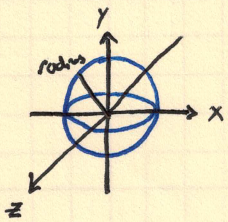
Outcomes

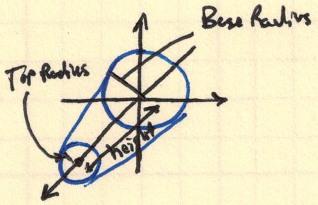
- Understand the basic concepts of GL Quadrics
- Understand the basic concepts and use of Bézier curves
- Understand the basics of loading objects defined in external programs
- Understand the basics of color

Glu Quadric Objects

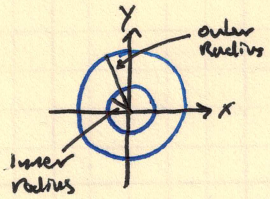
- Quadric objects provided by the OpenGL Utility Library (GLU)
  - Sphere
  - Tapered Cylinder
  - Annular Disc
  - Partial Annular Disk



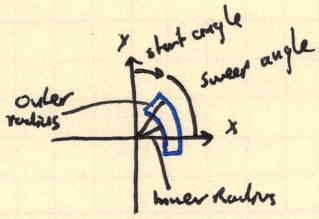
Sphere



Tapered Cylinder



Annular Disc



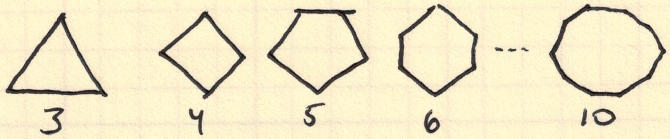
Partial Annular Disc

- gluSphere (*qobj, radius, slices, stacks) – creates a sphere of radius centered at the origin. The parameters slices and stacks determine the fineness of the quadrilateralization
- gluCylinder (*qobj, baseRadius, topRadius, height, slices, stacks) – Draws a tapered cylinder with its axis along the z-axis, whose base circle is of radius baseRadius lying on z=0 plane, and whose top circle is of radius topRadius lying on z= height plane.
- gluDisk (*qobj, innerRadius, outerRadius, slices, rings)
- gluPartialDisk (*qobj, innerRadius, outerRadius, slices, rings, startAngle, sweepAngle)
- GLU calls of the form gluQuadric*(*qobj, *) determine various properties of the quadric. I.e.,

  gluQuadricDrawStyle (qobj, GL_LINE)

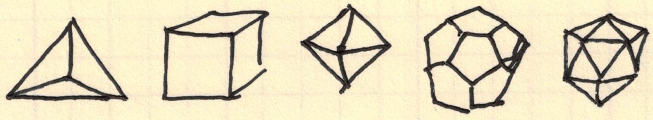
Regular Polyhedra

- A **regular polygon** is a simple planar polygon whose sides are of equal length and which has equal interior angles at its vertices



  - A regular polygon of n sides is convex and its vertices are spaced evenly along a circle at an angle of $2\pi/n$

- Regular polyhedra are a generalization of regular polygons to 3 dimensions



- A **polyhedron** is a solid object whose boundary is a polygonal mesh
- A **regular polyhedron** is a polyhedron all of whose faces are identical regular polygons
- Regular polyhedra are also called **Platonic solids**
- Polyhedra can be easily drawn using the freeGLUT library

Bézier Phrase Book

- Bézier and NURBS curves – special classes of curves and surfaces used in 3D design.
- They allow the programmer the capability to sculpt a primitive through the manipulation of control points, rather than by devising equations

• Curves

- A Bézier curve C is specified by a sequence $P_0, P_1, \ldots, P_n$ of control points in $\mathbb{R}^3$ whose number $n+1$ is called the order of C.
- The curve starts at P_0, ends at P_n, and approaches, but does not necessarily pass through the intermediate ones.
- Intermediate control points can be thought of as "attractors"
- Specifying the control points defines a Bézier curve C in a particular parametric form:

$$x = f(t),\ y = g(t),\ z = h(t),\ \text{where } t \in [t_1, t_2]$$

- The statements:

~~glEnable(GL_MAP1_VERTEX_3)~~

glMap1f(target, t1, t2, stride, order, *control points)

- defines a one-dimensional Bézier evaluator
  - target ≡ information to generate, i.e., GL_MAP1_VERTEX_3
  - t1 and t2 ≡ specifies the endpoints of the parameter interval of the curve
  - order ≡ specifies the number of control points
  - stride ≡ number of floating point values between start of one control point and the next
  - control points ≡ The array of control points

glEnable(GL_MAP1_VERTEX_3)

← should be same as target above

- Enables the curve

glEvalCoord1f((GLfloat) i/50.0)

→ parameter interval

- Used to evaluate and draw the curve

- Additionally, we can draw a curve using an evenly spaced grid of control points specified by:

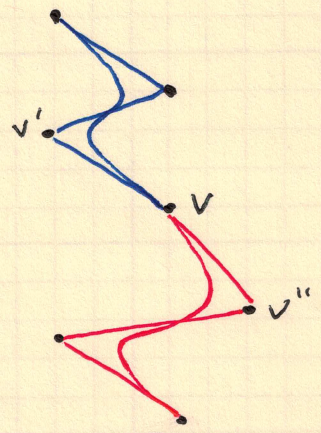
```
glMapGrid1f(n, t1, t2)
```

  - which specifies $n+1$ sample points in the parameter interval starting at $t1$ and ending at $t2$
  - Example:

```
glMapGrid1f(50, 0.0, 1.0) - 0.0, 0.02, 0.04, ..., 1.0
```

- The call `glEvalMesh1(mode, p1, p2)` works in tandem w/ `glMapGrid1f(...)`
- If mode is `GL_LINE`, then a line strip is mapped through the points

- End Tangents
  - The curve contains the first and last control points
  - The tangent at the first control point is along a straight line joining with the second control point.
  - The tangent of the last control point lies along the last control polygon segment.
  - Thus we can join two Bézier curves, which meet at a common end control point V



- Surfaces
  - A Bézier surface (or patch) s is specified by an $(n+1) \times (m+1)$ array of control points

$$P_{ij},\ 0 \le i \le n,\ 0 \le j \le m$$

  - The surface passes through the four "corner" control points $P_{00}, P_{n0}, P_{0m}, P_{nm}$ but not necessarily the others which act as attractors.
  - The control point array of a Bézier surface s defines the form

$$x = f(u,v),\ y = g(u,v),\ z = h(u,v) \text{ where } (u,v) \in [u_1, u_2] \times [v_1, v_2]$$

