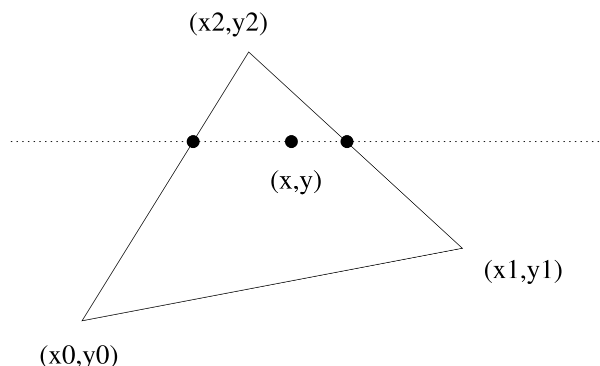
Flat shading

One approach which is perhaps the least interesting is to take one of the vertices of the polygon and choose the color of this particular vertex for all pixels that lie in the polygon. How OpenGL choose one particular vertex is specified here:
`https://www.opengl.org/sdk/docs/man2/xhtml/glShadeModel.xml`
This case is known as *flat* shading. (See `GL_FLAT`).

[ASIDE: In the lecture and slides I said that, for "flat shading", OpenGL computes the normal of the polygon (as defined by the vertices) and uses that normal to compute a single color for polygon. I realized afterwards that this wasn't quite correct, however. In OpenGL, "flat shading" is slightly more general than this. Flat shading means that OpenGL computes the color of one of the vertices of a polygon (depending on the surface normal of that vertex, however it is defined) and uses that vertex color for all of the points in the polygon.]

Linear interpolation (smooth shading)

A more common and interesting approach to choosing the colors in a polygon is to compute the intensities at all of the vertices v of the polygon and then, given these intensities (or more generally RGB values), fill the pixels inside the polygon[21] by *interpolating*.



A simple method of interpolation, which is possible for triangles, is *linear interpolation*. Suppose you have a triangle that projects into the image as shown above, and you have the intensities $I(x_i, y_i)$ for vertices $i = 0, 1, 2$. You wish to interpolate the intensities at the vertices to obtain the intensity $I(x, y)$ at an arbitrary interior point. Consider the edge between image vertices (x_0, y_0) and (x_2, y_2), where $x_0 < x_2$. We can interpolate the intensity along that edge as follows:

$$I(x', y) = I(x_0, y_0)\frac{x_2 - x'}{x_2 - x_0} + I(x_2, y_2)\frac{x' - x_0}{x_2 - x_0}.$$

Equivalently, this can be written:

$$I(x', y) = I(x_0, y_0) + (I(x_2, y_2) - I(x_0, y_0))\frac{x' - x_0}{x_2 - x_0}.$$

Similarly, one linearly interpolates the intensity along the edge between (x_1, y_1) and (x_2, y_2). Finally one linearly interpolates the intensity along the row that corresponds to the dotted line, yielding the desired intensity $I(x, y)$. Again, keep in mind that typically we will be interpolating R, G, and B values, not intensity values.

One can show (see Exercises) that the intensity values $I(x, y)$ that one obtains are a planar fit to the 3D points $(x_0, y_0, I(x_0, y_0))$, $(x_1, y_1, I(x_1, y_1))$, $(x_2, y_2, I(x_2, y_2))$. These are not physical 3D points, of course, they are just points defining a function in $(x, y) \times [0, 1]$, where intensity is in the interval $[0, 1]$.

Linear interpolation is a common method of shading in OpenGL. It is called smooth shading (`GL_SMOOTH`) and it is the default parameter of `glShadeModel`. See the URL on the previous page for the specification, if you are interested.

[21] that is, inside the image projection of the polygon

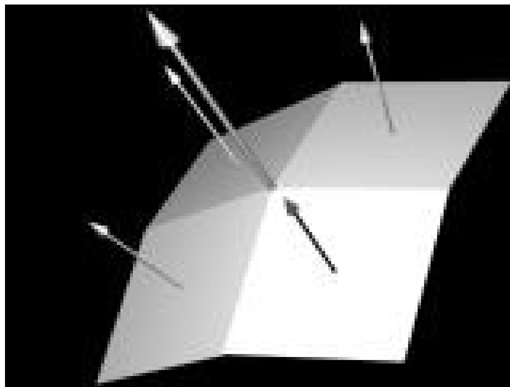
Gouraud shading

If we have a polygon *mesh* and the surface normal is constant across each polygon in the mesh, then we get sharp changes in intensity across the boundary between polygons when the surface normals of neighboring polygons are different. This is fine if the mesh is supposed to have discontinuities at the face boundaries, for example a cube. However, often meshes are used to approximate smooth surfaces. In that case, we don't want sharp intensity edges across face boundaries. Rather we want intensities to be smooth across face boundaries. How can we achieve this?

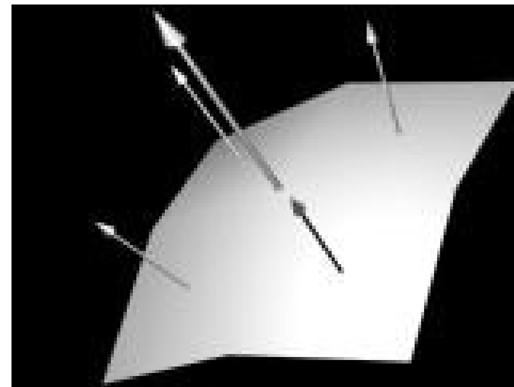
One idea is to define the normal at a vertex to be the average of plane normals $\mathbf{n}_i$ of all faces i that share this vertex,

$$\mathbf{n} \equiv \sum_i \mathbf{n}_i \;/\; \| \sum_i \mathbf{n}_i \| \;.$$

When we define normals in this way, faces that share a vertex will have the same normal at these vertices (since there is just one normal per vertex). When we interpolate the intensities along an edge that is shared by two faces, the intensities will be shared as well since the interpolation will be the same for the two faces along that edge. Thus, when we define normals in this way and we interpolate the intensities within the triangle faces from the intensities at the vertex points (and edges), we will get continuous intensities across edges. This method is called *Gouraud shading* (which is named after its inventor).



flat shading



Gouraud shading

Phong shading (interpolating normals)

The above example suggests that we should relax our concept of what we mean by the surface normal at each point. Taking this idea a step further, we can consider interpolating the surface normals across each polygon. This is called *Phong shading*. The term "Phong shading" is often used interchangeably to describe both Phong's lighting model (mentioned earlier) and the trick described here, where we interpolate the normals. Note, however, that that these are two separate ideas. Phong shading doesn't necessarily use the Phong lighting model. It just means that you interpolate the normals (and use the interpolated normal when coloring an interior point).

What does Phong shading buy you? Suppose you have a shiny smooth surface such an apple which you have represented using a coarse polygonal mesh, with vertices interpolated from the faces as described above. Suppose that the peak of the highlight occurs at a *non-vertex* point. If you use Gouraud shading, then you will miss this highlight peak, and the apple surface will not appear as shiny as it should. If, however, you interpolate the normal, then you have a good chance that some pixel in the interior will have an (interpolated) normal that produces a peak in the highlight.

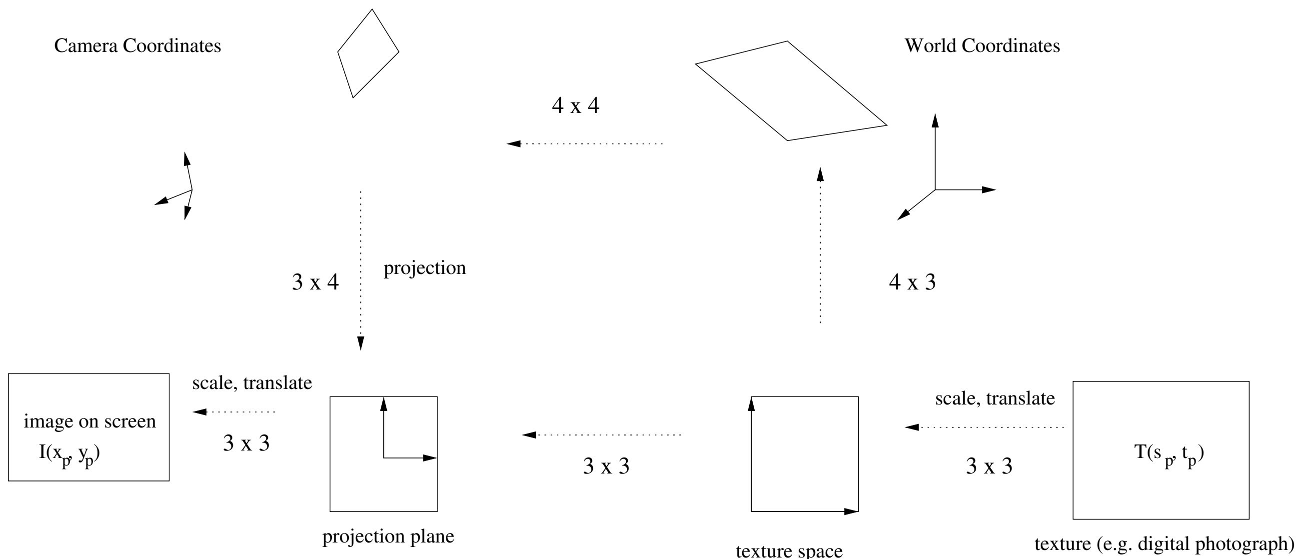
Texture Mapping

One often wishes to "paint" a surface with a certain pattern, or a *texture*. In general this is called *texture mapping*. The texture might be defined by a digital photograph $T(s_p, t_p)$ where (s_p, t_p) are pixels in the texture image. (These pixels are sometimes called "texels".) For example, suppose you want a ground surface to look like grass. You could model each blade of grass. Alternatively, you could just "paint" a grass pattern on the surface. The grass pattern could be just a photograph of grass. Similarly for a building: rather than making all the geometric details of the facade, you could justs use a single plane and paint the details. This might be sufficient if the building is in the background of the scene and the details are unlikely to be scrutinized.

To map a digital photograph onto a surface, one would need to compute the transformation that goes from the texture space (s_p, t_p) in which the digital photograph is defined, all the way to the pixel space (x_p, y_p) on the screen where the final image is displayed. The subscript p denotes that we are in discrete pixel space, rather than a continuum.

The mapping from the 2D texture space to the 2D pixel space can be illustrated with the sketch below. At the bottom right is a mapping from the texture image defined on (s_p, t_p) to the parameters (s, t) of the surface e.g. a square, a polygon. Then there is a mapping from these surface parameters (s, t) to world coordinates (which takes 2D to 3D) and a mapping from world coordinates to camera coordinates (3D to 3D) and the projection back to the 2D image domain (3D to 2D). Finally, there is the mapping from the image domain to the pixels (2D to 2D).

You've seen most of these mappings before. There an object to world and a world to camera ("modelview") and a projection. *In the lecture slides, I assumed that world coordinates were the same as camera coordinates, but in general that's not the case of course, and one needs to include that transformation in there.*



The projection is a bit subtle. It maps from camera coordinates to clip coordinates, which you recall are normalized device coordinates but prior to perspective division. So we have (wx, wy, wz, w). We are dropping the 3rd coordinate here, so it is just (wx, wy, w). The 3rd (z) coordinate is needed for hidden surface removal, but we are not dealing with that problem here.

The final mapping in the sequence is the window to viewport mapping, which we discussed in lecture 6.

You can see that the above sequence of linear mappings collapses to a 3×3 matrix, call it $\mathbf{H}$, and so we have the map:

$$\begin{bmatrix} wx_p \\ wy_p \\ w \end{bmatrix} = \mathbf{H} \begin{bmatrix} s_p \\ t_p \\ 1 \end{bmatrix}$$

The inverse of this map is:

$$\begin{bmatrix} ws_p \\ wt_p \\ w \end{bmatrix} = \mathbf{H}^{-1} \begin{bmatrix} x_p \\ y_p \\ 1 \end{bmatrix}.$$

To texture map the image $I(s_p, t_p)$ onto the polygon, we do the following:

```
for each pixel (x_p, y_p) in the image projection of the polygon {

      compute texel position using  (ws_p, wt_p, w) = H^{-1}(x_p, y_p, 1)
      I(x_p, y_p) = T(s_p, t_p)
}
```

Note that the computed texel position (s_p, t_p) generally will not be a pair of integers and so will not correspond to a unique texel. One therefore needs to be careful when choosing $T(s_p, t_p)$ for the color to write into the pixel. We will return to this issue later in the lecture.

Homography

The new concept here is the 3×3 linear transform $\mathbf{H}$ between texture coordinates $(s_p, t_p, 1)$ and the image plane coordinates $(x_p, y_p, 1)$. We examine this in more detail by looking at two components of the mapping, namely from $(s, t, 1)$ into world coordinates and then from world coordinates into the image projection plane.

The mapping from texture coordinates $(s, t, 1)$ into world coordinates is:

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} a_x & b_x & x_0 \\ a_y & b_y & y_0 \\ a_z & b_z & z_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s \\ t \\ 1 \end{bmatrix} \tag{9}$$

This mapping takes the origin $(s, t) = (0, 0)$ to (x_0, y_0, z_0). It takes the corner $(s, t) = (0, 1)$ to $(x_0 + b_x, y_0 + b_y, z_0 + b_z)$, and it takes the corner $(s, t) = (1, 0)$ to $(x_0 + a_x, y_0 + a_y, z_0 + a_z)$, etc. In homogeneous coordinates, it takes $(s, t, 1) = (0, 0, 1)$ to $(x_0, y_0, z_0, 1)$. It takes $(s, t, 1) = (1, 1, 1)$ to $(x_0 + a_x + b_x,\ y_0 + a_y + b_y,\ z_0 + a_z + b_z, 1)$, etc.

Let's assume, for simplicity, that world coordinates are the same as camera coordinates, i.e. there is an identity mapping between them. This is what I did in the slides. We then project points

from world/camera coordinates into the image plane $z = f$ by:

$$\begin{bmatrix} fx \\ fy \\ z \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

This gives

$$\begin{bmatrix} fx \\ fy \\ z \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} a_x & b_x & x_0 \\ a_y & b_y & y_0 \\ a_z & b_z & z_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s \\ t \\ 1 \end{bmatrix}$$

We get 3×3 matrix, namely the product of a 3×4 matrix, and a 4×3 matrix. This is the 3×3 linear transform $\mathbf{H}$ from $(s, t, 1)$ to (wx, wy, w).

In general, an invertible 3×3 matrix that operates on 2D points written in homogenous coordinates is called a *homography.* In the case of mapping from (s, t) to (x, y) space, we have in general

$$\begin{bmatrix} wx \\ wy \\ w \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} s \\ t \\ 1 \end{bmatrix}.$$

Example: a slanted ground plane

Let's consider a concrete example so you get a sense for how this works. Consider a plane:

$$z = z_0 - y \tan \theta$$

which we get by rotating the plane $z = 0$ by θ degrees about the x axis and then translating by $(0, 0, z_0)$. We parameterize the rotated 2D plane by (s, t) such that the origin $(s, t) = (0, 0)$ is mapped to the 3D point $(x, y, z) = (0, 0, z_0)$, the s axis is in the direction of the x axis. By inspection, the transformation from $(s, t, 1)$ to points on the rotated 3D plane, and then on to points on the image plane is:

$$\begin{bmatrix} wx \\ wy \\ w \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & 0 \\ 0 & -\sin\theta & z_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s \\ t \\ 1 \end{bmatrix} \qquad (*)$$

Multiplying the two matrices in (*) gives the homography:

$$\mathbf{H} = \begin{bmatrix} f & 0 & 0 \\ 0 & f\cos\theta & 0 \\ 0 & -\sin\theta & z_0 \end{bmatrix}.$$

The inverse happens to be:

$$\mathbf{H}^{-1} = \begin{bmatrix} \frac{1}{f} & 0 & 0 \\ 0 & \frac{1}{f\cos\theta} & 0 \\ 0 & \frac{\tan\theta}{z_0 f} & \frac{1}{z_0} \end{bmatrix}$$

The 4×3 matrix mapping texture space to 3D may be mysterious at first glance, but note that the first mapping is just a rotation about the x axis, with the third column removed. This mapping can be thought of as rotating a plane $(s, t, 0)$, where we don't bother including the $z = 0$ in the mapping since it has no effect. Another way to think of this mapping is to break it down into its components.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & 0 \\ 0 & -\sin\theta & z_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s \\ t \\ 1 \end{bmatrix} = s \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} + t \begin{bmatrix} 0 \\ \cos\theta \\ -\sin\theta \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ z_0 \\ 1 \end{bmatrix}$$

Although one can interpret the 4D vectors $(1, 0, 0, 0)$ and $(0, \cos\theta, -\sin\theta, 0)$ as points at infinity, this interpretation is not so helpful here. Rather think of $(1, 0, 0)$ and $(0, \cos\theta, -\sin\theta)$ as vectors that allow you to reach any point in the slanted plane, via linear combinations. (You need to use 4D vectors here because the slanted plane is shifted away from the origin and is not a 3D vector space i.e. it is not closed under addition.)

Aliasing

Recall from lecture 6 when I discussed how to scan convert a line:

```
m = (y1 - y0)/(x1 - x0)
y = y0
for x = round(x0) to round(x1)
   writepixel(x, Round(y), rgbValue)
   y = y + m
```

For each discrete `x` along the line, we had to choose a discrete value of `y` . To do so, we rounded off the `y` value. This gave us a set of pixels that approximated the original line.

If we represent a continuous object (line, curve, surface,..) with a finite discrete object then we have a many-to-one mapping and we lose information. We say that *aliasing* occur.s Here the word aliasing is used slightly differently from what you may be used to. In programming languages, aliasing refers to an object – or more generally to memory location – that has two different names.

```
x = new Dog()
y = x
```

The analogy in computer graphics (or signals in general) is that we have one discrete object (RGB values on a grid of pixels) that can arise from more than one continuous object (RGB values defined on a continuous parameter space).

To understand how aliasing arises in texture mapping, suppose I have continuous image $I(x, y)$ which is vertical stripes of width $w < 1$. The stripes have alternating intensities 0, 1, 0, 1, 0, 1 over continuous intervals of width w. If you now sample this image along a row (fixed y), where the pixel positions are $x = 0, 1, 2, 3, ...$ then the values that we get will typically not be alternating 0's and 1's. For example, run the python code:

```
#  sample from alternating 0 and 1 values on intervals of width w
w = 0.38
n = 15                  # number of samples
result = zeros(n)
for x in range(n):
   if (x / w) % 2 > 1:
       result[x] = 1
print result
```

The output for the given parameters is:

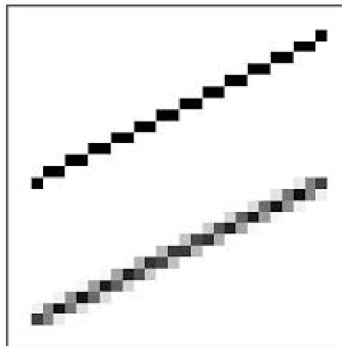
```
0  0  1  1  0  1  1  0  1  1  0  0  1  0  0
```

which is clearly not periodic. If you change w then you'll get a different output.

The main problem here is that the underlying image has structure that is at a finer scale than the sampling of the pixels. This problem often arises when you sample a continous signal. In the lecture slides I gave some other examples, including a Moiré pattern.

Anti-aliasing

In the case of scan converting a line, aliasing produces a jagged edge. Examples were given in the lecture slides. One idea to get rid of the jaggies (called "anti-aliasing") is to take advantage of the fact that RGB values are not binary but rather typically are 8 bit values (i.e. 0 to 255). For example, suppose that a line is supposed to be black (0,0,0) on a white (255,255,255) background. One trick to get rid of the jaggies is to make a pixel's intensity a shade of grey, if that pixel doesn't lie exactly on the line. The grey value of the pixel could depend on the closeness of the pixel to the continuous line. (There is no unique way to do this, and understand what's better or worse depends on factors take a while to explain – including issues in visual perception. We're not going down that rabbit hole.) See the example below. The first shows the jaggies; the second shows the "anti-aliased" line.



Aliasing and anti-aliasing in texture mapping

Aliasing arises in texture mapping as well. Recall the algorithm on page 2 of today's notes. For each pixel (x_p, y_p) in the image of a polygon, we map back to some pixel position (s_p, t_p) in the original texture. The simplest method would be to just round off (s_p, t_p) to the nearest integer and copy the (RGB) texture value $T(round(s_p), round(t_p))$ into $I(x_p, y_p)$.

A alternative approach is to consider a pixel as a small unit square, for example, centered around the point (x_p, y_p), namely the square $[x_p - \frac{1}{2}, x_p + \frac{1}{2}] \times [y_p - \frac{1}{2}, y_p + \frac{1}{2})]$. If we were to map the corners of this little square pixel back to the texture space (s_p, t_p) then we would get a four vertex polygon (quad) in that space. These four vertices typically would not correspond exactly to integer coordinates of the texture. GIven this lack of correspondence, how should one choose the color for the image pixel (x_p, y_p) ?