  - Specified using a 2D Bézier Evaluator

```
glMap2f(target, u1, u2, ustride, uorder, v1, v2, vstride, vorder, * controlPoints)
```

  - Enabled with: `glEnable(GL_MAP2_VERTEX_3)`

- Similarly to the curve, we can specify an (numberU+1) x (numberV+1) grid of sample points to define the surface with

```
glMapGrid2f(numberU, u1, u2, numberV, v1, v2)
```

and an evaluator

```
glEvalMesh2(mode, p1, p2, q1, q2)
```

Importing Objects

- Once you have a solid grasp of modelling by hand in OpenGL you may find it easier to use tools such as 3D Studio Max, Maya, or Blender to create objects.
- These tools can export these models to files which can then be imported into your OpenGL context
- One of the most common file formats is the OBJ format
  - developed by Wavefront Technologies for their Advanced Visualizer animation package.
- OBJ File format
  - A sequence of lines, where the starting characters of each line specify what the line is for:
    - v - specifies the (x,y,z) coordinates of a vertex
    - vt - specifies the s and t texture coordinates of a vertex
    - vn - specifies the (x,y,z) coordinates of a vertex normal
    - f - specifies a polygonal face as a sequence of 3 or more vertices, each vertex itself specified by a sequence of numbers separated by slashes, successive vertices are separated by spaces
  - We can simply read the file line by line and process the object as it is defined using either GL_TRIANGLE_FAN or GL_TRIANGLE_STRIP
  - A better approach would be using the Open Asset Import Library which is an open source library to import various 3D model formats and which works well with OpenGL in C++

    www.assimp.org

Color and Light

• Basics of Color

- Light is visible to the human eye in the approximate range 400-700 nm, which ranges from red to violet
- Below this range is infrared, above violet there is ultraviolet, both of which are invisible to the naked eye.
- The color spectrum is considered to have seven major bands

Red
Orange
Yellow
Green
Blue
Indigo
Violet

- Color intensities are usually either given as floats from 0 → 1, or as integers in one of the ranges 0 → 255, 0 → 32767, or 0 → 65535
- Conversions between these ranges are linear and so trivial

• Additive vs. Subtractive Color Systems (RGB and CMY)

- There are two basic categories of output devices for images:
  - Monitors
  - Printers
- This means that we have to deal with two sets of primary colors, which are called the additive primaries and subtractive primaries

- RGB

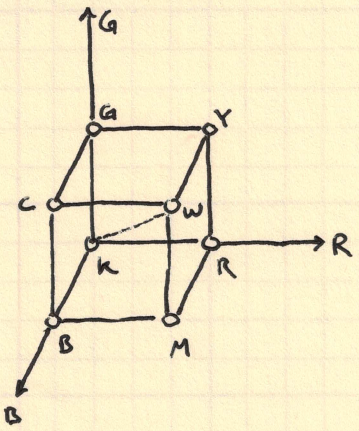
- Additive Primaries occur on monitors
- Typically these will have small triples consisting of red, green, and blue spots, which can be excited with a range of different intensities of an electron beam
- The background is black
- This system relies on the human visual system adding up the red, green, and blue colors to perceive the mixed color
- So, since the 3 colors are added by the viewer to create the color, RGB are the additive primaries

- CMY

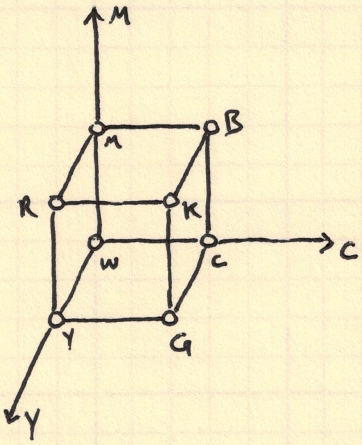
- Occur when we print an image
- We start with a piece of paper, and then add colored inks which filter out light at specific frequencies
- We can get the color we want by adding different amounts of Cyan, Magenta, and Yellow
- The background is white, and the 3 colors are mixed together to filter or subtract out light are called the subtractive color primaries

- RGB and CMY Color Cubes

- RGB Color Cube



- CMY Color Cube



R - Red
G - Green
B - Blue
C - Cyan
M - Magenta
Y - Yellow
W - White
K - Black

- The RGB cube, where axes show the intensities of red, green, and blue.
- Assume that the cube has sides with unit lengths, and that the intensity range of each color is $0 \rightarrow 1$. (i.e., all color mixtures of red, green, blue will lie inside the cube, e.x.:

  Yellow vertex is at RGB(1,1,0))

- A couple of things need emphasizing that we can find in the RGB cube:
  - (0,0,0) represents Black → K
  - (1,1,1) represents White → W
  - Full intensity of all three RGB values gives white, while if R=B=G, then the color lies on the main diagonal between black and white and the color will be an intensity of gray
  - All three subtractive primaries can be made by mixing two of the additive primaries at full intensity
    - C = (0,1,1)
    - M = (1,0,1)
    - Y = (1,1,0)
- The CMY cube shows similar properties
  - (0,0,0) represents white
  - (1,1,1) represents black
  - If C=M=Y then the color lies on the main diagonal from white to black and will be gray
  - All three additive primaries can be made by mixing two of the subtractive primaries at full intensity
    - R=(0,1,1)
    - G=(1,0,1)
    - B=(1,1,0)

- CMYK

  - The CMY color cube shows, full intensities of CMY should produce black, but unfortunately most inks and paints produce an ugly brown.
  - Another factor is that black ink is usually cheaper than the colored inks.
  - As a result most printers have black cartriges in addition to their three CMY cartiges
  - Consider the color CMY(0.5, 0.2, 0.7). It can be considered as the sum of the gray component (0.2, 0.2, 0.2) with C = 0.3 and Y = 0.5, as follows:

$$(0.5, 0.2, 0.7) = (0.2, 0.2, 0.2) + (0.3, 0.0, 0.5)$$

  - So instead of using the CMY colors at (0.5, 0.2, 0.7) we can use black at 0.2 intensity and CMY at (0.3, 0.0, 0.5)
  - So the technique when using CMYK to print a CMY specified color is to subtract the smallest CMY value from all three and assign to black.