There are two general cases to consider. Here we consider texels and pixels to be unit squares in (s_p, t_p) and (x_p, y_p) space, respectively.

- *magnification*: This is the case that a texel in (s_p, t_p) maps forward to a region in (x_p, y_p) that is *larger* ("magnify") than a pixel in (x_p, y_p). Note that if we consider the mapping in the inverse direction, then a square image pixel centered at (x_p, y_p) would map back to a region that is *smaller* than a texel square in (s_p, t_p) space. Magnification can happen, for example, if the camera is close to the surface that is being texture mapped. (You may have noticed this when playing cheap video games, and you drive too close to a wall.)

- *minification*: This is the case that a texel in (s_p, t_p) maps forward to a region in (x_p, y_p) that is *smaller* ("minify") than a pixel in (x_p, y_p). In the inverse direction, a pixel in (x_p, y_p) maps back to a region that is *larger* than a texel in (s_p, t_p) space. In the case of a slanted ground plane, minification can happen pixels that are near the horizon.

Notice that smaller or larger is a bit ambiguous here since there is both an x and y direction. It could happen that there is a minification in one direction but a magnification in the other direction, for example, if you are close to a surface but the surface is highly slanted.

The method that one uses to choose the color $I(x_p, y_p)$ could depend on whether one has magnification or minification at that pixel. For example, if magnification occurs, then it can easily happen that the inverse map of an image pixel quad will not contain a texture pixel. To choose the color of the image pixel, one could ignore the inverse map of the image pixel quad and just consider the position of the inverse mapped pixel center. One could then choose the intensity based on the nearest texture pixel (s_p, t_p) or take a weighted average of the nearest four texture pixels, which form a square. This can be done with *bi-linear interpolation*. See the Exercises.

If one is minifying (shrinking) the texture, then the inverse map of the square image pixel might correspond to a larger region (quad) of the texture image. In this case, one might scan convert this region use the average of the RGB intensities.

As you can image, there is great flexibility in what one does here. There is no single approach that is best, since some approaches are more expensive than others, and the aliasing issues that arise depend on the texture being used and the geometry involved.

Texture Mapping in OpenGL

At the end of the lecture, I mentioned the basics of how texture mapping is done in OpenGL. First, you need to define a texture. For a 2D texture such as we have been discussing, you specify that it is 2D,[22] and various parameters such as its size (width and height) and the data i.e. RGB values.

```
glTexImage2D( GL_TEXTURE_2D, ...., size, .., data )
```

You also need to associate each vertex of your polygon with a texture coordinate in (s, t) space. You need to do this in order to define the mapping from texture coordinate space to your polygon, as discussed at the bottom of page 2 of these notes. This definition of (s, t) values for each vertex is occurs when you declare the polygon.

The texture coordinate, like a surface normal, is an OpenGL state. Each vertex is assigned the texture coordinate at that current state. If you want different vertices to have different texture coordinates – and you generally do want this when you do texture mapping – then you need to change the state from vertex to vertex. For example, here I define the texture coordinates to be the positions $(s, t) = (0, 1), (0, 1), (1, 0)$. You can define the texture coordinates to be any (s, t) values though.

```
glBegin(GL_POLYGON)
glTexCoord2f(0, 0)
glVertex( x0, y0, z0 )
glTexCoord2f(0, 1)
glVertex( x1, y1, z1 )
glTexCoord2f(1, 0)
glVertex( x2, y2, z2 )
glEnd()
```

I'll say more about texture coordinates next lecture.

[22]Not all textures are 2D. You can define 1D and 3D textures too.

In the first half of today's lecture, I elaborated on the ideas of texture mapping that were introduced last lecture. In the second half, I discussed the idea of procedural shading, and looked at a specific example known as bump mapping.

Texture mapping (continued from last lecture)

OpenGL pipeline

At the end of last lecture, I mentioned how texture coordinates (s,t) are assigned to vertices in OpenGL. We discussed the mapping between pixel coordinates (x,y) and texture coordinates (s,t) and we saw how this mapping was given by a composition of matrices, ultimately resulting in 3×3 invertible matrix known as a homography. The homography maps from texture coordinates to pixel coordinates. The inverse maps points on the interior of (the image projection of a) polygon to their corresponding texture coordinates.

I began today's lecture by recalling the OpenGL pipeline and distinguishing which aspects of texture mapping involve vertex processing and which involve fragment processing. The vertex processing stage involves little extra work (beyond the usual mapping from object coordinates to clipping, etc). The work begins rather at the rasterization stage, where the fragments are generated. That would be the time that the texture coordinates for each fragment are generated, using the inverse homography. Note that I am not distinguishing whether the inverse homography maps to (s,t) or to (s_p,t_p) space here. At the fragment processing stage, the lookup of the texture color would occur **verify how you write a fragment shader for texture mapping in GLSL. Is the GL_LINEAR interpolation done by the fragment shader or by the rasterizer ?**

OpenGL: Magnification, minification, and MIP mapping

In the slides from last lecture, I briefly mentioned the magnification and minification issue. Let's have a brief look at how this is done in OpenGL. See
`https://www.khronos.org/opengles/sdk/docs/man/xhtml/glTexParameter.xml` for details.

In OpenGL texture mapping, you can specify one of several methods for sampling the texture. If you have magnification, then there are two options. You can choose the integer coordinates (s_p,t_p) that are closest to the (non-integer) inverse mapped values. Alternatively, you can compute the RGB value by bilinearly interpolating the values of the four nearest neighbors. To choose between these two methods, you set a texture parameter to be either `GL_NEAREST` or `GL_LINEAR`.

The above method/parameters are available for minification as well. But there is another method as well which is more subtle. Recall that with minification, a square pixel $I(x_p,y_p)$ is inverse mapped to a region in (s_p,t_p) space that is larger than a pixel – sometimes much larger, as in the case at the horizon. The method I suggested last class where you average over that region in texture space can be expensive in this case, since for each pixel in the image you need to average over many pixels in the texture and this can be slow.

One method for speeding things up is called MIP mapping[23] which can be thought of as multiresolution texture mapping. This method pre-computes multiple versions of a texture of different sizes. For simplicity, suppose the original texture is $2^n \times 2^n$. Then a MIP mapped texture would

[23] It was proposed in 1983 by Lance Williams.

also have sizes $2^{n-1} \times 2^{n-1}, 2^{n-2} \times 2^{n-2}, etc. 2 \times 2$. Texture $2^{n-l} \times 2^{n-l}$ is said to be "level" l. These images are just lower resolution version of the original. Because each extra level takes 1/4 as much space as the previous level, this extra texture takes up relatively little texture memory overall.

There are many details involved in using MIPmapping in OpenGL so let me just sketch out the basic ideas. First, you need to specify that you want to use MIP mappings and you need to tell OpenGL to (pre)compute the MIP maps. (I will not go into *how* the different resolution textures are computed, since this requires some background in signal processing which most of you don't have.) Once MIP maps are computed, they can be used roughly as follows. During rasterization, when a fragment is generated, the image pixel (x_p, y_p) is inverse mapped to a texture coordinate (s,t) as before. But now at the fragment processing stage, it must be decided roughly how many pixels m are occupied by the inverse mapped 1×1 little square pixel centered at (x_p, y_p). (There is an OpenGL command for this query.) Similarly it must be decided which level of the MIP map is appropriate, that is, we want the $i-th$ level MIP map which is of size $m \approx 2^{n-i} * 2^{n-i}$. Typically the m satisfying will fall between two levels i and $i+1$, and in that case the RGB texture value must be linearly interpolated from the values of these two levels (albeit with more processing cost). For the values at each of the two MIP map levels, i and $i+1$, one also needs to specify whether to use `GL_NEAREST` or `GL_LINEAR`. You can read up a bit more about these parameter settings here: `http://www.glprogramming.com/red/chapter09.html`

Procedural textures

All the texture mapping discussion up to now has assumed that we are working with an existing texture image $T(s_p, t_p)$. However, many aspects of the texture mapping mechanism that we've described do not depend on this assumption. Many aspects of texture mapping would still be applicable if $T(s_p, t_p)$ were not a pre-existing image, but rather it were a function that could be computed at runtime.

There are a few advantages to defining a texture by a function (or method, or procedure). One advantage is that we could save on memory, since we wouldn't need to store the texture. A second advantage is that the parameters of the function might not need to be integers (s_p, t_p) but rather could be floats (s,t) and so the issues of aliasing and interpolation might vanish.

There are disadvantages too, though. Having to compute the texture values at runtime is computationally more expensive than just looking up the values in memory.

See the slides for some examples of texture mapped images that used procedural textures.

Procedural shading

Until now, we have thought of texture mapping as just copying (painting) an RGB value onto a surface point Let's next generalize our notion of texture mapping. In lecture 12, we introduced several shading models for determining the intensity of light reflected from a point. We saw that OpenGL uses the Blinn-Phong model. There the intensity was determined by parameters such as diffuse k_d and specular reflectance coefficients k_s, e which are defined at each vertex. The model also depends on the relative positions of the vertex position $\mathbf{x}$, the light source $\mathbf{x}_l$, and the camera.

One can combine texture mapping with a shading model. For example, material properties such as k_d, k_s, e could be stored as texture images $T_d(s_p, t_p)$, $T_s(s_p, t_p)$, $T_e(s_p, t_p)$. When choosing the RGB values for a vertex, one could compute the texture position (s_p, t_p) corresponding to each pixel

position (x_p, y_p) as before, but now one could look up the three values from the three texture images, and plug these values into the Phong lighting model. Moreover, one should not be restricted to the Blinn-Phong model when choosing RGB values. One can define whatever model one wishes. All one needs is access to the parameter values of the model, and some code for computing the RGB values given those parameters. This more general idea of computing (rather than just looking up) the RGB values is called *procedural shading*.

Gouraud versus Phong shading, and the OpenGL pipeline

Before we turn more examples of procedural shading, let's recall an important difference between smooth (Gouraud) shading and Phong shading. Suppose we have a triangle. With Gouraud shading, we choose an RGB value for each vertex of the polygon. We then use linear interpolation to fill the polygon with RGB values. Choosing the RGB values of the vertices is done with the vertex shader. These RGB values are then the color of the vertex. The fill occurs at the rasterization stage, when the polygon is "broken down" into fragments. In OpenGL 1.x, the rasterizer linearly interpolates the colors for each fragment of the polygon. This happens automatically when you set the shading mode to `GL_SMOOTH`. There is nothing left for the fragment processor to do.

With Phong shading (not be confused with the Phong lighting model), the RGB filling is done differently. Instead of interpolating the RGB values from the vertices, one interpolates the surface normals. This interpolation is done by the rasterizer, which assigns an interpolated surface normal to each fragment that it generates. The rasterizer does not assign RGB values, however. Rather, the RGB values are computed by the next stage in the pipeline – the fragment shader. (OpenGL 1.x does *not* do Phong shading.)

To do Phong shading or any of the more general examples of procedural shading such as we will see later in the lecture, one has to use OpenGL 2.x or beyond and one has to write the fragment shader. This is not particular difficult to do. One just specifies code for computing RGB values from all the parameters. For an example of what the code looks like, see:

`https://www.opengl.org/sdk/docs/tutorials/ClockworkCoders/lighting.php`

Coordinate systems for shading calculations

In which coordinate system are these shading computations performed? Consider a vertex and its associated normal. The vertex processor maps from object coordinates to clip coordinates. If it were also to map the surface normals (as direction vectors) to clip coordinates, these transformed normals could not be useful for the Blinn-Phong lighting model. As I mentioned at the end of lecture 5 (just prior to the Appendix), the mapped surface normal will not be normal to the mapped polygon. Moreover, the angles between the normal and light source, and other angles needed by the Blinn-Phong model would be wrong. We simply cannot perform Blinn-Phong calculations in clip coordinates.

To compute RGB values using the Blinn-Phong model, one typically uses eye/camera coordinates. The vertices are mapped to eye coordinates using only the `GL_MODELVIEW` matrix. This allows one to use real 3D distances in the model, for example, the distance from the vertex to the light source in the case of a spotlight. (Note that the vertices themselves still need to be mapped using both the `GL_MODELVIEW` and `GL_PROJECTION` matrices, so that clipping and rasterization can

be done. But shading computations – whether in the vertex shader or fragment shader – need a separate representation of the vertex positions and normals which is in camera coordinates.)

If a surface normal vector $\mathbf{n}$ were transformed by the `GL_MODELVIEW` matrix, there is no guarentee that its angle with respect to the underlying surface will remain the same. (See Exercises.) If we can't use the `GL_MODELVIEW` matrix to transform the normal to eye coordinates in general, then what can we use? We discussed a similar issue at the end of lecture 5 so let's revisit it.

Let $\mathbf{e}$ be a 4×1 direction vector whose direction lies in the plane of the polygon e.g. it could be the vector from one vertex of the polygon to another vertex of the polygon, $\mathbf{e} = \mathbf{x}_2 - \mathbf{x}_1$. Let $\mathbf{n}$ be a 4×1 vector in the direction of the surface normal vector. By assumption, we have $\mathbf{n}^T\mathbf{e} = 0$.

We are representing $\mathbf{n}$ and $\mathbf{e}$ as direction vectors, namely 4-vectors whose 4th component is a 0. Let $\mathbf{M}$ be any invertible 4×4 matrix, for example, the `GL_MODELVIEW` matrix. Using linear algebra matrix tricks (MATH 223),

$$0 = \mathbf{n}^T\mathbf{e} = \mathbf{n}^T\mathbf{M}^{-1}\mathbf{M}\ \mathbf{e} = ((\mathbf{n}^T\mathbf{M}^{-1})^T)^T\mathbf{M}\ \mathbf{e} = (\mathbf{M}^{-T}\mathbf{n})^T(\mathbf{M}\ \mathbf{e})$$

where $\mathbf{M}^{-T}$ means $(\mathbf{M}^{-1})^T$ or equivalently $(\mathbf{M}^T)^{-1}$ (by linear algebra). Thus, to transform $\mathbf{n}$ so that it is perpendicular to $\mathbf{Me}$, we need to multiply $\mathbf{n}$ by $\mathbf{M}^{-T}$.

In OpenGL 2.x and beyond, the matrices $\mathbf{M}_{\texttt{GL_MODELVIEW}}$ and $\mathbf{M}^{-T}_{\texttt{GL_MODELVIEW}}$ are predefined. Their names are `gl_ModelViewMatrix` and `gl_NormalMatrix`, respectively. See the above URL (or the slides) for an example of how this is used in a vertex shader.

Other examples of Procedural Shading

Toon/Cel shading, Gooch shading (1998)

Several methods have been proposed for making images that have simplified shading, as in cartoons. See the slides for examples. Also see `http://en.wikipedia.org/wiki/Cel_shading` One of the nice ideas here is to automatically draw outlines around objects. This can be done by setting pixels to black when the surface normal is perpendicular to the viewing direction.

Let's consider another example, but this time go into some of the technical details.

Bump mapping (Blinn 1974)

Phong shading smoothly interpolates the surface normal across the face of a polygon (or patch of bicubic). Bump mapping is a way of choosing the surface normals to give the surface a bumpy or wrinkled appearance. The idea is to add small bumps to a smooth surface, *without re-meshing the surface*, that is, without defining a fine mesh to capture the little bumps. For example, a surface such as an orange might be modelled as a sphere with many little bumps on it. We don't change the geometry of the sphere. Instead, when we compute the intensitites on the sphere we pretend as if the surface normal is varying in a more complicated way than it would on a sphere.

How can we define these variations in the normal? Suppose we have a smooth surface $\mathbf{p}(s,t)$ such as a triangle or bicubic. The surface normal is in direction $\mathbf{n}(s,t)$ which is parallel to $\frac{\partial \mathbf{p}(s,t)}{\partial s} \times \frac{\partial \mathbf{p}(s,t)}{\partial t}$. We define a bumpy version of this surface, by defining a *bump map* $b(s,t)$. We wish to imitate the surface normal variations that arise if one were to displace the surface by a distance $b(s,t)$ in the direction of the surface normal, that is, if we had a bumpy surface

$$\mathbf{p}_{bump}(s,t) = \mathbf{p}(s,t) + b(s,t)\ \mathbf{n}(s,t)$$

where $\mathbf{n}$(s,t) is the unit normal to the surface at $\mathbf{p}(s,t)$. This is the unit normal to the original surface, not the surface with bumps added on it.

Consider the partial derivative

$$\frac{\partial \mathbf{p}_{bump}(s,t)}{\partial s} = \frac{\partial \mathbf{p}(s,t)}{\partial s} + \frac{\partial b(s,t)}{\partial s}\mathbf{n}(s,t) + b(s,t)\frac{\partial \mathbf{n}(s,t)}{\partial s}$$

which is a vector tangent to the surface. In Blinn's paper (1978), we simplifies the model by assuming that the original surface $\mathbf{p}(s,t)$ is so smooth that

$$\frac{\partial}{\partial s}\mathbf{n} \approx 0.$$

For example, if the original surface were a plane then this partial derivative would be exactly 0! In addition, we assume that the bumps are small and so $b(s,t)$ is small. Putting these two assumptions together, we approximate the partial derivative by dropping the last term, giving:

$$\frac{\partial \mathbf{p}_{bump}(s,t)}{\partial s} \approx \frac{\partial \mathbf{p}(s,t)}{\partial s} + \frac{\partial b(s,t)}{\partial s}\mathbf{n}(s,t)$$

Similarly,

$$\frac{\partial \mathbf{p}_{bump}(s,t)}{\partial t} \approx \frac{\partial \mathbf{p}(s,t)}{\partial t} + \frac{\partial b(s,t)}{\partial t}\mathbf{n}(s,t)$$

Taking their cross product gives an approximation of the direction of the surface normal of the *bumpy surface*.

$$\begin{aligned}\frac{\partial \mathbf{p}_{bump}}{\partial s} \times \frac{\partial \mathbf{p}_{bump}}{\partial t} &\approx \frac{\partial \mathbf{p}}{\partial s} \times \frac{\partial \mathbf{p}}{\partial t} + (\frac{\partial \mathbf{p}}{\partial s} \times \frac{\partial b}{\partial t}\mathbf{n}) - (\frac{\partial \mathbf{p}}{\partial t} \times \frac{\partial b}{\partial s}\mathbf{n}) \\ &= \frac{\partial \mathbf{p}}{\partial s} \times \frac{\partial \mathbf{p}}{\partial t} + \frac{\partial b}{\partial t}(\frac{\partial \mathbf{p}}{\partial s} \times \mathbf{n}) - \frac{\partial b}{\partial s}(\frac{\partial \mathbf{p}}{\partial t} \times \mathbf{n})\end{aligned}$$

There is also a term that contains $\mathbf{n} \times \mathbf{n}$ but this term disappears because $\mathbf{n} \times \mathbf{n} = 0$.

The above vector is the normal that is used for shading the surface at $\mathbf{p}(s,t)$, instead of the original normal to the surface $\mathbf{n}(s,t)$ at that point. Of course, the above vector needs to be divided by its length, so we have a unit vector.

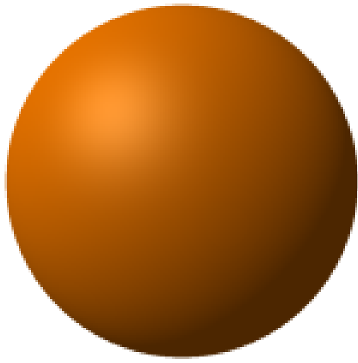
[ASIDE: Notice that there are three terms on the right side. The first term is parallel to the normal on the original smooth surface. The second two terms are parallel to the smooth surface's tangent plane. It is the second two components that define the perturbation of the normal on the surface, and these terms depend on the bump map $b(s,t)$.]

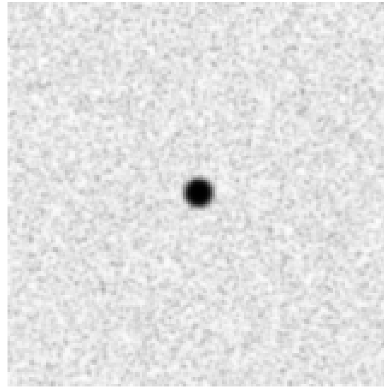
Given the above approximation, one can define the surface normal for the bumpy surface by

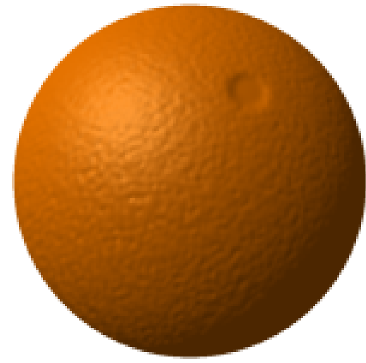
$$\mathbf{n}_{bump}(s,t) = (\frac{\partial \mathbf{p}_{bump}}{\partial s} \times \frac{\partial \mathbf{p}_{bump}}{\partial t}) \ / \ \mid \frac{\partial \mathbf{p}_{bump}}{\partial s} \times \frac{\partial \mathbf{p}_{bump}}{\partial t} \mid .$$

Assuming it is possible to find the texture coordinates (s,t) that corresponds to a given image pixel (x_p, y_p), one can compute a surface normal $\mathbf{n}_{bump}(s,t)$ using the above approximation. I will spare you the detail on this, since you really only need to go down that rabbit hole if you want to implement this.

An example (taken from the web) is shown below. I encourage you to search under "bump mapping" for other examples on the web.


smooth sphere


bump map


bumpy sphere

Normal mapping

The essential idea of bump mapping is that computes a fake normal at each point on the surface and uses that normal to compute the RGB image intensities. ()This work would be done by the fragment shader, not vertex shader).

Normal mapping is a similar idea to bump mapping. WIth normal mapping, though, we precompute the perturbed normals and store them as an RGB texture. That is, RGB $= (n_x, n_y, n_z)$. See Exercises.

In the slides, I gave an example of normal mapping. You can find that example along with more details here: `http://en.wikipedia.org/wiki/Normal_mapping`

Rendering mirror surfaces

The next texture mapping method assumes we have a mirror surface, or at least a reflectance function that contains a mirror component. Examples might be a car window or hood, a puddle on the ground, or highly polished metal or ceramic. The intensity of light reflected off each surface point towards the camera depends on what is visible from the surface in the "mirror direction." We write the formula for mirror reflection

$$\mathbf{r} = 2\mathbf{n}(\mathbf{n} \cdot \mathbf{v}) - \mathbf{v}$$

where $\mathbf{v}$ is the direction to the viewer and $\mathbf{r}$ is the direction of the incoming ray. This is essentially the same relationship that we saw in lecture 12 for two rays related by a mirror reflection, except there we had $\mathbf{l}$ instead of $\mathbf{r}$ for the direction of the incident ray. (We could have used $\mathbf{l}$ again here, but I chose not to because I want to emphasize that we are not considering a light source direction. Rather we are considering *anything* that can be seen in that mirror reflection direction.)

Note that since the mirror surface is a finite distance away, the $\mathbf{v}$ vector will typically vary along the surface regardless of whether the normal $\mathbf{n}$ varies. Since both the surface normal and $\mathbf{v}$ can vary along the surface, the reflection vector $\mathbf{r}$ vector typically varies along the surface as well.

Ray tracing

To correctly render the image seen "in the mirror," we would need to cast a ray into the scene from the surface point and in direction $\mathbf{r}$. We need to ask what is visible along that ray and what is the intensity of light along that ray. This is called *ray tracing*. If another mirror surface were visible along that ray, then we would need to repeat this process e.g. recursively. In Assignment 3, you will get some experience with the (very) basics of ray tracing.

When we discussed ray casting back in lectures 7 and 8, we saw several methods for speeding it up. We can apply these methods to ray tracing too, but you should appreciate that ray tracing is still computationally expensive. In particular, it cannot be done using the OpenGL pipeline, since it requires access to all the objects in the scene. In practice, vertex and fragment shaders don't have such access.

Environment Mapping (Reflection Mapping)

Let's next consider an alternative method for rendering mirror reflections, called *environment mapping*, which is much less expensive. An environment map is full "360 degree" RGB image of the a scene as seen from some particular position within the scene. More precisely, environment maps define RGB values over a sphere of directions, centered at some scene point. Defining functions on a sphere can be awkward since it is difficult to uniformly sample a sphere. We will discuss two methods for representing environment maps below.

Before we do, let's look at how environment maps are used. Define an environment map to be a function $E(\mathbf{r})$ of vector $\mathbf{r}$ which is the direction of a ray going out into the scene from the position where the environment map is defined. Here is the basic algorithm.

```
for each pixel (xp,yp){
    cast a ray to find the intersection point
    if ray intersects mirror surface at (x,y,z) ){
        compute normal  (nx,ny,nz)
        compute reflection direction (rx,ry,rz)
        I(xp,yp) := E(rx,ry,rz)
    }
}
```

This idea of this algorithm was proposed first by Blinn and Newell in 1976. (That paper also introduced the modified Blinn-Phong model.) See the slides for an example image which was shown in that paper. Note that, for this example image, the surface contains both a diffuse component (grey paint) and a mirror component.

Cube map

How does one represent the environment map $E(\mathbf{r})$? There are two common ways, and both are used in OpenGL. The first is a *cube map*. Here we represent the unit sphere of directions $\mathbf{r}$ by sampling the six faces of a cube. Each of the six faces of the cube defines a projection plane with a field of view of 90 degrees. To make a cube map using computer graphics we could render six images, each with a 90 degree field of view, and the six view directions being $\pm x, \pm y, \pm z$. The faces of the cube would be $x = \pm 1, y = \pm 1, z = \pm 1$.

The ray $\mathbf{r}$ intersects the cube at some position:

$$\frac{1}{max\{ \mid r_x \mid , \mid r_y \mid , \mid r_z \mid \}}(r_x, r_y, r_z)$$

and note that one of the coordinates has value either 1 or -1. Thus, the cube is $2 \times 2 \times 2$. The six images that define the environment could be written, say,

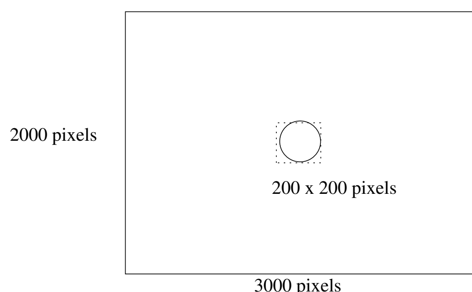
$$E_{z+}(r_x, r_y),\ E_{z-}(r_x, r_y),\ E_{y+}(r_x, r_z),\ E_{y-}(r_x, r_z),\ E_{x+}(r_y, r_z),\ E_{x-}(r_y, r_z).$$

To create the cube map of the environment, you render six images from the chosen scene point. (Here we assume that the scene does not have any mirror surfaces in it.) You can render these six images using OpenGL in the standard way using methods we have discussed up to now and which you are familiar with from the first two assignments (plus lighting and material).

Sphere map

A second method for obtaining environment maps is to use a real image of an environment, rather than rendering the environment. This is analagous to using real images for texture maps $T(s_p, t_p)$, rather than using computer generated textures.

With a *sphere map* (also known as a *disk map*), the directions of the unit sphere are represented by a disk. The environment map is defined by the image in this disk. The disk map typically uses a digital photographic image of a real scene that is reflected off a real mirror sphere, located in the scene. In the figure below, the dotted square contains a cropped image of a mirrored sphere in a



real scene. See the example of the floating dog that I discussed in the lecture, and the example of the Terminator 2 movie that I mention later.

Recall how the environment map is used. For each reflection direction, we would like to look up a value in the environment map. But now the environment map is parameterized by points on an image of a sphere which is a texture. So we need a correspondence between reflection directions $\mathbf{r}$ and "texels" (s_p, t_p) on the image of the sphere. In particular, we want a *mapping* from the vector $\mathbf{r}$ to texture coordinates (s, t) or (s_p, t_p). How to obtain this correspondence?

Assume the projection of the mirror sphere into the image is approximately orthographic. Then the direction of the viewer is approximately $\mathbf{v} = (0, 0, 1)$ for all points on the small mirror sphere. A (unit) sphere has the nice property that the unit surface normal is identical to the position of a point on the sphere. Let's normalize the image of the sphere so that the square containing it is $[-1, 1] \times [-1, 1]$ and let (s, t) be the parameters of this 2×2 'normalized' square. Then, for any point $(s, t, \sqrt{1 - (s^2 + t^2)})$ on the surface of the unit sphere mirror, the unit surface normal is

$$\mathbf{n} = (s, t, \sqrt{1 - (s^2 + t^2)}).$$

As we saw in lecture 12 (see also slides of this lecture near the beginning), a reflection vector $\mathbf{r}(s, t)$ is computed using:

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

So

$$(r_x, r_y, r_z + 1) = 2\sqrt{1 - (s^2 + t^2)}\ (s, t, \sqrt{1 - (s^2 + t^2)}) \qquad (*)$$

Taking the magnitude of both sides (and remembering that $\mathbf{n}$ is a unit vector) we get:

$$\sqrt{r_x^2 + r_y^2 + (r_z + 1)^2} = 2\sqrt{(1 - (s^2 + t^2))}$$

Substituting into the right hand side of Eq. (*) and rearranging gives:

$$(s, t) = \frac{1}{\sqrt{r_x^2 + r_y^2 + (r_z + 1)^2}}\ (r_x, r_y)$$

This is the mapping we seek. For any $\mathbf{r}$, we can calculate the texture coordinates (s, t) that correspond to $\mathbf{r}$.

A few points of clarification are perhaps needed here. First, in OpenGL, one uses texture coordinates $(s, t) \in [0, 1] \times [0, 1]$. To obtain such coordinates, one needs to map a bit more, namely

scale by $\frac{1}{2}$ and add $\frac{1}{2}$. This gives the formulas for (s,t) as a function of r_x, r_y, r_z that are found in the OpenGL specification and in some OpenGL/graphics textbooks.

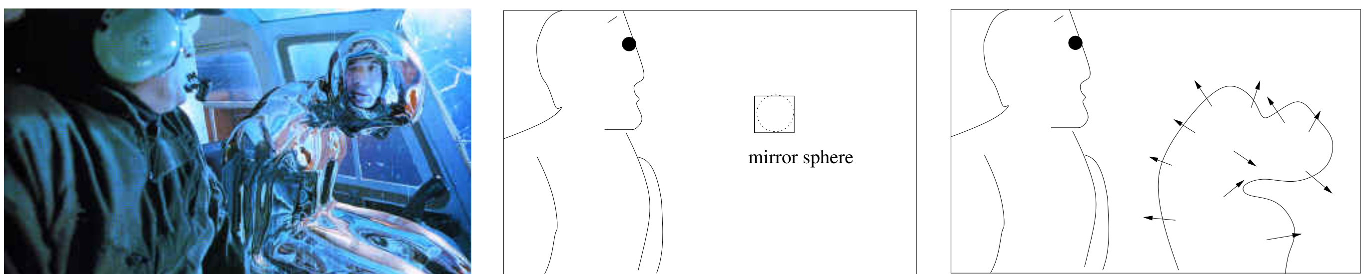
Second, you might get the impression from the above derivation of the mapping that, since we are assuming a constant $\mathbf{v} = (0,0,1)$ direcction, the reflection vector $\mathbf{r}$ depends only on the surface normal $\mathbf{n}$. This impression is not correct. The mirror reflection vector $\mathbf{r}$ depends both on the surface normal $\mathbf{n}$ and on the viewing direction $\mathbf{v}$ in the particular scene. The viewing direction $\mathbf{v}$ on the mirror surface being rendered typically will not be constant on the surface.

Example 1 (special effects in cinema: Terminator II)

Earlier I mentioned that the disk/sphere method uses a real mirror sphere and a real image of it. This gives an environment map texture image $E(s_p, t_p)$. In class, I discussed an example of how this was used in several scenes in the film Terminator II. (See see `/www.debevec.org/ReflectionMapping/`.) In the scene shown below, a computer graphics generated creature is inserted into the scene. The surface of the creature is a smooth mirror which seems to reflect the surrounding scene. The way this was done is illustrated in the figure below. The original scene was filmed with an actor and a real environment, and a mirror sphere was placed in the scene. In each frame of the film, the image in the mirror sphere served as the environment map for that frame.

Using these environment maps, a separate animation was created. The animation only consisted of the creature changing shape and moving over time. This was created entirely using computer graphics, of course, namely the environment maps were used to render a mirror reflection of the scene on the surface of the creature. Because the creature's surface is curved, the mirror reflection is distorted, like the reflection in a fun-house mirror. Finally, the animation of the moving creature was "inserted" into the film (covering up the mirror sphere).

Notice that the reflection of the actor's full face can be seen in the head of the creature, even though only part of the actor's face is visible in the original image (since his head is turned). This is exactly what should happen if there were a creature in the scene.



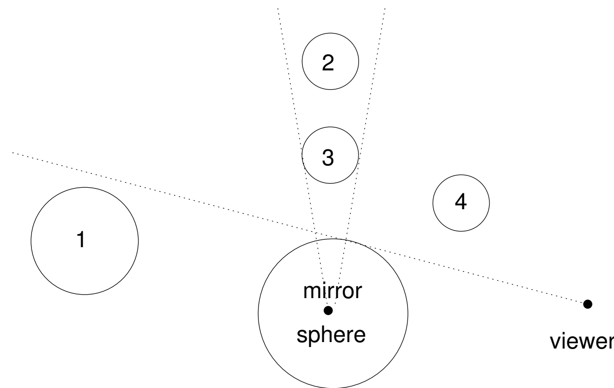
It is important to realize that environment mapping is only an approximation. It assumes that the intensity of a ray arriving at the mirror surface depends only on the direction of that ray and not on the position on the surface where the reflection occurs. This assumption is reasonable if that the environment that is being reflected is far away from the object and that there is nothing intervening between the object and its environment. However, if this condition is not met, then the rendering is not "correct". But generally it doesnt matter – people looking at it cannot tell.

Example 2

Consider the scene shown below which consists of a mirror sphere, a set of matte spheres 1 to 4, a viewer position (small black disk), and a position from which the environment map is originally computed. (If the discussion below is not enough for you, see the images used in the slides which added further 'lines of sight'.)

If the scene were rendered using ray tracing, then which of the matte objects would be visible, either directly or in the mirror? Sphere 1 would not be visible, neither directly nor in the mirror. Sphere 2 would be partly visible [24] in the mirror, but not directly (occluded by 4). Sphere 3 would be partly visible directly (i.e. only partly occluded by 4) and would be visible in the mirror. Sphere 4 would be visible directly and in the mirror.

If environment mapping were used, then the visibilities would be different. Sphere 1 would be visible in the mirror (but still not visible directly). Sphere 2 would not be visible either in the mirror (since it would not be seen in the environment map) or directly (since it would be occluded by 4). Sphere 3 would be partly visible directly, and would be visible in the mirror. Sphere 4 would be as before.



Environment mapping and the OpenGL pipeline

Ray tracing cannot be done using OpenGL graphics pipeline that we have been discussing. The reason is that, to know the RGB value of a reflected ray (off a mirror) from a point $(x, y, z(x, y))$, you need to find closest object in the scene along that reflected ray. However, finding the closest object requires access to a list of all the objects in the scene. The vertex and fragment processors do not have such access. They process the vertices and fragments individually using information about normals, surface attributes, texture coordinates. Similarly, the clipper processes the primitives and polygons individually.

[24] To see why, consider the ray that goes from the rightmost point on sphere 2 and passes near the rightmost point on sphere 3. This ray would reflect back and pass between sphere 3 and 4. Now consider a second ray which goes from the rightmost point of sphere 2 and touches close to the rightmost point of the mirror sphere. This ray would reflect to a ray that goes below the viewer. Somewhere between the above two rays from sphere 2 there must be a ray from sphere 2 that would reflect to the viewer. From this reasoning, you should be able to see that there are points on sphere 2 that are seen by the viewer.

The vertex and fragment shaders do have access to texture maps, of course, and this is what allows them to do environment mapping. So at what stages of the pipeline is environment mapping carried out?

There are two answers that should come to mind, which are similar to what we have been discussing with other methods in the last few lectures. The first answer is analogous to smooth shading. Here, the vertex processor would compute the RGB values for the vertices of a mirror polygonal surface – by computing the reflection vector and looking up the RGB value in an environment map. The rasterizer would then interpolate the reflection vector $\mathbf{r}$ across the fragments of the polygon. The fragment processor would then use the interpolated reflection vector for each fragment to lookup the RGB value in the environment map.

This first answer is *not* what OpenGL does. Why not? The problem is essentially the same here as it was for smooth shading with specular highlights. In the case of a mirror polygon, you want to know what is visible at each point in the interior of the mirror but this depends on the reflection vector at each point. If you just interpolate the RGB values from the vertices, then the RGB values of all interior points will just be a linear combination of the vertex RGB values – and this would just give a smooth blurry mush, not a mirror reflection.

The second method (which is what OpenGL does use "under the hood") is that the vertex processor computes the reflection vectors of the vertices, and passes these to the rasterizer which now interpolates the reflection vectors across the fragments. The fragment processor is then responsible for computing the RGB values, based on the reflection vectors.

Refraction

At the end of the lecture, I briefly discussed the problem of refraction whereby light passes through a transparent medium such as water or glass. In this case, the light direction changes at the interface between air and glass/water for reasons you learned about in your U0 physics course. The result is that the images of objects under water or behind glass are distorted. Ray tracing and environment mapping methods can be applied to render images in these situations. The basic idea is to trace rays of light (backwards) from they eye to the surface of these glass/water objects, through the objects, and then possibly out of the objects into the world. In the latter case, environment mapping can be used since one can just lookup the RGB value of the ray from the environment map $E(\mathbf{r})$.

To do this "properly", one considers the basic physics of refractions, e.g. Snell's law which you learned in U0 physics. In the lecture, I gave a few example images and videos which you should check out.

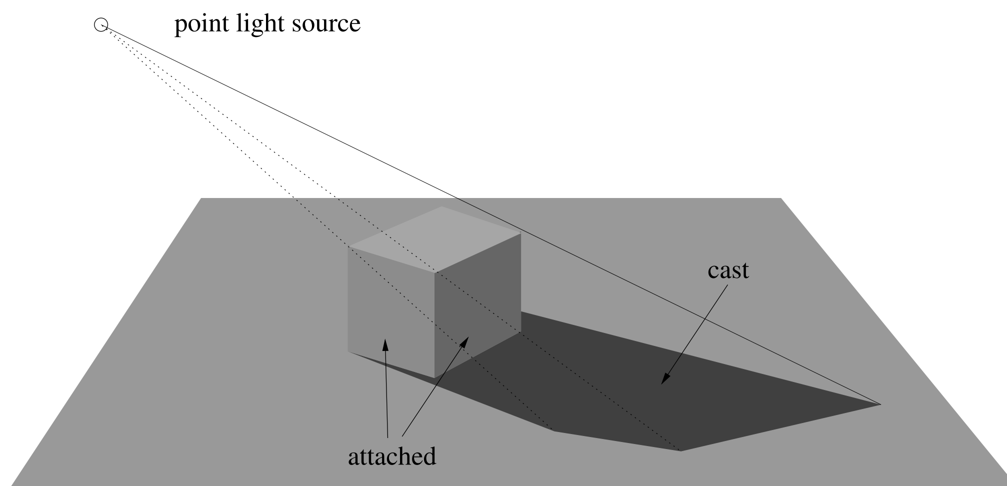
- show maps are depth textures, fast computation
- shadow mapping can be done with ffp or programmable ertex and fragment shaders
- Final pass of shadow map. See handwritten notes.

Shadows

Suppose we have a point light source. We say that a 3D point $\mathbf{x}$ is in shadow if the light source is not visible from the point. There are two reasons shadows occur. One is that the surface normal may be pointing away from the light source. We discussed this case in our lighting model when we introduced the term $\max(\mathbf{n} \cdot \mathbf{l}, 0)$. We say this is an *attached shadow* since we can think of the surface as blocking the light from itself and so (vaguely speaking) the shadow is attached to the surface. The other case is that the $\mathbf{n} \cdot \mathbf{l} > 0$ and there is another surface that is blocking the light source. We call this a *cast shadow*. Today we deal mostly with cast shadows.

In the sketch below, the sides of the cube lie in attached shadow and the dark shaded regions on the ground lie in a cast shadow. Dotted lines indicate some of the rays that that bound the shadow. [ASIDE: Notice that because the cube is a set of polygons, the shadow that is cast by the cube is also a polygon. Some methods for computing shadows take advantage of this. For a more general smooth surface that casts a shadow, however, the geometry of the cast shadow may be more complicated.]

Our illumination model can be updated to account for shadows as follows. Suppose we have just one light source. Consider a function S defined on surface points $\mathbf{x}$ in the scene such that $S(\mathbf{x}) = 1$ if the surface point $\mathbf{x}$ receives light from the source and $S(\mathbf{x}) = 0$ if this surface point lies in the shadow of the source. Here we let S account for both cast and attached shadows.