- Converting RGB ↔ CMY ↔ CMYK

  - Given any of the three models we need to be able to convert to any of the others
  - The approach that will be taken is to do conversions of the form RGB↔CMY and CMY↔CMYK
  - RGB↔CMY

    - Based on either of the color cubes: Cyan (0,1,1) RGB, can be referred to as not red, and similarly for the other three
    - This is usually formalized in the two equivalent matrix equations:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} C \\ M \\ Y \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

    - So, conversions between RGB and CMY are trivial; you just subtract the value from (1,1,1) to get the new value

Ex: (0.2, 0.7, 0.3) = (0.8, 0.3, 0.7)

- CMY ↔ CMYK

  - To convert from CMYK to CMY we just add the K value to all three of the CMY values and we're done. (if one of the values is > 1, then the CMYK value wasn't a legal color. Most systems would just consider a value like that as saturated, and set it to 1)

  - To convert from CMY to CMYK: find the smallest value of these three CMY values, subtract it from all three, and set K to that value)

  - Ex: (0.8, 0.6, 0.7) CMY = (0.2, 0.0, 0.1, 0.6) CMYK

    (0.3, 0.4, 0.0) CMY = (0.3, 0.4, 0.0, 0.0) CMYK

- Exercises

Convert: (0.3, 0.2, 0.4) RGB to CMYK

Convert: (0.3, 0.2, 0.4, 0.3) CMYK to RGB

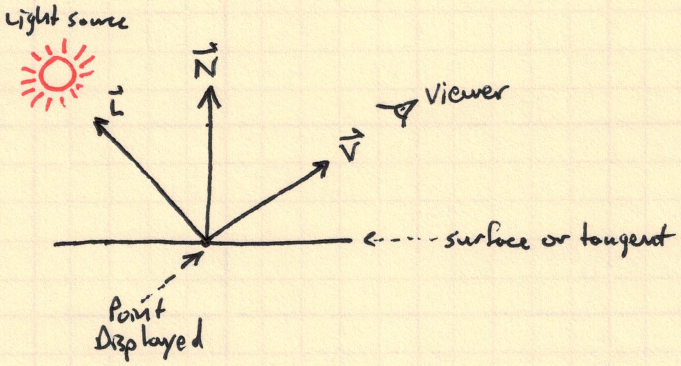
Phong Illumination

Overview of Illumination Methods

- There are two main components to displaying an illuminated object, an illumination model which determines how to average these colors across a surface so that it looks smooth, flat, bumpy, or whatever

- We'll assume that we have a viewer who is looking at the point on the surface from a known location, that we have a light source with known intensity, color, and location, that we can compute the normal to the surface of the object at the point, and that we know the color and shininess of the surface of the object.

- So the basic setup is as shown in the following figure:



$\vec{L}$ and $\vec{V}$ are unit vectors pointing towards the light source and viewer, respectively, and $\vec{N}$ is the unit normal vector. We can easily compute $\vec{L}$ and $\vec{V}$ by subtracting known points and we use standard techniques to find the the normal vector $\vec{N}$

- If the point is on a plane this is trivial, and if it is a point on a curved surface then the normal is to the tangent plane at that point. We'll assume throughout that we have already performed the calculations to determine that the point is visible to both the light source and the viewer

- The intensity of the light source will be named I_L. Initially we'll ignore the drop in intensity that occurs when you get farther from the light (this is called **attenuation**), and we will account for it later.

- For the most part we will assume that we are dealing with a single light wavelength so that we can develop the formulas without having to keep dealing with (R, G, B) triples. Then we will generalize the results to the RGB model.

- In these notes we'll be describing the Phong Illumination model. There are other models, but Phong dominates, and it is the model that is most supported by graphics systems.

- The primary alternative is the Torrance-Sparrow model, first described by Blinn, which is based on the physics of reflection. By comparison, the Phong model isn't based on much physical reality, but it works pretty well and is much less complicated than Torrance-Sparrow.

◦ Ambient, Diffuse, and Specular Components

- The Phong Illumination model has 3 components, ambient, diffuse, and specular. They give rise to three intensity (color) values: I_{AMB}, I_{Diff}, I_{spec}

- The total intensity at the point will be the sum of these components

$$I = I_{amb} + I_{diff} + I_{spec}$$

- Ambient Light: is background light that can't be attributed to any specific light source. Without it scenes look artificial because, for example, a side of an object that doesn't face a single light source will be completely invisible since it won't have diffuse or specular intensities.

- Diffuse Light: is scattered light from a light source. It is the same in all directions, so it doesn't depend on the viewer's location, but only on the position of the light source relative to the surface. If the light source is directly above the surface there will be more diffuse light than if the light is coming in at an angle to the surface.

- Specular Light: is the hot spot. If you take a shiny object like a waxed apple out into bright sunlight, then it will be colored by diffuse and ambient light. But there will also be a shiny hot spot which is around the reflection of the sun in the apple from your view point.

◦ Ambient Light

- The ambient component of the illumination model is the simplest. Since it just represents light that is everywhere in the scene, and isn't attributable to any light source, it is just a constant that is added into every computed intensity.

- A typical intensity is something like $I_a = 0.2$ or 0.3. Try a number like this and then adjust up or down until the image meets your needs.

- Different surfaces will reflect back different fractions of this light, depending on surface properties and color. We'll use a constant k_{amb} called the ambient reflectance coefficient, which satisfies $0 \le k_{amb} \le 1$, and represents the fraction reflected back to the current surface and wavelength.

- This constant basically represents the color of the surface, and so, for example, a yellow surface will have k_{amb} values that are high in the red and green frequencies and low in the blue frequency

- Then the ambient intensity is calculated as:

$$I_{amb} = k_{amb} \times I_a$$

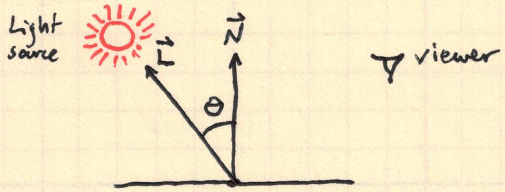
- Diffuse Light

  - This is scattered light, which only depends on the position of the light source relative to the plane we are displaying

  - If the light source is straight above the plane then we get the most scattered light

  - If it coming in at an angle that just grazes the plane then we get the least amount of scattered light.

  - So looking at the following figure, where θ is the angle between $\vec{L}$ and $\vec{N}$, what we need is a function that ranges from 0 to 1 and is at its maximum when $\theta = 0°$, and approaches zero as θ approaches 90°. Cosine does this for us, and so the diffuse model includes a multiplier of $\cos(\theta)$, which is $\vec{L} \cdot \vec{N}$



  - Now it is just a matter of filling in the details. The scattered light also depends on the intensity and color of the light source, I_l, and on the color of the surface, which will determine how much of the light at this frequency (color) will be reflected off the surface and not be absorbed.