Recalling our model of the diffuse, glossy, and ambient lighting terms,

$$\begin{aligned} I_{diffuse}(\mathbf{x}) &= I_l(\mathbf{x})\ k_d(\mathbf{x})\ max(\mathbf{n}(\mathbf{x}) \cdot \mathbf{l}(\mathbf{x}), 0) \\ I_{glossy}(\mathbf{x}) &= I_s(\mathbf{x}) k_s(\mathbf{x}) (\mathbf{r}(\mathbf{x}) \cdot \mathbf{v}(\mathbf{x}))^{e(\mathbf{x})} \\ I_{ambient}(\mathbf{x}) &= I_a(\mathbf{x}) k_a(\mathbf{x}) \end{aligned}$$

we could now write the sum as:

$$I(\mathbf{x}) \ = \ S(\mathbf{x})\{I_l(\mathbf{x}) \ k_d(\mathbf{x}) \ \mathbf{n}(\mathbf{x}) \cdot \mathbf{l}(\mathbf{x}) + I_{glossy}\} + I_{amb}k_a(\mathbf{x})$$

Notice that we no longer need a term "$max(\mathbf{n} \cdot \mathbf{L}, 0)$" because the possibility that $\mathbf{n} \cdot \mathbf{l} < 0$ is now handled by the shadow function S. Also notice that we do not apply the $S(\mathbf{x})$ to the ambient term. The reason is that we want to fill in the shadow region with some light so that it is not black.

Ray tracing techniques could be used to decide if a point is in shadow. From each image pixel, we could cast a ray through that pixel to a surface point, and then trace a ray from that surface point back to the light source and check if the traced ray to the source intersects an object. (If it does, then the point is in shadow.) While this ray tracing method works, we would like to come up with a more clever way to compute shadows (just as environment mapping simplified the ray tracing for mirrored surfaces).

Shadow map

We wrote above that a point is in shadow if the light source is not visible from the point. An equivalent way to express this is to say that a point is in shadow if it is not visible from the light source. So, let's consider what is visible from the light source.

Suppose we have a point light source. Consider a cube centered *at the position of the point source*. For each face of the cube, we define a square grid of pixels and we compute the depth of points in the scene using the depth buffer method. Call this depth map a "*shadow map.*" We have one shadow map for each of the six faces of the cube.

Now suppose we wish to render the scene. For any pixel (x_p, y_p) in the *camera's image*, we cast a ray into the scene as before, computing the visible point $\mathbf{x}$ on the surface. We then transform 3D this point from camera coordinates into light source coordinates. We need to consider six different light source coordinate systems, one for each face of the cube. For example, we could let the light source coordinate system have axes parallel to the axes of the camera coordinates, so that the six faces of the shadow map would have depth directions $\pm x, \pm y, \pm z$, respectively. This cube map is exactly analagous to what we described in the environment map from last class, except that now we are computing it from the position of a point source rather than from the origin of some mirror object.

To decide if a scene point $\mathbf{x}$ is visible to the light source, we first need to figure out which face of the light cube the point $\mathbf{x}$ projects to. This is easily done by transforming the scene point to each of the six coordinate systems, projecting to the plane defined by each, and then clipping to the face. Each scene point would project to exactly one cube face of the shadow map, i.e. it would be clipped in five of the six faces.

Once we've figured out which face a point $\mathbf{x}$ projects to, we compare the depth z of the point in that light source coordinate system to the depth that is stored in the shadow map at the point of projection. The scene point is in shadow if its depth is greater than the depth in the shadow map.

Two issues arise in practice which are due to the discrete nature of images. First, as we saw in texture mapping, the scene point will not in general project exactly to a pixel position in the shadow map. Rather, it will typically project to a point in a square whose four corners are pixel positions in the shadow map. The shadow map specifies the depth of the four corners of this square, but it does not specify the depth at points interior to the square. So how do we know what is the depth at the point where $\mathbf{x}$ projects to?

A first guess at obtaining the depth of a point interior to the square is to interpolate (as we did with texture mapping). Note, however, that interpolation is only appropriate here if the depth map is smooth in the interior of the square. This is not always the case e.g. if the boundary edge between two objects falls in the interior of this square. In this case, one needs to be more sophisticated in order to say correctly whether $\mathbf{x}$ is in shadow.

A second issue is that the depths in the shadow map are represented with finite precision. If you use only 8 bits per pixel in the shadow map then points that are in fact at different depths could be represented by the same (quantized) 8 bit depth value. This can lead to mistakes: you could conclude a point is in shadow when it in fact is not, or you could conclude a point is not in shadow when in fact it is. (This issue arises in general for z buffer methods, and in general you should use as many bits as you can afford when you use z buffers.)

Multiple light sources and environmental lighting

Most scenes have more than one light source. How can we model multiple light sources? This is easy. We just add the images produced by each of the sources.

One interesting and common example is that the light source is the entire sky. We could just treat the entire sky as a sampling of a hemisphere and add up the images from each sampled direction of the hemisphere. (Note that the points sources would be at infinity i.e. parallel sources, which is slightly different from the case we discussed above where the point source was in the scene.)

If we are modelling the lighting from the sky, then we can use an environment map $E(\mathbf{l})$ to represent the intensities of light from different directions, and we can write the image intensity as:

$$I(\mathbf{x}) \;=\; \sum_{\mathbf{l}\,\in\, hemisphere} S(\mathbf{l},\mathbf{x})\; E(\mathbf{l})\; \{k_d(\mathbf{x})\mathbf{n}(\mathbf{x})\cdot\mathbf{l} \;+ k_s(\mathbf{x})(\mathbf{r}(\mathbf{x})\cdot\mathbf{v}(\mathbf{x}))^{e(\mathbf{x})}\} + k_a(\mathbf{x})I_a$$

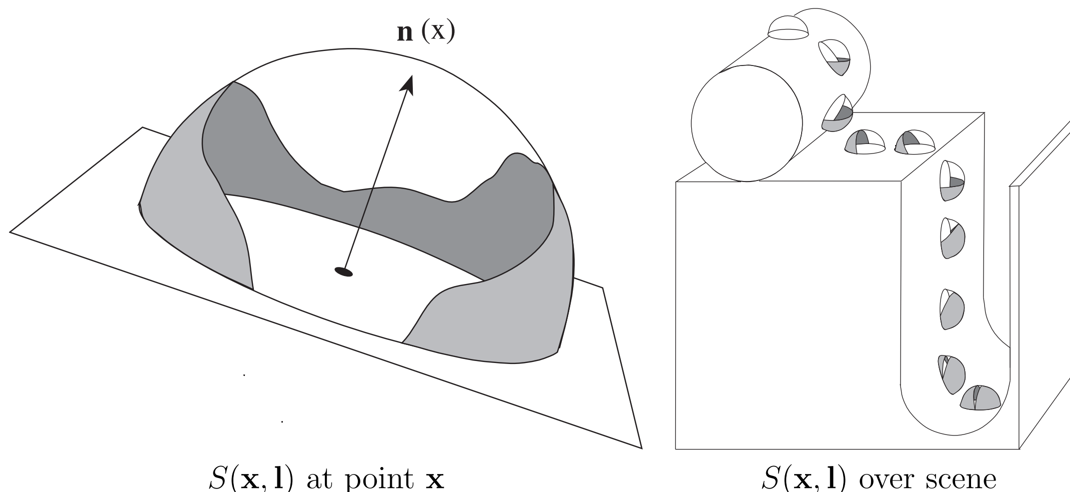
where we are now sampling[25] the directions $\mathbf{l}$ in a hemisphere centered in the normal direction. Notice that the shadow function (not to be confused with the "shadow map" defined on the previous page) is now written $S(\mathbf{x},\mathbf{l})$ to make more explicit that we are concerned with whether the environmental illumination is indeed visible from $\mathbf{x}$ in each direction $\mathbf{l}$. If the environment is not visible (because some surface is blocking it) then we do not want to include that blocked direction in the summation. We are still treating $S(\mathbf{x},\mathbf{l})$ as having binary values.

We illustrate the function $S(\mathbf{x},\mathbf{l})$ for a fixed $\mathbf{x}$ in the figure above (left). Only a hemisphere is shown, namely the hemisphere of directions $\mathbf{l}$ that are above the surface, from which the surface can receive illumination. We have eliminated the hemisphere below the surface since such directions cannot contribute to the shading – such directions produce attached shadows.

Two sets of directions within the hemisphere are shown. The darkened set illustrates those directions in which the environmental illumination is not visible ($S = 0$), and the clear set indicates the directions in which it is visible ($S = 1$).

The sketch on the right illustrates how the shadow function changes as we move along the surface. The scene shows a cylinder on the ground and a gully dug out of the ground. Notice that as $\mathbf{x}$ moves around the cylinder and along the ground and down into the gully, the amount of environment that is visible varies systematically.

[25] How exactly we sample the sphere depends on how the environment map is stored e.g. recall the cube map and disk map representation from environment mapping.



$S(\mathbf{x}, \mathbf{l})$ at point $\mathbf{x}$ $S(\mathbf{x}, \mathbf{l})$ over scene

Notice that using environmental lighting in combination with a Phong lighting model is different from environment mapping a mirror surface, where we "copied" (texture mapped) the intensity from incoming ray to outgoing ray. In the above equation, we are not assuming a perfect mirror. We don't just copy one intensity; rather we use all the intensities of the environment as parameters in a shading procedure.

Also, we see that there are essentially two computational problems that must be solved. One is that we need to compute $S(\mathbf{x}, \mathbf{l})$, namely we need to compute whether the point at $\mathbf{x}$ sees the sky/environment in direction $\mathbf{l}$. Then, we need to calculate the summation given all the parameters i.e. add up the intensity contributions from those directions in which the sky is visible i.e. $S(\mathbf{x}, \mathbf{l}) = 1$.

Although it is relatively expensive to compute the function $S(\mathbf{x}, \mathbf{l})$ for each vertex $\mathbf{x}$ in the scene, once we have computed this function we can use it again and again. Imagine you are trying to make an image of complicated scene, and you are not sure exactly what lighting and materials you want for this scene. You might try rendering the scene many times, changing the lighting and material parameters in each case. Obviously it would be wasteful to also recompute $S(\mathbf{x}, \mathbf{l})$ each time, since this function doesn't depend on the lighting E and material k, e parameters.

We can take this idea one step further. Consider one object in the scene, such as an airplane which might be flying through the scene. This object *self-occludes* in that points on the object can cast shadows on other points on the object. For example, the wings would occlude the sky from parts of the fusillage. If we just consider the airplane, then we can compute the function $S(\mathbf{x}, \mathbf{l})$ for points on the airplane, where the positions $\mathbf{x}$ and directions $\mathbf{l}$ are in the airplane object's local coordinate frame. As the airplane then moves through space and rotates, we can map $\mathbf{x}$ and $\mathbf{l}$ to the world coordinate frame, and avoid having to recompute this function.

This method of computing $S(\mathbf{x}, \mathbf{l})$ in the local object's coordinate frame (and considering only shadows that are cast by the object on itself), then transforming this function to world coordinates and rendering the object using an environment map is one of a family of shadow/environment mapping tricks know collectively as *ambient occlusion*.[26] The term "ambient" refers to the light

[26] "Ambient occlusion" is a relatively new concept in computer graphics, and the term is used differently by different people. Some just use it to refer to the case of uniform environment lighting i.e. $E(\mathbf{x}, \mathbf{l})$ is a constant, whereas other

that comes from all directions – as in the environment. The term "occlusion" refers to the fact that some directions of the environmental illumination are occluded.

Computing the ambient occlusion field during rendering allows one to change the lighting at the production step, without having to recompute all the geometric stuff.

Change the material - turn something into a shiny surface and use environment map, or light with different shadows (different positions)

---

consider the more general situation that I discussed above .

Image Compositing

The methods we have been developing assume that we have a complete scene model and that we render this entire model from some chosen viewpoint. In practice, however, when making images, one often works with only parts of the scene at a time. For example, one might have in mind a background scene and a foreground scene. This should be a familiar concept. In live theatre and often in film production, one sets up a stage with various objects and backgrounds. You do this in advance of the actors appearing. Similarly, in computer graphics animation, it is common to render a background scene before one renders the movement of the characters. (We discussed an example in the lecture on environment mapping: the mirror creature in Terminator 2 was rendered offline and then inserted into the film with real actors).

Separately rendering background and foreground scenes has certain practical advantages in the production stage, when you are developing and fine-tuning a scene model. For example, you can avoid having to re-render the entire background scene everytime you want change the foreground, or vice-versa. But it also raises technical challenges. How can you keep track of which points belong to the background and which belong to the foreground? How can you combine the background and the foreground layers?

The simplest approach is to have a separate binary image for the foreground – sometimes called a *mask* – which indicates which pixels are occupied by the foreground and which are not. Then, when writing the foreground over the background, you only write over the pixels that have value 1 in the mask.

One example of this idea is *blue screen matting* which has been used in cinema and television for many years. A person is shown in front of a complex background, and in fact the person is being filmed in a studio in front of a simple background, and the video of the person is superimposed over a background video/image. An example is a television weather report. The weatherman might be filmed standing in front of a blue background screen, but what is shown on the broadcast is the weatherman standing in front of a large meterological map.

The way this is achieved is as follows: The person is filmed in front of a blue screen, and the final video is created by making a decision, for each pixel, whether to use the filmed image/video (person in front screen) or the background image (weathermap). This decision is made based on whether the RGB value at that pixel is sufficiently blue - i.e. the R and G values are much smaller than the B value. If yes, then the pixel is assumed to belong to the background and the RGB value of the pixel is replaced by that of the new background.

You may have noticed with amusement that if the person being filmed has blue eyes or is wearing a blue necktie or has blue mascara, then the new background video is mistakenly substituted at these points. You may also have noted that the edges of the person (especially around the hair) are often less than satisfactory. The person's outline sometimes has a cookie cutter look.

RGBA (rgbα)

A more challenging version of the problem arises when a pixel is *partially* occupied by a foreground image, for example, a pixel on the boundary of a surface in the foreground image. Think of the pixel as a small square (rather than a point) and suppose that the projection of the surface in the foreground image covers only part of the square. For example, recall lecture 1 when we discussed how to draw a line and we had to deal with the issue of how to approximate a continuous line using

a square grid. The idea was to spread the values of a line between pixels in the (common) case that the line passed between pixels. A similar problem arises at the edge of a foreground image: we want to avoid jagged edges and to choose intensities correctly for the pixel, then we need to combine intensities from the foreground and background. That is, we need to *mix* the colors of the background and foreground which both contribute to the color of the pixel. If we don't do this, then the foreground image will have a jagged boundary, and this will not look good.

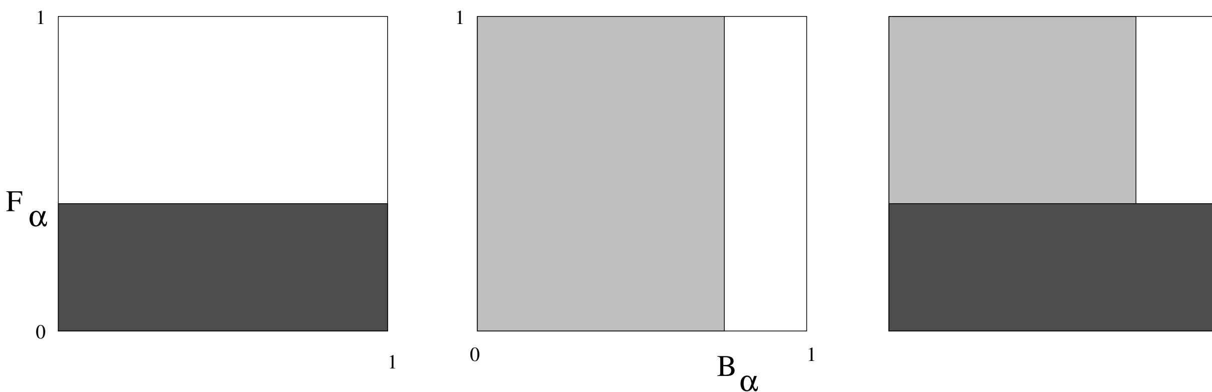
A common method for keeping track of how much of a pixel is covered by an image is to add a fourth component to each pixel, called "alpha" (α). We can interpret α as the fraction (between 0 and 1) of the pixel that is covered by this image. It is common to define RGBA to have values in [0,1]. For example, if we are using 8 bits for each value, then we consider the values to be in $\{0, \frac{1}{255}, \frac{2}{255}, \ldots, \frac{254}{255}, 1\}$. Often, though, we will just say the values are in $\{0, 1, 2, \ldots, 255\}$ to keep the notation simpler. We interpret the α values as follows: 0 is not occupied at all, 1 is completely occupied, and in between we have partial occupancy.

Suppose we have images $F_{rgb\alpha}$ and $B_{rgb\alpha}$, where F is a foreground image that we wish to draw over the background image B. For generality, we allow the alpha "channels" F_α and B_α to be anywhere from 0 to 1, in particular, for the moment we do not require that each pixel in the B image is entirely occupied.

Given $F_{rgb\alpha}$ and $B_{rgb\alpha}$, how do we write the foreground image over the background image? In the computer graphics problem of *image compositing* [27], this is called writing F over B. There are two steps. One is to define $(F \text{ over } B)_\alpha$, and the other is to define $(F \text{ over } B)_{rgb}$.

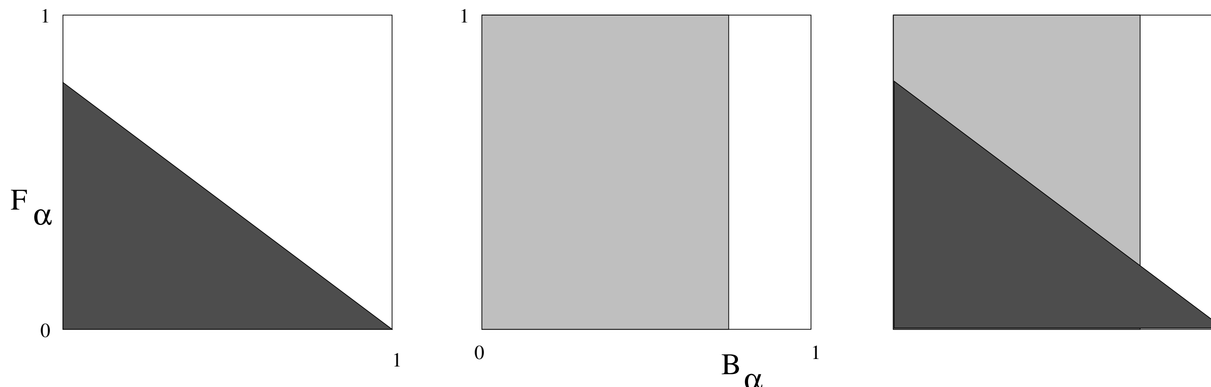
(F over B)$_\alpha$

To define $(F \text{ over } B)_\alpha$, we blindly assume a subpixel geometry that is illustrated in the following figure. The left figure illustrates the subpixel region that is assumed to be covered by the foreground image. The middle figure shows the subpixel region that is assumed to be covered by the background image. The figure on the right shows the subpixel region that is assumed to be covered by the combined image F over B. Of course, this particular geometry is not going correspond to what is actually present in most situations. We use this geometry *only as a way to motivate a formula for calculating* $(F \text{ over } B)_\alpha$.



[27] T. Porter & T. Duff - Compositing Digital Images Computer Graphics Volume 18, Number 3 July 1984 pp 253-259

An example of how the model can go wrong is shown below. The total of the foreground is the same as before, but because the geometry of the foreground is different, the amount of background that is covered will be different from the figure above.



The first figure is the one we use. It implies a model in which the α value (coverage, opacity) of the combined image is

$$(F \text{ over } B)_\alpha = F_\alpha + (1 - F_\alpha) B_\alpha$$

where F_α and B_α both depend on the pixel (x, y). The idea is that F covers some percentage F_α of the pixel and the background covers a percentage B_α of the remainder of the remaining fraction $1 - F_\alpha$ of the pixel.

(F over B)$_{rgb}$

We next turn to the rgb intensity values. Suppose we render part of the scene, i.e. compute the intensity of a point in the scene that projects to the pixel. If the alpha value of the pixel is less than 1, then the rgb intensity contributed by that pixel needs to be scaled by this α value, i.e. the surface will not contribute as much to the pixel intensity as it would if the whole pixel were covered by the surface.

You might be tempted to define:

$$(F \text{ over } B)_{rgb} = F_\alpha F_{rgb} + (1 - F_\alpha) B_\alpha B_{rgb}$$

that is, you weight the contributions of the foreground and background rgb images, each weighted according to the α value of the foreground image. However, notice that this is not quite correct, since it gives the final contributed rgb intensity of (F over B)$_{rgb}$, rather than the rgb color. To see the problem, consider the simple case that $F_\alpha = B_\alpha = \frac{1}{2}$ and $F_{rgb} = B_{rgb} = 1$. In this case,

$$(F \text{ over } B)_\alpha = \frac{3}{4},$$

and

$$(F \text{ over } B)_{rgb} = \frac{3}{4}.$$

But, if the rgb values are supposed to be the color (not weighted by α !), then in this case they should be 1, not $\frac{3}{4}$.

The problem is that the above definition of $(F \text{ over } B)_{rgb}$ gives the rgb intensity, not the color. To get the color, you could divide by the $(F \text{ over } B)_\alpha$, which is rather awkward.

Another approach is define F_{rgb} and B_{rgb} as *already* multiplied by F_α (or B_α). For example, suppose the RGB surface color for a polygon is computed to be $(.1, .8, .6)$ and the α value for a particular pixel is computed to be .5. Then the (r, g, b, α) image would be given values $(.05, .4, .3, .5)$.

If we use this premultiplied definition for rgb values, then we define $(F \text{ over } B)_{rgb}$ as follows:

$$(F \text{ over } B)_{rgb} \;=\; F_{rgb} + (1 - F_\alpha)\; B_{rgb}$$

Note that the operations are the same as we saw earlier for computing the α value. Thus, when we used premultiplied rgb values, we have:

$$(F \text{ over } B)_{rgb\alpha} \;=\; F_{rgb\alpha} + (1 - F_\alpha)\; B_{rgb\alpha}.$$

Examples

Lets consider a few examples of how α is used in image compositing.

- Consider $rgb\alpha$ 4-tuples: $(0, 0, 0, 0)$ and $(0, 0, 0, 1)$. The former is a completely transparent pixel; the latter is a completely opaque black pixel.

- How do we darken an image, without changing its coverage?

  $$darken(I_{rgb\alpha}, \phi) = (\phi I_r, \phi I_g, \phi I_b, I_\alpha)$$

  where $\phi < 1$.

- How do we change the opacity (coverage) of a pixel, without changing the color?

  $$dissolve(A, \delta) = (\delta A_r, \delta A_g, \delta A_b, \delta\alpha)$$

  This amounts to sliding the underlying surface away from the pixel so that it covers the pixel less.

Alpha and OpenGL

You may have noticed that OpenGL allows you to define surface material parameters (diffuse, specular) with an alpha value i.e. RGBA. Here the alpha value is referring to surface transparency, rather than partial occupancy of a pixel. If $\alpha = 0$, then the surface is fully transparent. If $\alpha = 1$ then the surface is opaque. If $0 < \alpha < 1$, then the surface is partly transparent.

OpenGL does not use premultiplied alpha for surface materials. That is, when you declare `glMaterialfv(GL_FRONT, GL_DIFFUSE, material)` where `material` is a 4-vector, the first three components specify the reflectance color of the material (assuming $\alpha = 1$) and the fourth component specifies the opacity α.

The surface α values are used in OpenGL in a similar way to how they are used in image compositing. But there are a few subtleties. Recall that OpenGL uses a depth buffer to decide whether to draw a surface at a given pixel. But once we allow for surfaces to be partly transparent, it is no longer clear what we mean by "the depth" at a pixel, and one needs to be very careful. We will have more to say about this next lecture.

Application 1: "Pulling a matte"

We have discussed how image compositing works when you know the $rgb\alpha$ images. In computer graphics, one can compute these images using methods we have discussed. (Although I have not gone into details of how subpixel geometry is estimated and α's are computed, you can probably imagine how this might be done.) But how could one acquire an $rgb\alpha$ image from images taken with a real camera? This is an important problem, since often one would like to insert real foreground videos of actors in front of computer graphics rendered backgrounds. Computing an α image from a real RGB image is called *pulling a matte*.[28] Let's take a more mathematical look at this problem.

Suppose you are given a background image or video B_{rgb} and you assume the background is fully opaque at all pixels, so $B_\alpha = 1$. For example, the background might be a blue screen, so the rgb values would be constant. You then film a real foreground scene in front of this background, for example, a person standing in front of the background. You obtain the image $(F \text{ over } B)_{rgb}$. Notice that $(F \text{ over } B)_\alpha = 1$, since we assumed earlier that $B_\alpha = 1$. You would like to composite the foreground scene over a different background, say B^* where again we assume that $B^*_\alpha = 1$. That is, you would like to compute $(F \text{ over } B^*)_{rgb}$. For example, you would like to show the weatherman standing in front of a big map.

To solve this problem, it is sufficient to compute $F_{rgb\alpha}$. Unfortunately this is non-trivial! The reason is that there are four unknowns to be solved at each pixel namely $F_{rgb\alpha}$, but there are only three measured values at each pixel, namely $(F \text{ over } B)_{rgb}$. That is, we have three equations

$$(F \text{ over } B)_{rgb} = F_{rgb} + (1 - F_\alpha)B_{rgb}$$

where the left side is measured, but we have four unknowns at each pixel namely $F_{rgb\alpha}$. The fourth equation

$$(F \text{ over } B)_\alpha = F_\alpha + (1 - F_\alpha)B_\alpha$$

reduces to

$$1 = F_\alpha + (1 - F_\alpha)1$$

which gives no information about F_α.

Notice that using a "pure blue screen" for the background, $B_{rgb} = (0, 0, 1)$ or using a black screen $B_{rgb} = (0, 0, 0)$ does not solve the problem. We still have four unknowns and only three equations. (Note that this problem didn't arise for the weatherman discussed at the beginning of the lecture, because we assumed that α was binary.)

One approach is to use a foreground scene that has neutral colors only, so that $F_r = F_g = F_b = F_*$, the exact value of which might vary from pixel to pixel. This gives us three equations

$$(F \text{ over } B)_{rgb} = F_* + (1 - F_\alpha)B_{rgb}$$

with only two unknowns, namely F_* and $(1 - F_\alpha)$, so we can solve for these unknowns at each pixel as well. In many movies where blue screen matting is used, neutral foreground objects are used (silver, grey).

[28] α channels are sometimes called *mattes*. This is not to be confused with *mask* which is binary only.

Application 2: Sports events (not on final exam)

Another application is in television broadcasting of professional sports, e.g. the *first down line* used in American football (NFL) [29]. The broadcasters draw a synthetic thick yellow line across the image of the football field. (This "first down line" has significance for the rules of the game, namely the team with the ball needs to bring the ball over this line.) The line appears to be "on the field" and is such that the line is hidden by any player that moves in front of it, consistent with hidden surface removal. See image below which should be viewed *in color*.

For any pixel and any frame of the video, the first down line is drawn at that pixel if two conditions are met: (1) the color in the video is sufficiently green i.e. color of grass, and (2) the pixel lies sufficiently close to a given line, namely the image projection of the first down line on the football field. Condition (1) is necessary so that the line is not drawn on top of the players. To achieve condition (2), the broadcasters need to enter by hand the position on the first down line on field. (On the real field, this position is marked by a linesman who holds an orange stick – see image above.) A homography is used to to map the position of the first down line on the field into camera coordinates and then into the image plane. For more information see
`http://entertainment.howstuffworks.com/first-down-line.htm`

[29] See www.sportvision.com, for examples.

Volume rendering

Many applications of computer graphics are concerned with making images of 3D data that represents a distribution of matter in a volume. A good example is medical imaging, where one has measured[30] a spatial distribution of bone, fat, muscle, tumour, etc in a volume and one would like to visualize this distribution somehow. Other examples include rendering fog or smoke in a scene. Often these *volume rendering problems* are addressed by treating the 3D volume as a set of slices, and blending the slices together. Let's first look at how this might be done, by extending the method developed last lecture.

Compositing layers in a volume

Suppose we have a $N \times N \times N$ cube of α and rgb color values $F_{rgb\alpha}(x, y, z)$, where the rgb values are premultiplied by the α values. The cells (x, y, z) of the cube are sometimes called *voxels* i.e. volume elements.

Consider the image formed by orthographic projection in the z direction. Let $z = 1$ be closest to the camera and $z = N$ be furthest from the camera. We consider

$$F(x, y, 1) \text{ over } F(x, y, 2) \text{ over } \ldots F(x, y, N-1) \text{ over } F(x, y, N)$$

where I have dropped the subscript $rgb\alpha$. The reason I can do this is that the formula for combining the layers is the same for each of these variables, as we saw last class. I also drop the x, y symbols to keep the notation simple.

Notice also that I have not specified which "over" operation is performed first. This is allowed here, since the "over" operation is associative. (See Exercises 3).

We can compute the above expression *front to back* as follows. Let $I(k)$ be the result of compositing layers 1 to k. We wish to compute $I(N)$, namely we wish to composite layers 1 to N, where

$$\begin{aligned} I(1) &= F(1) \\ I(k+1) &= I(k) \text{ over } F(k+1) = I(k) + (1 - I_\alpha(k))F(k+1) \end{aligned}$$

where $I_\alpha(k)$ is the α component of $I(k) = I_{rgb\alpha}(k)$. Note that there is an advantage of solving from front-to-back instead of back-to-front, namely once the opacity at a pixel (x, y) has accumulated to 1 (or near 1), there is no need consider deeper layers at that pixel. The reason is that these deeper layers will not be visible in the image.

It is not difficult to show using induction that

$$I(k+1) = F(1) + \sum_{i=2}^{k+1} F(i) \prod_{j=1}^{i-1} (1 - F_\alpha(j))$$

This formula becomes especially interesting when the $F_\alpha(j)$ are small for each j (e.g. when have fog – see below). In this case, we use the fact that $e^x \approx 1 + x$ when x is very small, and rewrite the above formula as

$$I(k+1) = F(1) + \sum_{i=2}^{k+1} F(i) \prod_{j=1}^{i-1} e^{-F_\alpha(j)}$$

[30] Using CT, MRI, ...

Fog

Let's consider the case of a natural scene in which there is a small amount of fog or mist. The scene contains surfaces whose depths are at $z(x, y)$, and whose colors are $F^{surf}_{rgb}(x, y, z)$. The surfaces are opaque so $F^{surf}_{\alpha}(x, y, z) = 1$ for the surface voxels, and so the color of the surface is the same as the premultiplied color $F^{surf}_{rgb}(x, y, z)$.

We also have a fog function $F^{fog}_{rgb\alpha}(x, y, z)$ which is defined over all non-surface voxels (x, y, z). Recall that the F_{rgb} values are premultiplied by the F_α values – in particular, since fog voxel has a very small (but non-zero) α, the premultiplied F^{fog}_{rgb} values are very small as well.

One simplification which is made in OpenGL to model fog is to assume F^{fog}_{α} is constant (say α_0). We will make this assumption in the following. Suppose for a pixel (x, y), the visible surface is at depth $z = k + 1$. Then, by inspection, we would write the above equation as:

$$I_{rgb}(k+1) \ = \ F^{fog} \sum_{i=1}^{k} \ e^{-(i-1)\alpha_0} + F^{surf}(k+1) \ e^{-k\alpha_0}$$

But, since the sum is of the form $1 + x + ... + x^{k-1}$, we can use the fact that

$$\sum_{i=1}^{k} \ e^{-(i-1)\alpha_0} = \frac{1 - e^{-k\alpha_0}}{1 - e^{-\alpha_0}}$$

to rewrite the above as

$$I_{rgb}(k+1) \ = \ F(k+1)e^{-k\alpha_0} + \frac{F^{fog}}{1 - e^{-\alpha_0}}(1 - e^{-k\alpha_0}).$$

Recall that $F(k + 1)$ is just the surface color. Also, note that $1 - e^{-\alpha_0} \approx \alpha_0$ and recall that F^{fog} is the premultiplied color of the fog. So, $\frac{F^{fog}}{1-e^{-\alpha_0}}$ just undoes the premultiplication and gives us the "color" of the fog. Thus, $I_{rgb}(k+1)$ is just a blending (weighted average) of the colors of the surface and fog.

Such a formula should not be surprising, in hindsight. The attenuation of the light from a surface through the fog is $e^{-\alpha \ z(x,y)}$, i.e. exponentially decreasing with distance travelled through the fog. We can think of the attenuation factor as the fraction of the pixel at (x, y) that is covered by the fog (and blocks the light). This part of the pixel that is covered by the fog also contribute an intensity, which is related of the color of the fog.

One often writes this formula explicitly in terms of depth, and indeed OpenGL offers a fog model which makes use of the depth buffer:

$$I_{rgb}(x, y) \ = \ F^{surf}_{rgb} e^{-z(x,y)\alpha} + F^{fog}_{rgb}(1 - e^{-z(x,y)\alpha}).$$

where this now written in *non-premultiplied* form.

Transfer functions

Let's now turn to a different problem, which arises when we are trying to "visualize" the 3D distribution of some material or property of the world. This could be a 3D medical image, for

example. Let's suppose that we have some 3D data that is defined by a value v at each spatial location (x, y, z) in a 3D cube, so we have $v(x, y, z)$. We typically represent v as a 3D matrix.

One approach to visualizing v is to choose an opacity α at each voxel (x, y, z), based on the measured value $v(x, y, z)$. This choice is not meant to be "physically correct", for example, in the case of a medical image, the body is opaque ($\alpha \equiv 1$). Typically the measured values $v(x, y, z)$ represent "densities" of some particular material in the volume. For example in medical imaging different ranges of v correspond to different types of tissue: bone vs. soft tissue[31] vs. air.

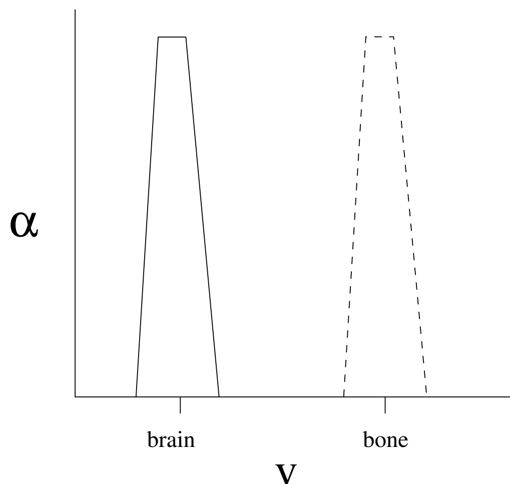
In general, one defines a *transfer function*

$$\alpha : v \rightarrow [0, 1]$$

that maps the volume data $v(x, y, z)$ to opacity values $\alpha(x, y, z)$ which are used for visualization purposes. For example, we might use a linear transfer function, e.g. $\alpha = av + b$, where the constants a, b might be chosen interactively by a user to try to get a "good visual effect".

Rendering of iso-surfaces

A common situation is that one wants to render the (x, y, z) points that have a certain range of v values only, e.g. to make an image of the bones but not the soft tissue, or vice-versa. In this case, we would define α values to be non-zero only near these selected v values. Such a transfer function would typically have a single maximum in the middle of the range of desired values. For example, suppose we had a CT image of a head such that a certain range of values of v corresponded to bone and a different range of values corresponded to brain. We could render two different images (brain vs. bone) by using two different transfer functions (see sketch below).



The opacity values determine which of the voxels are seen in the rendering. This does not yet specify what the colors of the voxel are, however. One interesting approach to choosing the *rgb* color at each (x, y, z) is to render the volume using the Phong lighting model. This requires choosing a surface normal at each point. At first glance, this might seem very strange since there is no meaning

[31] skin,muscle,...

to "surface normal" at a voxel (x, y, z) in a 3D volume. One way to think of the surface normal is to consider the set of points $\{(x, y, z) : v(x, y, z) = v_0\}$ for each fixed v_0. If the $v(x, y, z)$ is sufficiently smooth, then this set of points defines a smooth 2D surface, called the *iso-surface* of value v_0. Such a surface has a well-defined surface normal, namely the 3D gradient of v

$$\nabla v = (\frac{\partial v}{\partial x}, \frac{\partial v}{\partial y}, \frac{\partial v}{\partial z})$$

as long as the gradient is non-zero, i.e. we are not at a local maximum or minimum of v.

In practice, we need to estimate the gradient using discrete derivatives, e.g.:

$$(\frac{\partial v}{\partial x}, \frac{\partial v}{\partial y}, \frac{\partial v}{\partial z}) \approx (v(x+1, y, z) - v(x-1, y, z),\ v(x, y+1, z) - v(x, y-1, z),\ v(x, y, z+1) - v(x, y, z-1)\ ).$$

Once we have the normal at (x, y, z), we can compute a color at this point using our favorite lighting model (e.g. Phong). To use the compositing formulas above, we multiply this color by the opacity so that the resulting F_{rgb} is "pre-multiplied."

Note that, although we are defining rgb values at each point, typically only a relatively small fraction of points (x, y, z) will play a role in the final image since we will be setting the opacities α of most points to near 0 (see transfer functions in the sketch above). An example[32] of several such renderings is shown below. Two pairs of images are shown, corresponding to different transfer functions and different viewing directions.

Levoy

Ray casting and trilinear interpolation

So far today, we have simplified the situation by assuming that the scene was viewed using orthographic projection along the z axis. Often when we are examining a volume of data, however, one would like to view it from an arbitrary direction and position, that is, view it in perspective. Here we can apply the standard methods of ray casting.

Suppose we have transformed the scene into camera coordinates and we have chosen an image plane (i.e. projection plane). For each pixel (x_p, y_p), a ray is cast though the corresponding position in the projection plane and into the volume to be rendered.

RAYCAST VOLUME

To use the compositing method similar to above, we choose a front and back clipping plane and a discrete set of depth layers in between. We then compute where the ray intersects each of these depth layers. We wish to compute opacity values and colors at these points.

Because these intersection points will not correspond exactly to (x, y, z) voxel corners in the original volume, we will need to interpolate. We are in 3D, we could use *trilinear* interpolation.

[32]These were computed in the late 1980's by Marc Levoy as part of his PhD thesis. See Levoy's current home page at Stanford for more information and links to the original papers (which are nicely written).

Trilinear interpolation is just the natural extension of linear (1D) and bilinear (2D) interpolation into 3D. (See Exercises 3.) In the sketch above, the dotted line is the ray we are casting and the tick marks along the dotted line are sampled (constant z) depth planes. The points where the ticks meet the dotted rays are the points (x, y, z) where we may to know the $F_{rgb\alpha}$ values. We need to interpolate these values from the points on the cubic grid (only a square grid is shown).

Color

We have seen that digital color images have three intensity values per pixel (RGB). In OpenGL, these values are calculated based on the surface RGB reflectance values and light source RGB intensity values.

In discussing color in the next two lectures, we are not going to be concerned with how scenes have been rendered into digital RGB images – we have already discussed this is great detail. Instead we going to be concerned with what happens when the image is displayed, either on a monitor or on a projection screen, and when it is capture by some imaging system – either the eye or a camera. This requires that we consider not just digital (RGB) colors, but also *physical* aspects of color.

Wavelengths of light

A good place to start our examination of physical color is with Isaac Newton and his prism experiment (late 17th century). Newton observed that when a beam of sunlight is passed through a prism, the beam is spread out into a fan of different color lights – like a rainbow. He argued, based on this experiment and many variations of it, that a light beam from common sources such as the sun or a flame is composed of a mixture of colored lights from many superimposed beams.

The theory that explains Newton's experiments is now well known. Roughly, light is composed of electromagnetic waves, with wavelengths ranging from 400-700 nanometers.[33] The intensity of a light ray can be thought of as a function of wavelength (called a *spectrum*).[34]

Photoreceptor cells

If a light ray has a spectrum of intensities (a many dimensional vector), then why do we represent colors using only three dimensions (RGB) ? The reason is that human vision is *trichromatic*, that is, the human perception of light is restricted to three dimensions only. What does this mean? At the back of our eyes is an array of *photoreceptor cells* that respond to light. There are two classes of such cells. The first class, called *cones*, respond a high light levels such as during the day. These are the ones that are used in color vision.[35]

The cone cells come in three types called L, M, and S, where L is for long wavelength (red), M is for medium wavelength (green) and S is for short wavelength (blue). You can think of them as RGB cones, if you like. Each cone type is defined by a pigment (a protein) that absorbs certain wavelengths of light better than it absorbs others. e.g. the pigment in the L cones absorbs longer wavelength light better than medium or shorter wavelength light.

When a pigment absorbs light, a physical response is triggered inside the cell. This response increases with the *total* light absorbed. The response does not depend on the spectral composition

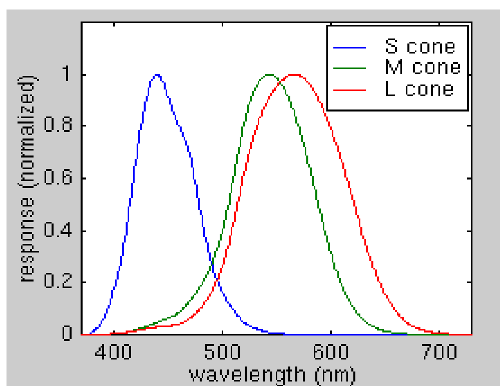
[33] A nanometer is 10^{-9} meters. e.g. for a wavelength of 500 nm, you need about 2000 "waves" to cover one millimeter.