  - This will be modeled by k_{diff}, the diffuse reflectance coefficient where $0 \leq k_{diff} \leq 1$

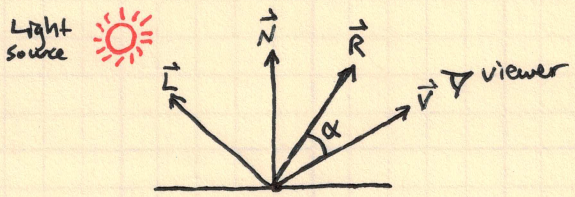
  - Putting all this together, the diffuse intensity will be defined by

$$I_{diff} = I_l \times k_{diff} \times (\vec{L} \cdot \vec{N})$$

  - Since k_{amb} and k_{diff} are dependent on surface color, they are usually set to the same value.

Specular Light

- Although the whole illumination model is usually named after Phong, he is primarily responsible for the specular component
- The basic idea is that in addition to light source intensity and color the hot spot is based on two things; the shininess of the object and how close the viewer is to looking down the pure reflection vector.
- A shiny object (e.g., obsidian or a waxed apple) will have a small hot spot, whereas a less shiny object (e.g., a keyboard) will have large hot spots.
- First, we need a bit more detail in our illumination ~~model~~ diagram. We've added a new unit vector, $\vec{R}$, which is a pure reflection vector. I.e., the angle between $\vec{L}$ and $\vec{N}$, which we called θ in the previous figure, is also the angle between $\vec{R}$ and $\vec{N}$, α is our name for the angle between $\vec{R}$ and $\vec{V}$
- For the rest of this discussion we'll assume that $|\alpha| < 90°$ (so $\vec{R} \cdot \vec{V} > 0$). If it isn't then the specular component is just set to 0



- As we discussed above, the hot spot should only exist if α is very small. It would be nice to just use the dot product again, and use $\cos(\alpha)$ to model this, but unfortunately cosine drops off relatively slowly.
- E.g., if $\vec{R}$ and $\vec{V}$ were 60° apart, we would still get half ($\cos(60°)$) of the intensity that we would get if the viewer were looking straight down the reflection vector.
- To avoid this slow drop off, Phong uses $\cos^n(\alpha)$ for large enough n, which gets the effect that we want.
- If α is small, and so its cosine is close to 1, then taking it to the nth power (for large n) will still give a value close to 1.
- If α were, say, 60°, and n were 20, then its cosine's nth power would be about 0.000001, which means there would be no specular effect.
- So the multiplier in Phong's specular model is $(\vec{R} \cdot \vec{V})^n$, which gives the cosine of α to the nth power. This leads to the specular intensity equation.

$$I_{spec} = I_l \times k_{spec} \times (\vec{R} \cdot \vec{V})^n$$

- Where $0 \leq k_{spec} \leq 1$ is the specular reflectance coefficient. With diffuse light the reflectance coefficient depended on the color of the surface. However if you look at a shiny object in a bright light you'll see that the color of the light source, not of the object.

- It also depends on θ, but that is usually ignored given a constant k_{spec}. So select a k_{spec} that mainly depends on the color of the light source, but also includes some component from the surface color.

- The main thing that you need to remember when using a Phong Illumination model system is that increasing the size of n will decrease the size of the hot spot and vice versa

- So shiny surfaces should have bigger n values, dull surfaces should have smaller n values. Try a number like 50, and then adjust it up or down until you are happy with the image.

- 50 might seem like a large power to use, but the shape of the cosine function is so flat near 0° that you need to use a larger power. To show why, I've put together a table of the powers of cosines for angles in the range that are relevant

	$\alpha = 5°$	$\alpha = 10°$	$\alpha = 15°$	$\alpha = 20°$
$\cos(\alpha)$	.9962	.9848	.9659	.9397
$\cos^{10}(\alpha)$	.9626	.8581	.7070	.5369
$\cos^{20}(\alpha)$	.9266	.7363	.4999	.2882
$\cos^{30}(\alpha)$	.8919	.6317	.3534	.1547
$\cos^{40}(\alpha)$	.8586	.5421	.2499	.0831
$\cos^{50}(\alpha)$	.8264	.4651	.1767	.0446
$\cos^{60}(\alpha)$	.7955	.3991	.1249	.0239
$\cos^{70}(\alpha)$	.7658	.3425	.0883	.0129

- So another approach to selecting a value for n, as compared to just trying 50 and adjusting, is to use this table. E.g., if you want a hot spot that is less than 10% of central intensity by the time you are 15° from the center, then you'll need to use an exponent close to 70

○ Attenuation

- As you get further and further from a light source, the intensity drops off by the square of the distance. This is called attenuation.

- If the light source is about the same distance to all the objects in the scene, the attenuation can often be ignored, but in many cases you must account for it.

- E.g., say that you are building a night view of a city, lit by street lights. You won't want the illumination from a street light in one part of the city to be just as bright as it is a few miles away in a different part of the city.

- Most graphics systems let you specify an attenuation function that is based on the distance, d, of the light source to the object with the formula

$$f_{att}(d) = (a + bd + cd^2)^{-1}$$

and then use $f_{att}(d) \times I_l$ as the intensity of the light source at a distance of d from its location.

- In practice it is usually sufficient to ignore the squared term (although this really offends physicists) by setting $c=0$

- Also, when $d=0$ it makes sense for $f_{att}(d)$ to be 1, so this implies that a should always be set equal to 1.

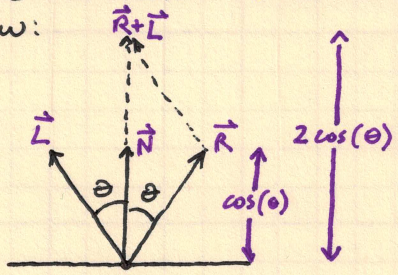
- Another factor to consider is a light source at infinity or great distance. In that case we want to ignore attenuation and so we just set $f_{att}(d)$ to 1

- Incorporating this into the overall model we get the illumination formula:

$$I = I_{amb} + f_{att}(d) I_l (k_{diff}(\vec{L}\cdot\vec{N}) + k_{spec}(\vec{R}\cdot\vec{V})^n)$$

Computing the Reflectance Vector

- To compute the unit vector $\vec{R}$ from our known quantities, $\vec{L}$ and $\vec{N}$, we need the figure below:



- Here we have added $\vec{L}$ to $\vec{R}$ which gives the vector $\vec{R}+\vec{L}$ extending up in the direction of $\vec{N}$. Looking at the bottom of the right triangle, its height is $\cos(\theta)$ since the length of $\vec{R}$ is 1 (unit vector), and so the vector $\vec{R}+\vec{L}$ has the same direction as the unit vector $\vec{N}$, with length $2\cos(\theta)$. I.e.,

$$\vec{R} + \vec{L} = 2\cos(\theta)\vec{N}$$

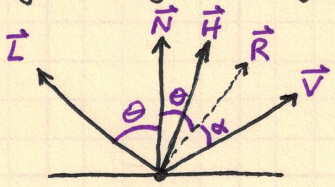
- Using the dot product $\vec{L}\cdot\vec{N}$ for $\cos(\theta)$, this gives us an equation for $\vec{R}$

$$\vec{R} = 2(\vec{L}\cdot\vec{N})\vec{N} - \vec{L}$$

○ Using the Half Angle Vector of the Reflection Vector

- If we have to run the Phong Illumination model on a lot of points (which is typical) then computing the reflection vectors as described in the last section is relatively expensive.