[34] For the light emitted by natural light sources such as the sun or a candle or even burning metal (a tungsten light bulb), the spectrum is relatively smooth function of wavelength. For some artificial sources such as fluorescent lights in our classroom, the spectra are not smooth at all but rather have spikes at certain wavelengths.

[35] There is also a second class of photoreceptor cells, called *rods*, that respond at low light levels such as at night. These do not play any role in color vision, however. At low light levels, we do not see in color. This is because only the rod cells can measure such light levels. This is the reason we do not see color at night.

of the light absorbed by each cell, but rather only on a weighted sum of light intensity at different wavelengths.

The three curves below (figure is in color) show the weights that determine the responses of each of the three cone types to the range of wavelengths of visible light. Each of the curves has been normalized so that its maximum is 1. These curves are called *spectral sensitivity functions*. They are the weights that different cells give to different wavelengths.



The wavelength variable λ is a continuous variable, but in practice the spectra are smooth enough functions of λ that one can approximate the spectra very well using a finite dimensional vector. One represents the interval 400 to 700 nanometers into N_λ bins, for example, 30 bins each having a 10 nanometer range. We approximate the spectral response function as constant within each bin.

Similar response sensitivities can be defined for the photoreceptor "cells" (pixels) in digital cameras.[36] In the remainder of this lecture, I will use the term *cell* but not distinguish whether I am referring to a human photoreceptor cell or to a camera photoreceptor cell.

Linear model of cell responses

Suppose that a particular cell is measuring the total light from some beam of light rays that enters the eye. Let $I(\lambda)$ be the spectral intensity of the beam as a function of wavelength λ. (To relate this to what we discussed last lecture, recall the accumulation buffer – but now imagine that we are considering the summed[37] light spectrum of rays that land on some pixel/cell at the back of the eye/camera.)

The three cell types filter (or re-weight) the spectrum as described qualitatively above. We now formally describe this reweighting, as follows. Consider a $3 \times N_\lambda$ matrix $\mathbf{C}$, whose three rows are C_L, C_M, and C_S, respectively. Each row specifies the relative spectral sensitivity of each cone type, i.e. the three functions shown above in the case of the eye. The LMS response depends on the total

[36] For a digital camera, typically RGB filter arrays are placed in front of the camera's sensor array. An example is the Bayer array – http://en.wikipedia.org/wiki/Bayer_filter. Each filter preferentially transmits the light of different wavelengths, e.g. a red filter transmits longer wavelengths better than shorter wavelengths, so the sensor behind a red filter will "see" spectra in which the longer wavelengths are more heavily weighted.

[37] minor point: with the accumulation buffer, we were considering the average i.e. we divided by NUMDELTA

light absorbed by the cell and this can be represented as:

$$\begin{bmatrix} I_L \\ I_M \\ I_S \end{bmatrix}_{3\times 1} = \begin{bmatrix} C_L \\ C_M \\ C_S \end{bmatrix}_{3\times N_\lambda} I(\lambda)_{N_\lambda \times 1} \quad (10)$$

The key property to understand is that the response of a cell depends only on the weighted sum. Once a cell has absorbed light energy of any wavelength, the cell has no "memory" of what that wavelength was, or more generally, what the distribution $I(\lambda)$ was. Given the weighted integrated spectra $I_L,\ I_M,\ I_S$ of the three cell types, it is impossible to "invert" the above equation and compute the N_λ dimensional light spectrum $I(\lambda)$.

Color matching and color displays

One key implication of the above linear model is that if two spectra I_1 and I_2 produce the same responses in the three types of cells, i.e.

$$\mathbf{C}I_1 = \mathbf{C}I_2$$

then these two spectra produce identical images and hence are indistinguishable to the camera/eye. (Such spectra are called *metamers.*)

Let's apply this idea of color matching to a practical problem. Suppose we have two different *displays* (say, computer monitors) and some RGB image. We would like to display this image on both displays such that the two displayed images looks exactly the same to the eye. That is, the two spectra produced by corresponding pixels are indistinguishable to the two arrays of RGB cells in the eye.

To understand this problem, we need to know something about how displays work. [38] Each display emits three spectra of light, corresponding to the RGB channels of images. By putting larger or smaller RGB value at any pixel, one does not change the shape of the corresponding spectrum, but rather one only scales the spectrum by multiplying the whole spectrum by a constant. Let's refer to these constants as $\mathbf{s}$, keeping in mind that there are three of them at each pixel (RGB).

The spectral distribution of light emitted by the RGB display elements can then be represented by the three columns of a $N_\lambda \times 3$ matrix $\mathbf{D}$. Let the strength of these three emitters be 3×1 vector $\mathbf{s}$. Then, the sum of the three spectra at each pixel is $\mathbf{Ds}$, which is the spectrum leaving that pixel on the display.

Returning to our problem, suppose we have two different displays with RGB emission spectra $\mathbf{D}_1$ and $\mathbf{D}_2$ respectively. These two displays could be made by different companies, for example. Suppose we want corresponding pixels on the two displays to be a color match, that is, we want the two displayed images to look exactly the same, regardless of what image $I_{RBG}(x, y)$ is shown on the screen. For this, we need to choose $\mathbf{s_1}$ and $\mathbf{s_2}$, for each pixel, such that

$$\mathbf{C}\ \mathbf{D}_1\ \mathbf{s_1}\ =\ \mathbf{C}\ \mathbf{D}_2\ \mathbf{s_2}$$

[38] If you wish to learn more about how a CRT works, check out e.g.

`http://en.wikipedia.org/wiki/Cathode_ray_tube`

LCD displays are more complicated than this, but the details don't concern us here.

Since $\mathbf{CD}_1$ and $\mathbf{CD}_2$ are 3×3 matrices, we require

$$(\mathbf{C}\ \mathbf{D_1})^{-1}\ (\mathbf{C}\ \mathbf{D_2})\ \mathbf{s_2}\ =\ \mathbf{s_1}.$$

Given weights $\mathbf{s_2}$, this result tells us how to choose the weights $\mathbf{s_1}$ of the monitor 1 so that they produce a color match between monitors 1 and 2. (Similarly, if we are given $\mathbf{s_1}$, we could choose $\mathbf{s_2}$ to get a match.) If the $\mathbf{s}_1$ weights are chosen this way for each pixel of monitor 1, then the image is guaranteed to appear exactly the same[39] as the image on monitor 2.

Notice that this equation does not guarantee that for any $\mathbf{s_2}$, the three corresponding $\mathbf{s_1}$ will be positive numbers. Indeed, if the $\mathbf{s_2}$ are positive numbers, but the $\mathbf{s_1}$ are not all positive numbers, then we cannot produce a match.

Monochromatic spectra

We can approximate an arbitrary continuous spectrum $I(\lambda)$ as a finite sum of *monochromatic* spectra. We use the term "monochromatic spectra" to refer to spectra that have non-zero values in only one of the N_λ wavelength bins. Such spectra don't occur naturally, but they can be created using lasers. More important, in our linear algebra model, these monochromatic spectra define the **canonical basis** vectors $\mathbf{e}_m$ of the space of the N_λ dimensional set of intensity spectra.

We approximate any spectrum $I(\lambda)$ as:

$$I(\lambda) \approx \sum_m I_m \mathbf{e}_m$$

where the index m runs over the N_λ wavelength bins, $\mathbf{e}_m$ is a canonical basis vector, namely it has a "1" at the mth element (which corresponds to one of the N_λ wavelength bins) and "0" in all other elements. The scalar I_m is the (mean) value of the spectrum $I(\lambda)$ over the mth wavelength bin. This approximation is excellent if N_λ is large.

Then,

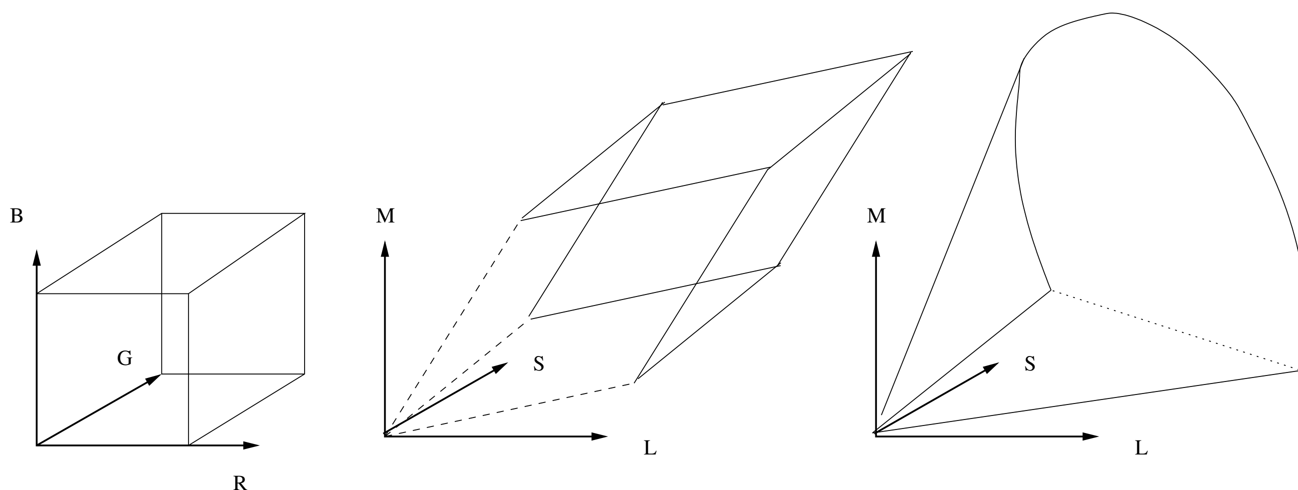
$$\mathbf{C} \sum_m I_m \mathbf{e}_m = \sum_m\ I_m\ (\mathbf{C}\mathbf{e}_m)\ .$$

We see that the LMS coordinates of any spectrum I is just a linear combination of the LMS coordinates of the monochromatic spectra. The weights of this combination are positive, so we can think of any spectrum as a weighted average of monochromatic spectra. In this sense, the monochromatic spectra define a bound or envelope for all possible spectra. In LMS space, the monochromatic spectra define a curved envelope (a 2D surface) that surrounds the possible LMS coordinates of all possible (non-negative) spectra.

The figure below helps to illustrate these ideas. For a given camera/cone sensitivity matrix $\mathbf{C}$ and display matrix $\mathbf{D}$, consider $\mathbf{s}$ values from 0 to 1, i.e. the triplet $\mathbf{s}$ can take values in the unit cube $[0,1] \times [0,1] \times [0,1]$. The values of $\mathbf{CDs}$ lie in a paralleliped, contained in the positive octant in LMS space. That is, the canonical unit basis vectors of $\mathbf{s}$ map to three vectors in LMS space and the values of $\mathbf{s}$ in [0,1] define a paralleliped (see middle sketch).

If two displays have different RGB spectra, i.e. $\mathbf{D}_1 \neq \mathbf{D}_2$, then their parallelipeds ($\mathbf{CD}_1\mathbf{s_1}$ and $\mathbf{CD}_2\mathbf{s_2}$) do not in general coincide. This means there are some colors (or, more precisely, LMS triplets) that can be produced on one monitor but that cannot be produced on the other.

[39] *to a camera/eye defined by cell sensitivity C*



The sketch on the right illustrates a smooth surface, defined by the the set $\{\mathbf{C} I_m \mathbf{e}_m\}$ where $\mathbf{e}_m$ are the monochromatic spectra. The paralleliped defined by any display $\mathbf{D}$ will always lie in the volume bounded by this smooth surface.

Illumination and reflectance spectra

The light coming from a real source such the sun, a light bulb, or a blue sky have characteristic spectra $I_l(\lambda)$. Surfaces in the scene then reflect this light, and the reflected light is measured by the eye or camera. The reflected light does not have the same spectrum of the light from the source. Rather, this spectrum has been re-weighted wavelength-by-wavelength by a "reflectance spectrum". Here we can use the same notation that we used in the Phong lighting model, but now consider the full spectrum instead of just RGB, e.g. for the diffuse component we consider $k_d(\lambda)$, where $0 \leq k_d(\lambda) \leq 1$ for all λ. For the diffuse component, the light that is reflected from the surface thus has a weighted spectrum $I(\lambda) = I_l(\lambda) k_d(\lambda)$.

1 EXTRAMATERIAL ?

Following the discussion above, the eye or camera filters this reflectance spectrum:

$$\begin{bmatrix} I_L \\ I_M \\ I_S \end{bmatrix} = \begin{bmatrix} C_L \\ C_M \\ C_S \end{bmatrix}_{3 \times N_\lambda} (I_l(\lambda) k_d(\lambda)) = \begin{bmatrix} \sum_\lambda C_L(\lambda) I_l(\lambda) k_d(\lambda) \\ \sum_\lambda C_M(\lambda) I_l(\lambda) k_d(\lambda) \\ \sum_\lambda C_S(\lambda) I_l(\lambda) k_d(\lambda) \end{bmatrix}$$

An immediate observation is that there are far more variables in this equation than there are measurements. The weighted spectra measured by the three cone types in the eye (or by the RGB cells of a digital camera) define just three values, and these three values are determined by the source spectrum and the surface reflectance spectrum as well as the filtering done by the eye. This situation is much more complicated than the naive situation that we considered in the Phong lighting model where the RGB intensity was determined by $(I_l k_d)_{rgb}$ that is, by source triplets $(I_l)_{rgb}$ and reflectance triplet $(k_d)_{rgb}$.

2 EXTRA ???

What do the chromaticity coordinates mean ? Consider we have some intensity spectrum, I, and it has a chromaticity coordinates (x0,y0). If (x0,y0) is not the white point, then (x0,y0) occurs along a unique line from the white point to a point the U shaped monochromatic curve in (x,y) space. We can say that the spectrum, I, is a color match to the sum of two spectra: a white spectrum (constant over wavelength) and a monochromatic spectrum, with wavelength defined by the monochromatic point intersected by the line segment.

$$CI = CI_{white} + CI_i, for some wavelength bin i.$$

That wavelength is often (loosely) called the dominant wavelength of the spectrum. I say "loosely" because the actual spectrum "I" does not necesarily have a peak intensity at the wavelength. The color name associated with that wavelength is the hue of the spectrum I. Notice that the chromaticity can lie anywhere along the line segment joining the white point to the monochrome point. The position along this line is called the saturation of the spectrum. If the chromaticity is near the white point, we say it has "low saturation". If it is near the monochrome point we say it has "high saturation".

Two spectra are said to be complementary if their sum has the chromaticity of (1/3, 1/3) i.e. the white point. For this to happen, the chromaticities of the spectra have to lie on opposite sides of the white point.

Application to CRTs

RGB Color Model (13.3.1) You will need to know the following for the final exam !

What is the range of XYZ values reached by a given CRT ? To answer this, let the three columns of the matrix P represent the three spectra of the phosphors when each of the guns is emitting its maximum amount of electrons. In a general situation, we can then let the intensity of the guns take values from 0 to 1. 0 means the gun is off and 1 means it is turned on to its maximum. The range of XYZ is a paralleliped in XYZ space.

To transform from gun strengths (RGB) to XYZ coordinates, you use the equation

$$(X, Y, Z)^T = C_X Y Z P e.$$

The matrix product "$C_{XYZ}P$" is a 3x3 matrix which maps from the unit cube of RGB values (gun strengths) to XYZ space. This is known as the RGB-to-XYZ matrix for a CRT monitor.

Each column in the RGB-to-XYZ matrix represents the XYZ coordinates of one of the guns. The first column represents the XYZ coordinates of the R gun (turned to maximum intensity). The second column represents the XYZ coordinates of the green gun. The third column represents the XYZ coordinates of the blue gun.

Once we have the RGB-to-XYZ matrix, we can compute the chromaticities (x,y) of the three guns. This defines three points in chromaticity space. The triangle defined by the three points define the range of chromaticities that can be presented by mixing together the light from the different phosphors, that is, the range of chromaticities that can be reached by the gun. This is important for comparing two CRTs. If the chromaticity triangles of the two CRTs are not identical, then the set of colors that can be displayed by the two CRTs are not identical. That is, you will be able to

display colors on one the CRTs that cannot be matched to any any color displayable on the other CRT. The chromaticity triangle of a given CRT monitor is called the gamut of the monitor.

Make an analogy here to depth vs. image coordinates. Color vision in perspective. The overall strength of the spectrum is related to the brightness. The chromaticity has to do with the up and down direction of color space.

What about the ambient light? What about homogeneous coordinates ?

Image Capture

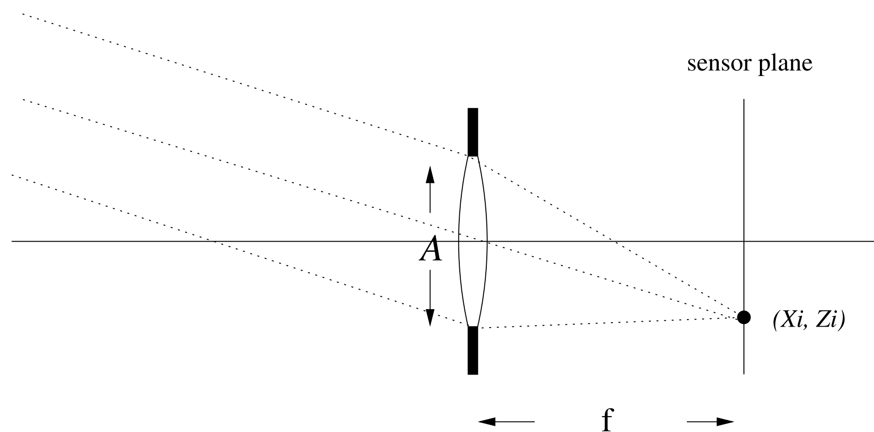
It is very common in computer graphics to use digital images that have been captured from real scenes. One well known example is texture maps and environment maps. But other methods are emerging as well, including clever ways of video to capture motion for animation, or even to combine image capture and projection in live theatre.

Today, we look a few basic but fundamental issues in how images are captured. We begin by reviewing the concept of *exposure* in photography.

Aperture

The lens/sensor system we described in lecture 21 had a lens with focal length f and a sensor at a distance z_s from the center of the lens. As discussed in lecture 21, a lens with focal length f only brings one depth plane into focus (that is, substitute z_s for z_i in the thin lens formula and solve for z_s^*). Object points that do not lie on depth plane z_s^* will not be in focus, in that the rays from such object points will not converge on a single point in the sensor plane.

To keep things simple today, let's suppose that the sensor is at a distance $z_s = f$ from the lens. In this case, observe from the thin lens equation that the surfaces that are in focus are those that are at $z_o = \infty$ (see figure below). Surfaces that are a finite distance away have a certain blur width on the lens, but we ignore this today: blur is not our main concern today.



In the figure above, the *aperture* A is the width of the region of the lens through which rays are allowed to pass. In typical cameras, the user can manually vary the aperture. One often refers to the aperture A by "$f/\#$". The symbol # is called the "f-stop" or "f-number" and is defined by

$$\# = \frac{f}{A}.$$

Thus, for example, if one is using an $f = 80mm$ lens and one chooses an f-number 8.0, then the aperture is 10 mm, i.e. $A = f/8.0 = 80/8 = 10$ mm.

Exposure

The total amount of light that reaches each photoreceptor (pixel) on the sensor surface depends on the "number of rays" that arrive there. This number is proportional to the angle subtended by the lens aperture, as seen from that pixel. Because the lens aperture is 2D, one needs a 2D notion of angle which describes an area of the unit sphere, rather than an arc length on a unit circle. Here the term *solid angle* is used. A solid angle is a surface area on the unit sphere of directions. The unit of solid angle is *steradians* – note the similarity to the term *radians* which is used for an angle measured by an arc of a unit circle. Note: the unit sphere has total area (solid angle) of 4π steradians, and a hemisphere has a solid angle of 2π steradians.

The solid angle of the lens aperture seen by a point on the sensor plane is approximately proportional to A^2 (since the area of a circular aperture is πA^2) and is inversely proportional to the square of the distance of the lens from the sensor plane, which are assuming now is f^2. Thus, the number of rays reaching a point on the sensor plane is proportional to $(\frac{A}{f})^2$. Notice that $\frac{A}{f}$ is the inverse of f-number. Thus, the amount of light reaching a pixel varies with the inverse of f-number squared.

Another important factor in determining the amount of light reaching each sensor is the time duration t over which the sensor is exposed to the light. The inverse of t is called the *shutter speed.* A shutter speed of 60, for example, means that the aperture ("shutter") is open for 1/60 second.

Finally, recall that we are assuming the surface is far away ($z_0 = \infty$) and the sensor plane is at $z_s = f$, and so each of the rays reaching a pixel has the same spectrum $I(\lambda)$. This spectrum is weighted by some color sensitivity matrix $\mathbf{C}_{rgb}(\lambda)$ as discussed last class, and by the proportionality factors above.