- So the common approach is to use the half angle vector, $\vec{H}$, which is the vector half way between $\vec{L}$ and $\vec{V}$ as shown in the figure below which, as we'll see, is a hack, but gives a good enough approximation of specular effects



- Phong's basic idea was that $\cos^n(\alpha)$ is a convenient value which is close to 1 for small α, and then rapidly drops off to 0 as α gets larger

- So, we should be able to use $\cos^m(\frac{\alpha}{2})$ where m is larger than n, since it is even closer to 1 for small α, and also rapidly drops off to 0 as α gets larger, to get just as good of an effect. That is what the half-angle approach does.

- Looking at the figure, the angle between $\vec{L}$ and $\vec{V}$ is $2\theta + \alpha$, and so the angle between $\vec{L}$ and $\vec{H}$ (which is half way between them) is $\theta + \alpha/2$

- Since the angle from $\vec{L}$ to $\vec{N}$ is θ, this leaves $\alpha/2$ as the angle between $\vec{N}$ and $\vec{H}$. So we can replace $(\vec{R}\cdot\vec{V})^n$ in the illumination model with $(\vec{N}\cdot\vec{H})^m$, without loss of quality. By convention we use n instead of m, and so the specular component becomes

$$I_{spec} = I_l \times k_{spec} \times (\vec{N}\cdot\vec{H})^n$$

- Of course this only makes sense if computing the unit vector $\vec{H}$ is faster than computing $\vec{R}$, but we can do this with:

$$\begin{aligned}\vec{H} &= (\vec{L}+\vec{V})/2, \text{ normalized} \\ &= \vec{L}+\vec{V}, \text{ normalized} \\ &= (\vec{L}+\vec{V})/|\vec{L}+\vec{V}|\end{aligned}$$

which is fast, and so this is the approach usually taken

○ Multiple Light Sources

- If a scene has multiple light sources, we just add their diffuse and specular effects. The ambient component will only be included once.

- So the illumination equation becomes:

$$I = k_{amb} I_a + \sum_{l=1}^{n} \{ f_{att}(d_l) I_l [k_{diff}(\vec{N}\cdot\vec{L}) + k_{spec,l}(\vec{N}\cdot\vec{H})^n] \}$$

- Note that the specular reflectance coefficient varies by light source it depends, at least partially, on the color of the source, whereas the diffuse reflectance coefficient and n are constant since they only depend on the surface color and properties.

- I'm assuming that the same attenuation function applies to all light sources, but the result will, of course, depend on d_l, the distance from light source l

o Multiple Frequencies (e.g., RGB)

- Until now we've simplified the equations by assuming that we have a single frequency, which will only apply in the rare case when we are doing gray scale images, while usually we want to be dealing with the RGB system.

- To change to RGB the light intensities and the k values will all become triples.

- The light source and the ambient light will be represented as $I_l = (I_{lR}, I_{lG}, I_{lB})$ and $I_a = (I_{aR}, I_{aG}, I_{aB})$, respectively.

- In most cases light sources and ambient light are white light, so the three components will usually be equal.

- The k values will also be triples:

$$k_{amb} = (k_{ambR}, k_{ambG}, k_{ambB})$$

$$k_{diff} = (k_{diffR}, k_{diffG}, k_{diffB})$$

$$k_{spec} = (k_{specR}, k_{specG}, k_{specB})$$

- As we discussed earlier, usually the k_{amb} and k_{diff} triples will be identical.

- Tying this together you can either consider the final illumination equation to be over a set of triples, or (and this probably easier) as three equations with one equation for the primary color.

Shading Techniques

- Summary of Shading Techniques

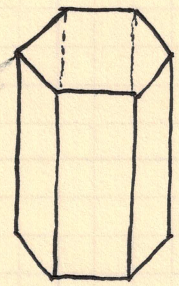
  - Two basic approaches for displaying a polygonal object:

    - Flat shading
    - Smooth shading

  - Example: say we want to display a can of soda, which is represented as a series of polygons; in the figure below we have eight polygons, the two on top and bottom and the six around the outside.

  - If we display the object using flat shading we'll see the object as it appears here with dramatic jaggies and not much resemblance to a cylinder.

  - If, however, we use smooth shading, then the outside will appear cylindrical, with no obvious vertical edges, and the top and bottom will appear flat.



  - There are three major techniques, one four flat shading and two for smooth shading.

  - The flat shading method, which is called Lambert shading, is by far the fastest, and so it is used not only for polygonal objects which should have flat surfaces, but also for smooth objects when we want to render them faster.

  - The two smooth shading methods are much slower than Lambert shading.

    - Phong shading produces better output than Gourand shading, but it takes much longer to render the images

    - Bui-Tuong Phong has been very productive, and his name is attached to both the Phong illumination model and Phong shading.

    - These are completely different systems and it is very possible that you will be using the Phong illumination model with any of the three shading models: Lambert, Gourand, or Phong

- Lambert (Flat) Shading

  - With Lambert shading each polygon is uniformly colored. Computationally it is very simple and very fast.

  - For each polygon in the scene we select one point use the Phong illumination model to compute its color, and then flood fill that color through the polygon.