Putting all these factors together, we get an expression for the *exposure* ξ, which is the total amount of light measured by the sensor over the time duration t:

$$Exposure \xi = (\frac{A}{f})^2 \; t \; \mathbf{C}_{rgb} I(\lambda)$$

Note that $\mathbf{C}_{rgb}$ will be a different function from what we saw in the lecture 22 (for one thing, we give it the subscript rgb rather than LMS) since we are now talking about camera photoreceptors rather than the cones in the human eye.

It is standard in photography to allow the user to vary the shutter speed ($\frac{1}{t}$) roughly in powers of 2, namely, $1, 2, 4, 8, 15, 30, 60, 125, 250, 500, 1000$.[40], and to allow users to vary the $f{-}numbers(\#)$ in powers for $\sqrt{2}$, namely: $1, 1.4, 2, 2.8, 4, 5.6, 8, 11, 16, 22, 32, 45, 64, 90, 128$. The reason for the latter is that the amount of light varies with the square of $\frac{A}{f}$ and so if you increase $\frac{f}{A}$ by a factor of $\sqrt{2}$ then you decrease the amount of light by a factor of 2.

In particular, note that increasing the shutter speed $\frac{1}{t}$ by one step corresponds to decreasing the amount of light by a factor of 2, and this has the same effect as increasing the f-number by one step. Thus, for example, one can tradeoff the depth of field (i.e. the blur, which depends on the aperture) with the effects of camera shake or scene object motion, which depend with shutter speed.

[40] As computer scientists, you should note that the "correct" numbers really should be $1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024$, at least, if the intent is to use powers for 2.

Camera response function and dynamic range

The RGB intensity that you get at a pixel is not identical to the exposure. Why not? In many scenes that you wish to photograph, the range of darks to lights that are present is too great to be captured in a single image. Some regions of the image will either be overexposed (meaning that these regions are mapped to the maximum intensity, say 255) or underexposed (meaning that all intensities are mapped to 0). One of the big challenges of shooting a good photograph is avoiding areas that are under or over exposed. This requires choosing the camera position and orientation appropriately, and also setting the camera parameters: the focal length, the aperture, and the shutter speed.

The rgb values of the image produced by the camera depend on the exposure ξ via some non-linear response function ρ. Typically this function ρ is the same for rgb. Then,

$$I_{rgb} = \rho\ (\xi_{rgb}\ )$$

where

$$\rho : [0, \infty) \rightarrow \{0, 1, 2, \ldots, 255\}.$$

The function $\rho()$ typically has an S shaped curve, as the sketch on the left on the next page illustrates. The function takes the value 0 for low exposures (underexposed) then ramps up over some range of exposures, and then maxes out at 255 beyond some maximum exposure (overexposed).

Note that the function $\rho()$ is not continuous, but rather it consists of 256 discrete steps. This is because the range of $\rho()$ is the set of 8 bits numbers from 0 to 255. Each step thus corresponds to a range of exposures which all map to the same 8 bit value. This will matter when we take the inverse of $\rho()$ below since, given an 8 bit intensity value, we would like to infer the exposure. We cannot do it exactly. Rather we can only choose a discrete sampling of 256 exposures corresponding to the 256 image intensities.

The *dynamic range* of a camera is the range of exposures values that can be distinguished by the camera (up to quantization effects that are due to finite precision i.e. 8 bits per rgb). This is roughly the range of exposure values that do not under or overexpose the pixels. Typically one refers to the dynamic range as a ratio of max:min exposures that the camera can measure (without over or underexposing). Similarly, we refer to the dynamic range of a scene to be the max:min ratio of exposures that this scene produces.

High dynamic range imaging (HDR)

Photographers are often faced with scenes have a dynamic range that is greater than the dynamic range of the camera. There is no way to shoot such a scene without over or underexposing some pixels. Recently, methods have been developed that attempt to expand the dynamic range of images.[41]

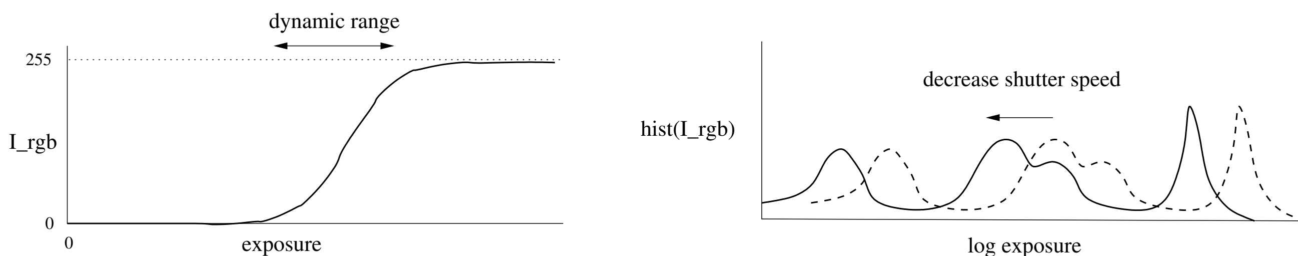
The main idea of *high dynamic range imaging* is to take multiple images with different t values. To properly expose the darker regions of the scene, you would need a slower shutter speed (smaller $\frac{1}{t}$) whereas for very bright regions of the image you would need a very fast shutter speed (large $\frac{1}{t}$).

[41]"Recovering High Dynamic Range Radiance Maps from Photographs" by Paul Debevec and Jitendra Malik, SIGGRAPH 1996.

To understand better what is happening here, consider the log (base 2) of the exposure,

$$\log \xi_{rgb} = \log(\mathbf{C}I(\lambda)) + \log(t) + 2\log(A/f)$$

and note that changing the shutter speed just shifts the log of the exposure. If the shutter speeds are consecutive factors of 2 (which is typical with cameras), then doubling the exposure time t decrements the log exposure (at each pixel) by 1. This is illustrated in the figure below right, which shows the *histogram* (a frequency plot i.e. in this case the number of pixels at each quantized log exposure value.). The histogram gets shifted to the left when you decrease the shutter speed.



Since any pixel will be properly exposed for *some* value of t, if one has photographs over a sufficiently large range of shutter speeds, then one can in principle infer the exposure ξ_{rgb} values via:

$$\rho^{-1}(I_{rgb}) = \xi_{rgb}.$$

i.e. for any pixel, there should be some photograph in which the exposure at that pixels falls into the operating range of the sensor.

Response function of the human eye (and visual system)

Is there a similar non-linear "response function" in the human eye, and more generally in human perception of brightness and color? This turns out to be a more complicated question than you might think. The "answer" depends on whether you are talking about photoreceptor responses, or certain physiological activity in various visual parts of the brain, or whether you are talking about perception i.e. asking people questions about the brightnesses they see and having them answer questions.

Let's address one aspect of this problem, namely perception of brightness and how well people can distinguish different levels of light intensity when you ask them. This will be important next lecture, when we discuss displays.

Consider the figure below, which I made using a 16×16 array of squares of different grey levels. (The exact intensities of light coming off these squares depends on the properties of your display system – printer or monitor – see next lecture. For now, let's just assume that the light intensities leaving the display are linearly increasing from left to right in each row i.e. there are 256 grey levels shown and they increase in scan order.)

A surprising (but very important) fact about human vision is that our ability to detect differences in the intensity ΔI from one square to the next is not uniform across intensity levels, but rather it *depends on the intensity* I itself. In psychology, one refers to people's ability to detect differences

by a *just noticeable difference* (JND), which the difference[42] in intensity that is big enough for you to notice. Typically the JND depends on $\frac{\Delta I}{I}$ and so the JND *is much smaller at smaller intensity levels.* For more details, have a look at Weber's Law in wikipedia.

This phenomenon – that you are more sensitive to differences at lower intensities levels – is very common in perception (not just vision, but also for all our senses). For example, consider our perception of object weight. Objects that have greater mass *feel* heavier, obviously. However, the perceptual difference in weight between a 1 kg object and a 2 kg object is not the same as the perceptual difference between a 20 kg object and an 21 kg object, even though in both cases the difference is 1 kg. i.e. if you were to alternately pick up a 1 vs. 2 kg object, you could easily say which was heavier. But if you were to alternately pick up a 20 vs. 21 kg object you might have great difficulty saying which is heavier.

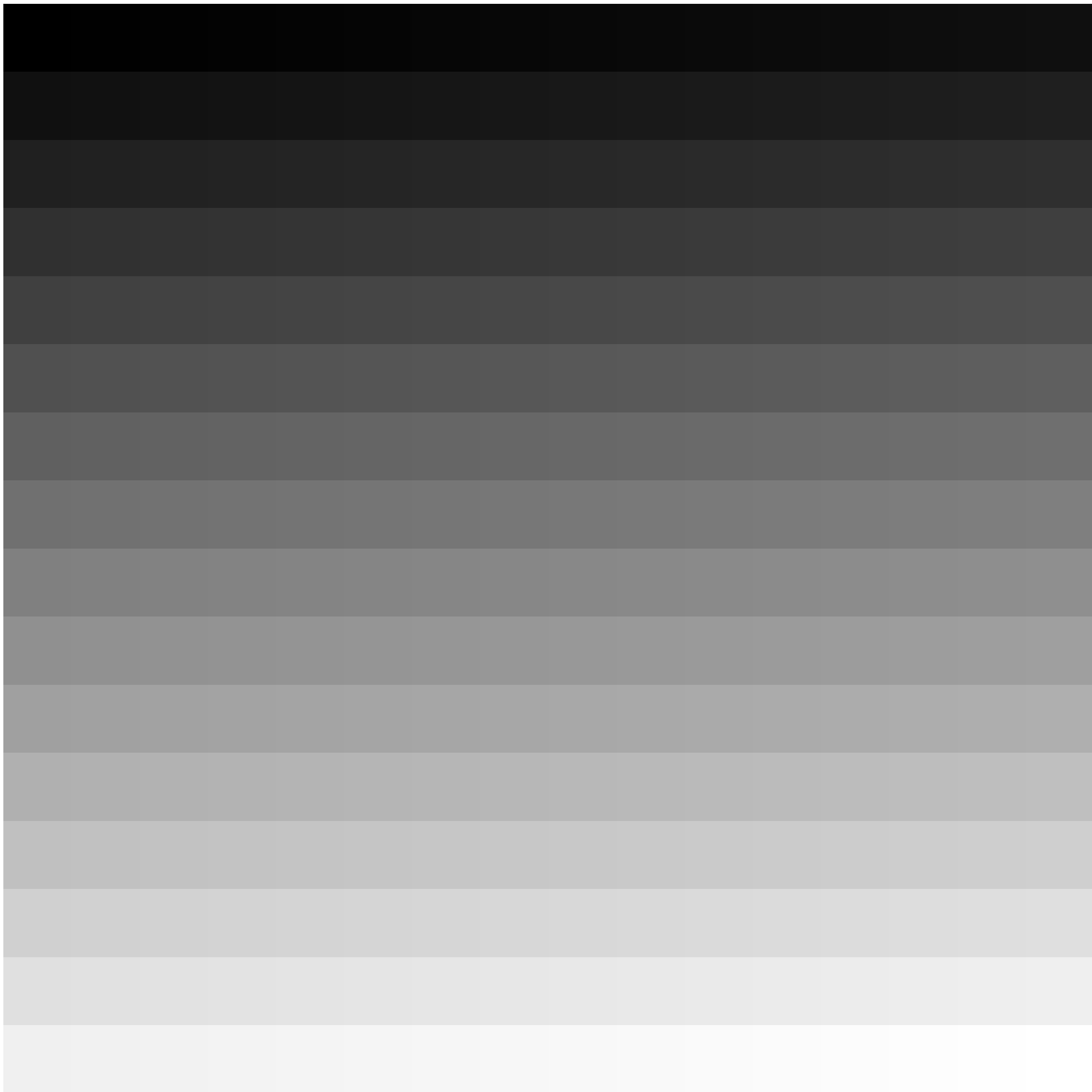
[ASIDE: one student pointed out to me that, while he agrees with the above, he noticed that he cannot discriminate something that weighs 1 g from something else that weighs 2 g. This is true, so the Weber Law I mentioned above holds over a range, bounded both at the bottom and top. (The same holds for brightness perception, of course.)]

Another example is in taste. You can discriminate a cup of tea that has one teaspoon of sugar from one that has 2 teaspoons of sugar, but you probably cannot discriminate 7 teaspoons from 8 teaspoons with the same ease (even though in both cases, the difference is 1 teaspoon).

Hearing scales are also non-linear. The loudness scale for example is logarithmic (decibels are units of log intensity of sound).

Bottom line: the non-linear "perceptual response function" for brightnesses of the various parts of an image is a very large effect. Next lecture, I will discuss one important way in which this affects how images are displayed.

[42]Technically, this difference is the amount required for you to notice that two intensities are different if you are forced to say "same" or "different" and asked to performed the experiment repeatedly. Typically one chooses a correctness percentage – say 75 percent – and says that the JND is ΔI at intensity I meaning that 75 percent of the time you will say "yes, they are different".



As an example, consider a set of pieces of paper ranging from black to dark grey to light grey to white. If you ask people to choose a sequence of say 10 of these papers that define a perceptually uniform scale, you will find that the physical reflectance will increase roughly as follows.[43]

value	description	physical reflectance
1	black	.01
2		.03
3	dark grey	.06
4		.12
5	middle grey	.2
6		.3
7	light grey	.43
8		.59
9	white	.78

Typical printed black inks have a reflectance between 1-3 per cent and whites are typically less than 80 percent. So a typical dynamic range of a printed sheet of paper is about 30:1. and how we should choose the 256 levels of intensities (8 bits for each rgb) that can be displayed on a monitor so that we can make the best use of the available range. Our main criteria here is not *necessarily* to have uniform increments of intensities but rather that the increments are determined by intensity differences that people can notice.

Weber's Law

People are clearly perceiving light intensities on a non-uniform scale. What exactly is this non-uniformity ? The Munsell scale roughly obeys a power law[44] with exponent 1/3, i.e.

$$value = reflectance^{\frac{1}{3}}.$$

Alternatively it can be fit using a log curve.

Let's briefly consider the latter, which is a case of Weber's Law. Weber's Law says that perceived intensity of some stimulus is a logarithmic function of the physical intensity of the stimulus. For example, in the case of pieces of paper

$$(perceived)\ lightness\ value\ \approx\ \log(reflectance).$$

One way to understand the log scale is to note that it implies the perceived relationship between two intensities depends on their ratio, that is, $\log I_2 - \log I_1 = \log \frac{I_2}{I_1}$, i.e. mathematically, the difference in the log of the physical intensity values depends the ratio of the physical values.

So, let's return to the issue of dynamic range of real or rendered scenes versus the dynamic range of displays. Real scenes have a dynamic range that can easily be 10,000 to 1, whereas monitors have a dynamic range of about (say) 200:1 and printed paper has a dynamic range less than that. Our problem is how to choose represent high dynamic range images on a low dynamic range displays.

[43] The numbers below correspond to the Munsell scale, which is a well known standard

[44] See Steven's Law. http://en.wikipedia.org/wiki/Stevens'_power_law

Aside: This problem is not a new one. For hundreds of years, painters have had to deal with this problem. Certain painters such as Caravaggio and Rembrandt, for example, are known for their ability to perceptually stretch the perceived dynamic range available to them. Photographers also have to deal with this problem, and have employed many darkroom tricks such as dodging and burning to increase the dynamic range of their prints. Ansel Adams is one photographer who is known for his techniques for solving this problem. Film makers also face the problem, and often introduce artificial lighting in order that the visible parts of the scene fall in a range that can be captured by the camera's dynamic range.

The problem that graphics researchers face is related to the above problems, but not identical. In graphics, one *has* a high dynamic range image and wishes to map it to a low dynamic range display device. The problem of remapping RGB values in an image so as to make the details more visible (or generally look better) is known as *tone mapping*. Those of you who work with simple digital photo software such as Adobe's Lightroom^{TM} are familiar with this.

Consider a 4 megapixel (MP) camera (2400×1800) whose sensor array is about 24mm $\times$ 16mm, so that in any row or column of the pixel grid we have a density of about 100 pixels per mm.

Let Δ be the width of the blur on the sensor plane. (In the above figure, the blur disk is a dark line on the sensor plane. For a 2D sensor plane, it is a disk.) Then, by similar triangles,

$$\frac{A}{z_i} = \frac{\Delta}{\mid z_i - f \mid}$$

and so

$$\Delta = \frac{Af}{z_o}$$

which is measured in mm. To convert to pixels, we multiply by 100,

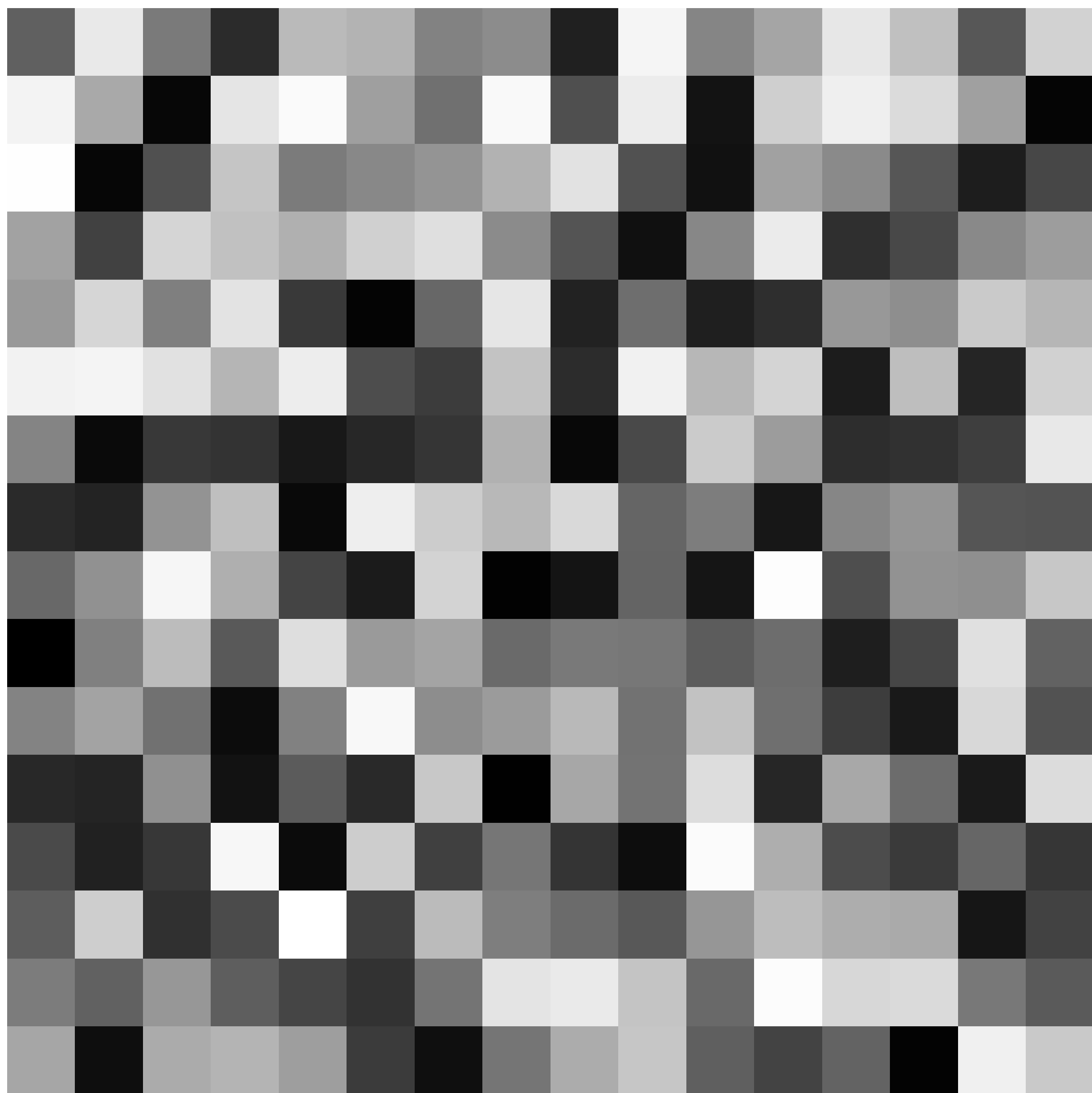
$$\Delta_{pixels} = \frac{100Af}{z_o}$$

Let's plug in some numbers. Suppose we use an $f = 80$ mm lens and we let the f-stop be 8.0. Then, $A = 80/8.0 = 10mm$, and so $\Delta_{pixels} = \frac{100*10*80}{z_o}$ where z_o is in mm, or $\Delta_{pixels} \approx \frac{1000}{z_o}$ where z_o is in m. Thus, an object that is 1000 m away has a blur of 1 pixel, an object that is 10 m away has a blur width of 100 pixels. This is quite significant!