  - In theory, it is best to use the center of the polygon for the point that is selected for coloring, but in practice many systems will just select one of the vertices since this isn't a very high quality rendering system and so computing the center of gravity of the vertices might not be worth the tiny amount of extra effort

- Smooth Shading and Vertex Normals

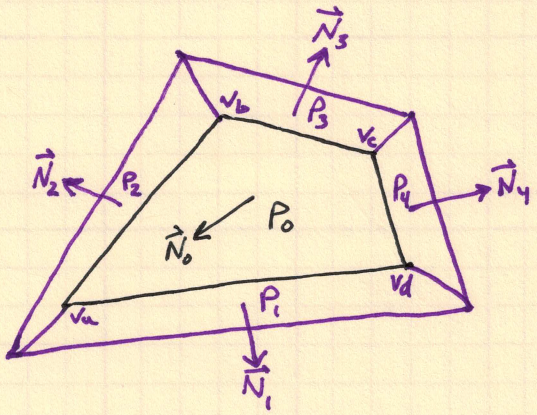
  - Both smooth shading methods are based on the concepts of vertex normals and on linear interpolation across the polygon

  - The difference is that Gourand shading only runs the illumination model to calculate intensities (i.e., colors) at the vertices, and then linearly interpolates these intensities across the polygon.

  - By comparison, Phong interpolates the normals across the polygon and then does an illumination calculation at every point in the polygon that will project to a pixel on the screen.

  - The first thing we need to define is a vertex normal. Say that we have the polygon P_0 (in black), with four vertices V_a, V_b, V_c, V_d, shown below, which has the four adjacent polygons P_1, P_2, P_3, P_4

  - Also assume that the unit polygon normal for each polygon P_i is $\vec{N}_i$, as shown.



  - Think of this figure with P_0 higher than the other polygons, which slope down the hill.

- If we use Lambert shading, then the polygonal structure will be displayed. Say, however, we want this to look like the smooth top of a hill, which we've represented with polygons. This is where we will use Gouraud or Phong shading

- In both cases we first define the unit vertex normals as being the average of the surrounding polygon normals that we want to appear to be smoothly connected In this case we will compute

$$\vec{N}_a = (\vec{N}_0 + \vec{N}_1 + \vec{N}_2) / |\vec{N}_0 + \vec{N}_1 + \vec{N}_2|$$

$$\vec{N}_b = (\vec{N}_0 + \vec{N}_2 + \vec{N}_3) / |\vec{N}_0 + \vec{N}_2 + \vec{N}_3|$$

$$\vec{N}_c = (\vec{N}_0 + \vec{N}_3 + \vec{N}_4) / |\vec{N}_0 + \vec{N}_3 + \vec{N}_4|$$

$$\vec{N}_d = (\vec{N}_0 + \vec{N}_1 + \vec{N}_4) / |\vec{N}_0 + \vec{N}_1 + \vec{N}_4|$$

- Alternatively, say you wanted a smooth saddle across polygons P_4, P_0, and P_2, with polygons P_1 and P_3 dropping off with an abrupt edge. Now we'll change the vertex normals for P_0 so that they no longer average in P_1 and P_3. I.e., we'll use the equations:

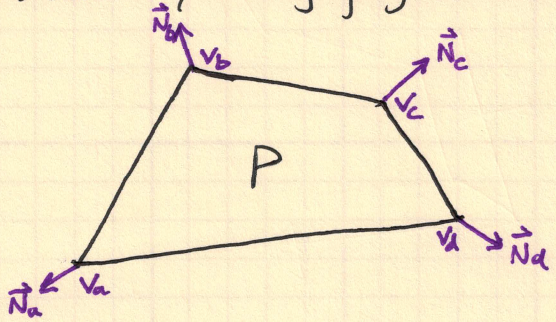
$$\vec{N}_a = (\vec{N}_0 + \vec{N}_2) / |\vec{N}_0 + \vec{N}_2| \qquad \vec{N}_b = (\vec{N}_0 + \vec{N}_2) / |\vec{N}_0 + \vec{N}_2|$$

$$\vec{N}_c = (\vec{N}_0 + \vec{N}_4) / |\vec{N}_0 + \vec{N}_4| \qquad \vec{N}_d = (\vec{N}_0 + \vec{N}_4) / |\vec{N}_0 + \vec{N}_4|$$

- The difference between Gouraud and Phong shading, as we'll see in more detail in the following sections, is that Gouraud uses these normals to compute color intensities at the vertices and then interpolates the intensities across the polygon, while Phong interpolates the normals across the polygon and then uses the interpolated normals to compute colors at each pixel.

○ Gouraud shading

- Consider the polygon, P, shown below, where we have computed vertex normals as shown by averaging against the surrounding polygons



- We will now use these vertices to compute intensities (color) at the vertices, usually by using the Phong illumination model with the vertex normal as the normal at that point.

FRACTALS

○ The History of Fractals

- Benoit Mandelbrot, an IBM researcher, is credited with developing fractal mathematics. Fractals were first recognized early in the 20th century, but they were placed, along with space filling curves and other weird and unpleasant things, into a general classification called mathematical monsters until Mandelbrot developed them and named them fractals because of their fractional dimension

- Most mathematics is Euclidean, where objects behave well. For example Euclidean objects have normals everywhere, except at a finite number of points of discontinuity called vertices, and finite Euclidean objects have well defined lengths or surface areas. They also have integral dimensions, and a point is one-dimensional, ~~two-dimensional~~, a circle is two-dimensional, and a sphere is three-dimensional. Once we get to fractal objects these restrictions go away

- The length of a fractal curve or the area of a fractal surface can depend on the length of the ruler being used to measure the curve, and can be infinite even if the curve is closed and is in a finite area. The object can be completely rough, so that it doesn't have a well defined normal anywhere. Finally, a fractal object can have a dimension like, say, 2.15, as compared to being restricted to integers.

- Man-made objects can usually be modeled well using Euclidean approaches, but it turns out that most natural objects are fractal and not euclidean in nature.

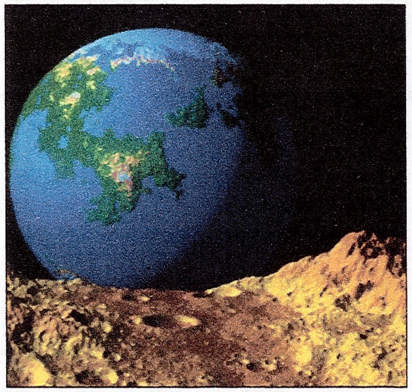
- As a simple (and classical) example, consider the border between France and Germany. This is a very well defined border since a number of wars have been fought that fixed it precisely

- However when France and Germany published the length of their mutual border they differed by 30% and both were basically correct. Much of the border follows the line of natural objects like rivers. You can get a map and trace along the river. However if you now get some larger scale maps you will find that what looked like a large curve in the river when you first measured it now has lots of small bends and twists within the curve, and if you follow them the length goes up.