How far would we need to move the sensor back from the lens in order to bring the surface at $z_0 = 10$ m into focus? From the thin lens equation, we want to solve for z_i where $\frac{1}{f} = \frac{1}{z_0} + \frac{1}{z_i}$ and $f = .08$ and $z_o = 10$, and so $z_i = 80.6mm$. Thus, we would only need to move the sensor back by about 0.6 mm to bring the object at distance 10 m into focus.

Let's look at a much more extreme case and consider an object at a distance $z_0 = 1m$. Solving we get $z_i = 87mm$, and so we would need to move the sensor plane $7mm$ (from 80 to 87 mm from the lens) to bring the object into focus.

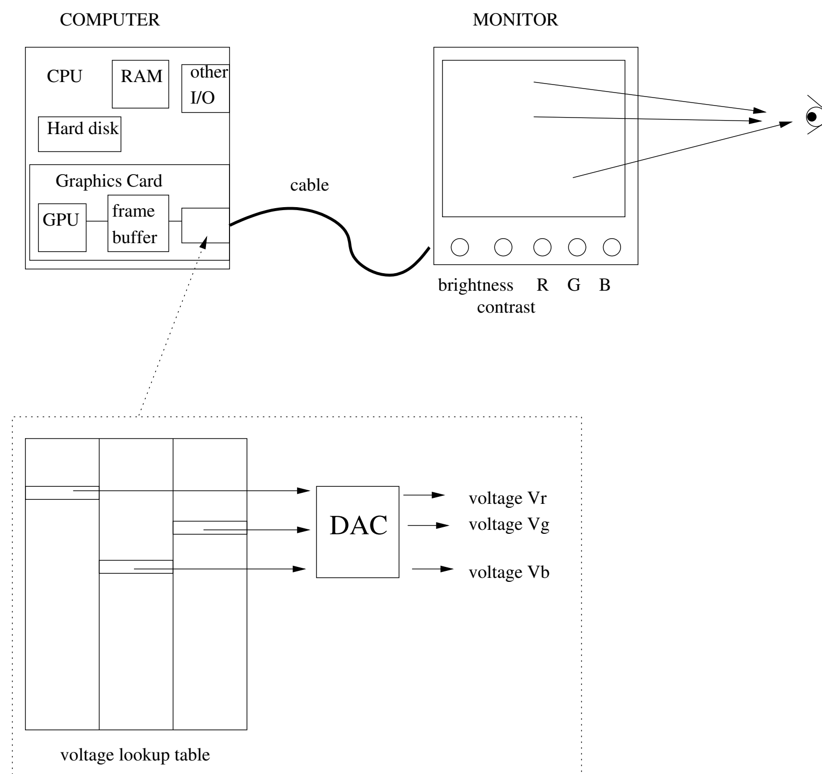
This example suggests that only very small translations of the lens relative to the sensor plane are required to change the focus of objects to what is desired. Thus, as long as we are focussed at a sufficiently large distance from the camera, we can approximate f as the distance to the focal plane.



Displays

Today we are going to consider the basics of how images are displayed. The typical display is a monitor, but the same discussion applies to projection displays.

Consider the sketch below. On the left is shown the computer, which consists (among other things!) of the CPU/RAM/disk/etc and the graphics card. The graphics card contains a processing unit called the GPU (graphics processing unit) which is devoted to performing operations that are specific to graphics. The graphics card also contains the frame buffer, the depth buffer, texture buffers.



The graphics card is connected to the monitor via a cable. (In the case of a laptop, you would not see the cable.) The signal that is sent to the monitor along the cable specifies RGB levels for each monitor pixel, but these are typically *NOT* the same as the intensity values that are written in the frame buffer. Rather, the cable typically carries the RGB intensities as an analog signal (voltage level) rather than a digital signal (bits). e.g. Those of you who have plugged your laptop into a projector may know that you typically use a VGA cable (VGA is analog). CRT displays expect analog inputs. LCD can expect either analog or digital.

Where does the analog signal specified? Each RGB triplet in the frame buffer is used to index into a lookup table (a readable/writable memory) which sits on the graphics card. The table contains a triplet of RGB *voltage* values V_{rgb} which typically have precision greater than the intensities in the frame buffer. e.g. the voltages values in the LUT might have 10 or 12 bit precision, rather than 8 bits. The voltage values are converted to an analog signal (see DAC in the figure, which stands for "digital to analog converter") and the analog signals are sent to the monitor via the cable

mentioned above. For simplicity, let's say that the RGB values and voltage values are in the range 0 to 1.

We will return to the question of how the voltages are chosen later. For now, let's go to the monitor and see what happens when the voltage value arrives there.

Brightness, contrast, and gamma

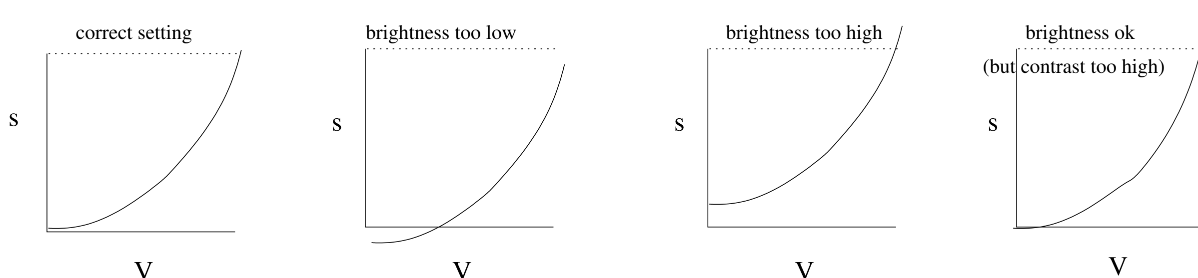
In the lecture on color, we considered the spectra of light that is *emitted* from a monitor (or projector). In particular, we examined the RGB spectra of the light emitted from the display and wrote these spectra as three columns of a display matrix $\mathbf{D}$. We used $\mathbf{s}_{rgb}$ to scale these spectra, namely by considering the spectrum $I(\lambda) = \mathbf{D}\mathbf{s}_{rgb}$ leaving the display at a pixel.

The voltages V_{rgb} that arrive on the cable to the monitor are not the same physical thing as the values $\mathbf{s}$ that weight the physical spectrum of light emitted from the monitor. Rather, the $\mathbf{s}$ values at each pixel *depend on* the voltage values V. The $\mathbf{s}$ values also depend on the "brightness","contrast" and "color" settings on the monitor which the user can adjust. (These adjustments help control the electronics of the monitor itself – they do *not* affect the voltage signal that arrives via cable to the monitor.)

Most monitors are designed so that, if the brightness and contrast are set properly, then the $\mathbf{s}$ values should depend on the voltage V values via a power law:

$$\mathbf{s}_{rgb} = V_{rgb}^{\gamma}$$

where $\gamma \approx 2.5$ (see sketch on left). Crudely speaking, the brightness setting controls the height of the curve at $V = 0$. If brightness is too low, then the curve shifts downward – in practice, you cannot have negative $\mathbf{s}$ values and so there is a cutoff whereby a range of values near $V = 0$ all give $\mathbf{s} = 0$ (see second fig below). If brightness is set too high, then the curve is shifted up and there are no voltages V that give the minimum emitted intensity that can be displayed by the monitor (see third figure).



The contrast should be set as large as possible to span (but not exceed) the range of what the monitor can display. If contrast is set too high, then the highest voltages (V near 1) will all map to the same (max) intensity of the display which is also not what we want (see figure on right). The color settings are similar to the contrast settings. They can raise or lower the maximum values, but they do so only by operating on one of the RGB channels of the monitor. Assuming the basic gamma model (with same gamma for RGB), we can treat the color settings as if we are multiplying by a constant κ_{rgb}.

In the end, the spectra that comes off a pixel on the monitor, if the monitor's brightness and constrast is set correctly, should be roughly:

$$I(\lambda) = \mathbf{Ds} \; = \; \mathbf{D} \begin{bmatrix} \kappa_r V_r^\gamma \\ \kappa_g V_g^\gamma \\ \kappa_b V_b^\gamma \end{bmatrix}$$

Gamma correction

I mentioned earlier the voltage lookup table which converts the I_{rgb} values in the frame buffer into voltages V_{rgb}. If we were to let the lookup tables contain the identity mapping i.e. $I = V$, then the scaling factors $\mathbf{s}$ for light spectra intensities emitting from the monitor would be non-linearly related to the rendered values I_{rgb} in the frame buffer, because of the gamma built into the monitor.

The standard way to correct the monitor gamma is to let the voltage table perform the mapping

$$V_{rgb} = (I_{rgb})^{\frac{1}{\gamma}}.$$

i.e.

$$(V_{rgb})^\gamma = ((I_{rgb})^{\frac{1}{\gamma}})^\gamma = I_{rgb}$$

This is known as *gamma correction*. The result is that

$$I(\lambda) \; = \; \mathbf{D} \begin{bmatrix} \kappa_r V_r^\gamma \\ \kappa_g V_g^\gamma \\ \kappa_b V_b^\gamma \end{bmatrix} = \mathbf{D} \begin{bmatrix} \kappa_r I_r \\ \kappa_g I_g \\ \kappa_b I_b \end{bmatrix}$$

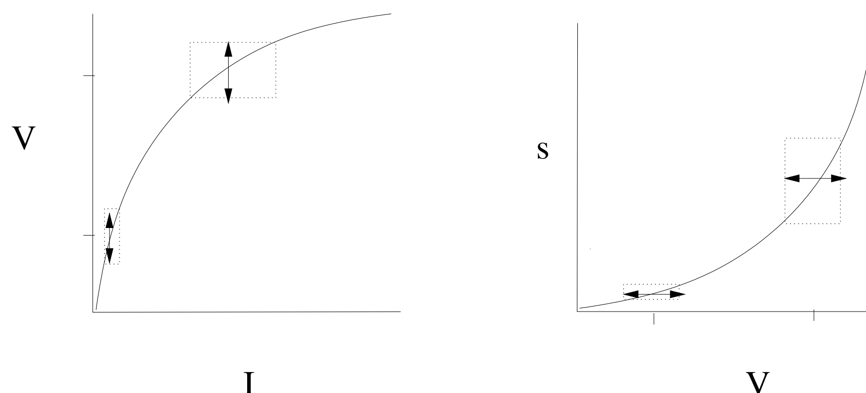
Gamma correct causes the spectrum $I(\lambda)$ that is displayed to be linearly related to the original I_{rgb} values.

Note that the above discussion did not address *why* monitors have a gamma behavior $\mathbf{s} = V^\gamma$, in the first place. Let's now turn to this question. The voltage signal that is carried on the cable is an analog (rather than digital) signal and is susceptible to small amounts of noise. So, although the voltage levels are specified with relatively high precision (say 10 bits), the *accuracy* will be less than this and the monitor will receive $V + n$ where n is some small random number (with mean 0).

Last class, I discussed how the visual system is more sensitive to changes ΔI in light intensity at lower intensities I (i.e. Weber's Law). Thus, if the monitor scaling factors s were proportional to V any given amount of noise n that is added to the voltage signal would be relatively more noticeable at low intensities than at high intensities.

In order to counter this perceptual effect, one effectively needs to encode the low light levels more accurately. To do so, one encodes the image intensities as voltages using a compressive non-linearity. When uniform noise is added to the voltages on the cable, this noise perturbs the low intensities by a small amount and perturbs the large intensities by a relatively large amount. (The perturbations shown in the figure below are magnified to exaggerate the effect.) Such a non-uniform perturbation is appropriate for reducing the effect of noise on human visual perception, since human vision is more sensitive to perturbations of intensities at low intensities. The monitor then inverts the non-linearity so that the resulting signal d_{rgb} is proportional to the original value of I_{rgb}.

Many video cameras *record* intensities using this gamma compression scheme. For example, the NTSC video standard which has been used in broadcast television for decades assumes that the



video is to be displayed on a monitor that has a gamma of about 2.5 and so the signals that are stored on tape have been already compressed relative to the intensities in the actual physical image using a $1/\gamma = 1/2.5 = .4$ compressive non-linearity power law. (Broadcast television has always worked this way, i.e. the NTSC signal is sent via electromagnetic waves through the air and picked up by an antenna.)

Thus, the main reason that monitors have this V^γ behavior and that we perform a non-linear transformation prior to sending the voltage signal to the monitor to protect the voltage signal against noise in the transmission and this non-linear transformation needs to be inverted.

Does OpenGL apply an inverse gamma encoding prior to writing images in the frame buffer? No, it doesn't. Graphics systems typically ignore gamma at the rendering stage and instead use the voltage lookup table to handle the gamma correction. Note: if you build in the gamma compression into the graphics system, you need to be careful since many operations such as interpolation of image intensities (Gouraud shading) and compositing require linear operations on the RGB values. This would be awkward to do if the values had been gamma corrected.

Summary

Let's briefly summarize the sequence of transformations by which rendered image intensities I_{rgb} in the frame buffer are presented on a display, and then observed by a human.

$$I_{rgb} \rightarrow V_{rgb} \rightarrow V_{rgb} + n \rightarrow \mathbf{s}_{rgb} \rightarrow I(\lambda) \rightarrow I_{LMS}$$

The transformations are determined, in order, by the LUT, the cable, the brightness/contrast/gamma of the monitor, the light emitter(s) in the monitor, and the spectral sensitivities of the photoreceptors.

ASIDE

I finished the lecture with an informal discussion of various other display systems, in particular, what happens when you use a projector to display an image on a screen. I also discussed an example of an interesting application which has arisen recently, in which people are attempting to display images onto non-uniform surfaces. For more information on *many* such applications, see the Spatial Augmented Reality website: `www.uni-weimar.de/medien/ar/SpatialAR/`.

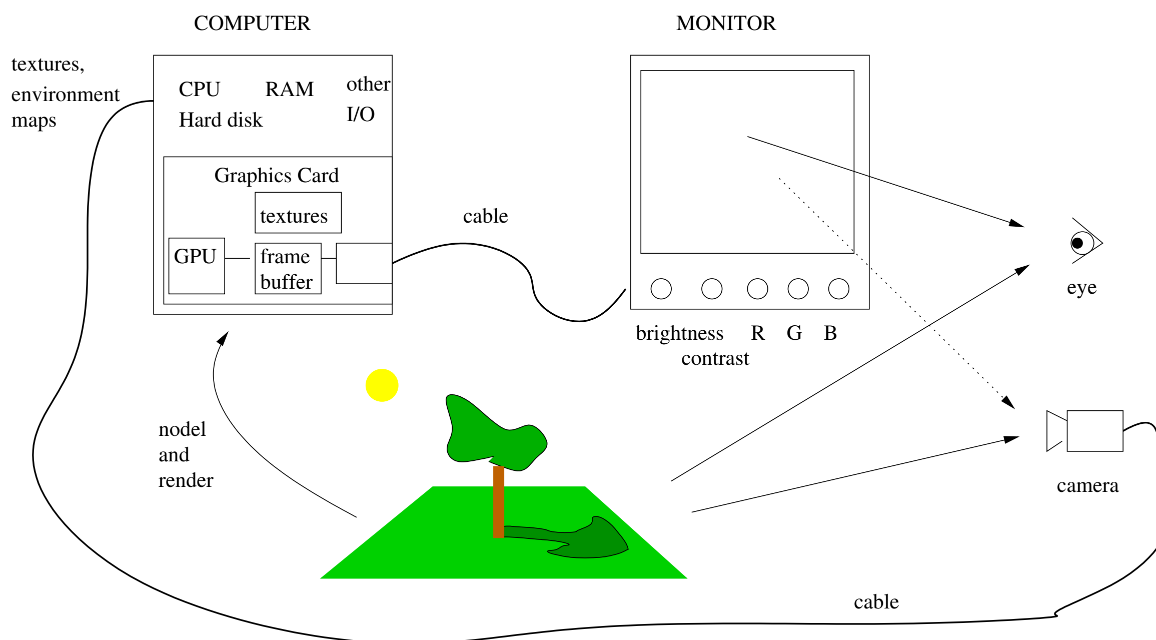


Figure 1: **maybe add a discussion of projector system if there is time**

Dynamic range and tone mapping

A typical display has a range of intensity values (called *dynamic range*) that it can produce.[45] For example, a typical CRT has a dynamic range of 200:1, meaning that is the ratio of maximum to minimum intensity of light emitted. The minimum intensity is always greater than zero. For example, displays always have some non-zero emitted or reflected light intensity. This can be due to noise in the voltage signal, noise within the monitor. (There is also light reflected off the monitor.) So in practice the dynamic range is finite.

The dynamic range of the display is often much less than the dynamic range of the image that you wish to display. What to do? The limited dynamic range is a concern, since the dynamic range of the rendered image (prior to quantizing to 255 values) is typically much greater than the dynamic range that can be produced by the display. For example, suppose you render a scene that includes a black painted surface in shadow and a white painted surface that is directly illuminated. The white surface in this scene might have rendered rgb values that are many thousands of times greater than the values of the black paper in shadow. We cannot re-produce this intensity range on the CRT. You also find very high dynamic ranges when there is a light source in the scene such as a lamp, or when there are specular reflections.

To keep it simple, let's assume that we are gamma correcting the monitor using the voltage lookup table. We this correction done, we now concern ourselves now with the mapping from the rendered image intensities I_{rgb} to the d_{rgb} values that determine the intensities on the display (via $\mathbf{D}$ d, where $\mathbf{D}$ is a 3 column matrix with the spectra of the display elements e.g. phosphors).

Because the rendered scene may have a dynamic range that is much greater than the dynamic range of the display, we need to somehow compress the intensities in the scene so that they fit on

[45] Also called the *contrast ratio*.

the display. The obvious way to do this, which turns out to be terrible, is to scale each *rgb* channel linearly. Let's ignore color for now. Linearly rescaling would give us the relationship:

$$\frac{d}{d_{max}} = \frac{I}{I_{max}}$$

where d_{max} and I_{max} are the maximum display intensities and rendered image intensities respectively.

This reason this works very poorly is similar to what I discussed with just-noticeable-differences, namely that people's perception of light intensity doesn't not obey a linear scale. If you ask people to choose *uniform* perceptual step sizes for an intensity variable, the intensity steps they will chose will be physically quite *non-uniform.* As with JNDs, the steps will be smaller at smaller intensity levels.

[ASIDE: THE MATERIAL BELOW IS NOT ON THE FINAL EXAM.]

Let's continue to ignore color for the moment. Let's suppose that the dynamic range of a rendered scene is $I_{max} : I_{min}$, whereas the dynamic range of the display is $d_{max} : d_{min}$. Rather than naively remapping the I intensities to d intensities as suggested above, we could instead remap such that intensity *ratios* are preserved. i.e. Weber's Law says that constant ratios of intensities should be perceived as perceptually uniformly space intensities. So let's try to preserve ratios.

Let's say we need to represent 256 different intensity levels (i.e. 8 bits). To preserve ratios, we would like the ith intensity level $I(i)$ to satisfy

$$\frac{I(i+1)}{I(i)} = r \ .$$

What is the ratio r ? It is easy to see that $I(i) = I_{min} \ r^i$ and $I_{max} = I_{min} \ r^{255}$ and thus

$$r = (\frac{I_{max}}{I_{min}})^{\frac{1}{255}}.$$

For example, if the dynamic range is 10,000:1, then $r \approx 1.04$.

Similarly, to preserve ratios in the displayed intensities, we should have

$$\rho = (\frac{d_{max}}{d_{min}})^{\frac{1}{255}}$$

and

$$d(i) = d_{min} \rho^i.$$

If the dynamic range is 100:1, then $\rho \approx 1.02$.

I will leave it as an exercise to show how to map I to d in this case.

Weber's Law arises often in perception. For example, musical scales (ABCDEFGA..) are defined by a logarithmic scale, rather than linear scale. The notes ABCDEFGA cover *one octave* which is a factor of two in sound frequency, so two octaves (ABCDEFGABCDEFGA) represents a factor of four in sound frequency, etc. Within an octave, the notes are related by a constant ratio of frequencies, rather than a constant different in frequency. The reason this is so for music is presumably that the human ear's sensitivity to frequencies is determined by ratios of frequencies rather than absolute differences of frequencies.

Weber's Law also arises in perceiving the loudness of a sound. Sound loudness is measured in decibels, which is a log scale.

Psychologists have carried out thousands of studies to determine under what conditions a perceptual scales either obey a power law (Stevens) or a Weber Law (logarithmic), and the details depend on exactly how you carry out the experiments. These details don't concern us here. The key for us is to understand that the non-linearity is compressive, and to realize that we can use at least a rough approximation of these perceptual phenomena to motivate tricks for displaying images.