- Or you can get out onto the river, and find that even the bends and twists that you can follow, and so on until you are tracing round rocks. This phenomenon is described by saying that the length of this curve depends on the length of the ruler that you are using to measure the curve. Clearly Germany used a smaller ruler when they were measuring the border.

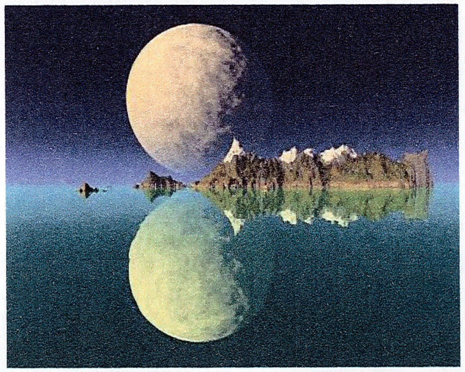
- Mandelbrot originally got into fractals when he was studying long term patterns in the stock and futures markets. He noted that if you looked at stock patterns over any period (e.g., week, month, year, decade, or century) the graph patterns looked very similar, and followed similar statistics. I.e., these curves showed self-similarity, where they behave in a consistent fashion at any scale, and this is another property of many fractal objects. E.g. if you look at an image of a fractal coast line and then zoom in, it usually won't be possible to tell the scale that you are at because a close-up fractal coastline looks just like the coastline itself.

- Mandelbrot quickly moved on to using fractals to develop natural looking objects, first with Sig Handelman, who developed wire frame fractal scenes, and then most significantly with Rich Voss, which led to recognition through the publication of Mandelbrot's book The Fractal Geometry of Nature, and through IBM's TV ads during the Calgary Olympics. Both featured the planetrise image, on the next page, for Mandelbrot's book it was the cover picture and for the IBM Olympics TV ad campaign it was part of a fractal animation that finished with a flythrough of the mountains on the planet



Fractal Planetrise by Rich Voss from Mandelbrot's *The Fractal Geometry of Nature*

- Fractals can also be created just for their beauty, Eg, the Mandelbrot and Julia sets have been analyzed more for their beauty thar for their mathematical importance. Computer art has also often included fractal components, both for their realism effects and for beauty. An example from Musgrave's work is shown below:

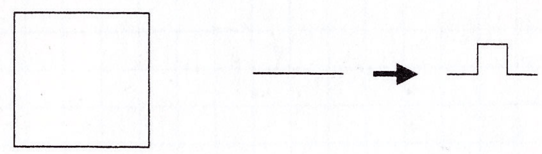


Blessed State by Kenton Musgrave

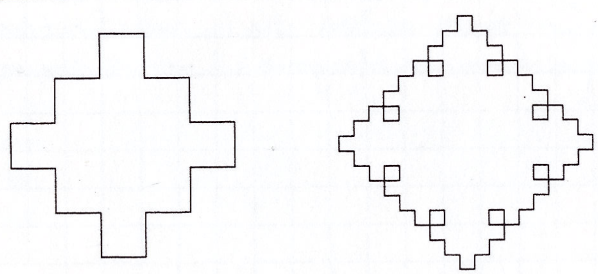
- In these notes I'll first look at an example of a Koch curve, because they provide the easiest approach to fractal dimension, and will then look at using fractals for natural objects

o Koch Curves and Generative Fractals

- Consider the square and generator shown below



- If we apply apply this generator to every straight line in the original curve which we'll call C_0 (the square), keeping the bump pointing out from the center, then we get the figure C_1, below. Applying the same generator to C_1 gives C_2, shown on the right. Obviously we can repeat this indefinitely getting each C_{i+1} by applying the generator to C_i.

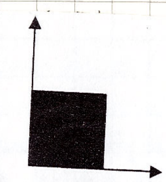


C_1 C_2

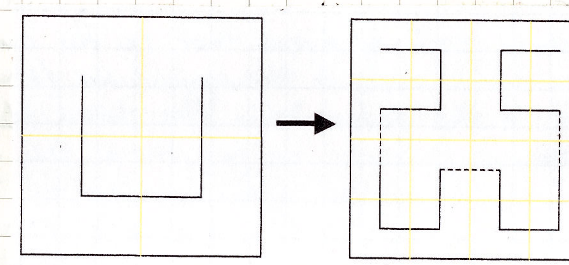
- The length of the curve C_i is $4(\frac{5}{3})^i$, assuming that the sides of the original square had unit length. Defining this Koch curve as $\lim_{i\to\infty} C_i$, the length of the curve is infinite. The curve is also infinitely rough, since every straight line has been broken down, and exhibits self similarity. E.g., the top piece

could have come from any of the ~~curve~~ C_i curves, $i>0$, but at different scales. So looking at it we have no idea which curve we are currently using.

- There are many other curves that can be constructed using generators like this. E.g., a space filling curve on the unit square has the property that for every point (x,y) where $0<x, y<1$, the curve passes through, or gets arbitrarily close to, (x,y), and also the curve never crosses itself. So if we draw the space filling curve it will look like:

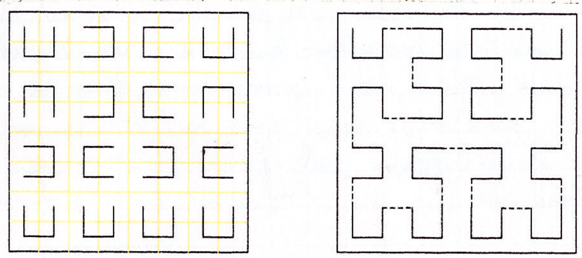


- There are a few space filling curves; I'll be using the Peano curve. The generator for the Peano curve is a bit more complex. We start with a U shape, as shown, and the generator converts it to the shape on the right which has 4 U shapes that are connected with lines.



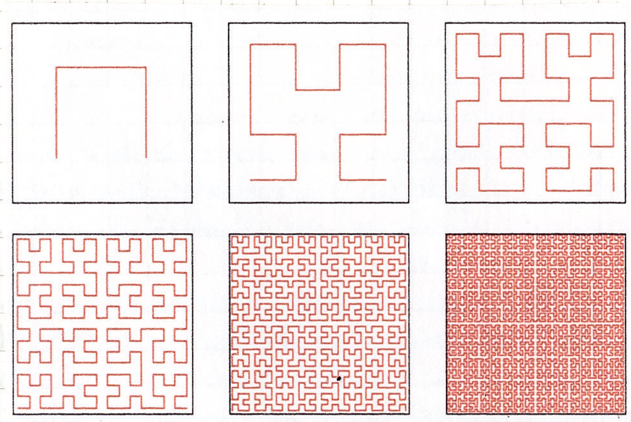
- The yellow grid lines are just there to help you maintain orientation, and will be more useful when I get to the next generation of the curve that is being generated.

- I've drawn the three connecting lines as dashed, to make the construction more obvious, but they are part of the curve and will be normal (not dashed) lines. To get the next generation of the curve, we delete the connecting lines, replace the four U shapes in the same way, with the appropriate orientation, and then reconnect the results, getting the curve on the next page. I've it in two steps; first without the connecting lines so that the four U shapes replacing each of the four U's in the previous curve are obvious, and then with the connecting lines included.



- The next curve is developed in the same way. We currently have 16 U shapes, each of which will generate four new new U's, and so it will have 64 U's, and so on. To see later generations of this curve, if you still need to be convinced that it does slowly fill the square, there is an applet at http://www.cut-the-knot.org/do_you_know/hilbert.shtml which lets you click through the first generations.

- However it should be fairly obvious that (a) the curve is slowly getting into all areas of the square, and (b) it never crosses itself. In the picture below I've shown the first six generations



- Now the Peano space filling curve is a simple curve, but it fills a space, and so what is its dimension? In fractal terms it will have a dimension of 2.0, which is intuitively reasonable

- To determine the fractal dimension of objects, without getting into some really horrible topology, we use self similarity. First I'll look at traditional 1D, 2D, and 3D objects (line, square, and cube) since fractal dimensions must be consistent with Euclidean dimensions for those objects, and then I'll move on to the Koch and Peano curves. All derivations that I use depend on self-similarity at different scales

- For the first three I'll use a scale of $1/2$, although any other inverse integer would work equally well. If we take a line, we can replace it with two lines that are scaled by $1/2$. For a square, it is equivalent to four squares which are scaled (in all dimensions) by $1/2$. For a cube it is 8. To compute the fractal dimension we use the the formula:

$$D = \frac{\log(n)}{\log\left(\frac{1}{s}\right)}$$

- Where n is the number of pieces that we get, and s is the scale. Obviously it doesn't matter which base log we use since we are dividing two logs. So using $\log_2$ for convenience, for a line we get $n=2$ and $\frac{1}{s}=2$, and so the dimension is 1. For a square $n=4$ and $\frac{1}{s}=2$, so $D=2$, and for a cube $n=8$ and $\frac{1}{s}=2$, so $D=3$. i.e., the formula follows the Euclidean values.

- For the Koch curve our generator produced 5 pieces each of which were $1/3$ the length of the original, and so the dimension is:

$$D = \frac{\log(5)}{\log(3)} = 1.47$$

- For the Peano space filler, if we just look at the U shapes, which we replace one U with four U's which are scaled by $1/2$, and so just like the square this has a fractal dimension of 2, which is reasonable since it fills a 2D space.

Fractals | Objects | CIE Chromaticity | HSV (HSB) Model | Other Color

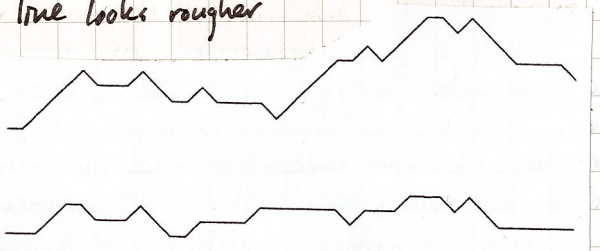
Modeling Natural Objects

- I'll concentrate mainly on the generation of 2D and 3D fractal mountains, since they provide a relatively simple example

- There are two basic approaches to developing fractal objects like mountains or planetary surfaces, Fractional Brownian Motion and Random Midpoint Displacement. The second used to be the most popular approach, but once whole terrains began to be modeled instead of just mountain ranges the Brownian approach became more common. I'll start with fractional Brownian motion

- Given a starting point, 2D Brownian motion generates the next point by generating a random vector (random in both direction and length). It then repeats this process to get, say, the 2D shape of a mountain range. The random direction will usually have a Gaussian distribution, where the mean of this distribution will be in the general direction that we want our mountain range profile to take and the variance will affect the fractal dimension.

- Higher variances will lead to rougher looking curves, and hence also higher fractal dimensions. I've faked this out with a couple of rather artificial examples below. I used rand() on Perl to give me a bunch of random numbers. For the first curve I went NE, E, or SE with equal probability, with vector lengths of 1 (horizontal) or 1.414 (NE or SE) to keep life simple. For the second line both NE and SE had probability 0.25, and E had probability 0.5, which was to reduce the variance. Clearly the first line looks rougher



- To get, say, a 3D mountain range instead of just this profile, we generate a 2D Brownian grid, where for each point we go in two directions (this also requires averaging two predicted points each time), and putting triangles on these points lets us render our mountain range

- This approach is commonly taken to generate planetary surfaces. In this case we put a Brownian elevation grid onto the planet surface. Anything above some threshold gets treated as land mass, and is rendered using triangles, and typically colored with snow above certain elevations, etc.

- Anything below the threshold is rendered as though at the threshold elevation, using water colors, possibly with different blues for different depths. Higher threshold values will give less land mass, and so for Earth-like planets the threshold level will be set higher than the planetary surface to get more water.

- If we also want atmospheric effects on the planet, then they are usually rendered as a fractal effect on a surrounding sphere

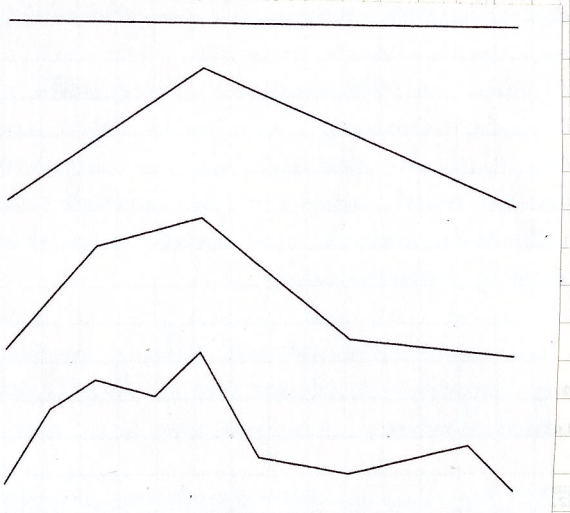
- ~~The basic idea, for a 2D mountain profile, is that one starts with a line, and then randomly displaces the midpoint up or down. This gives two lines, and one randomly displaces their midpoints, and keeps doing this recursively until the image is good enough.~~

- Random midpoint displacement was the first technique used to get good mountain ranges. It had the advantage that it was a recursive procedure and so the quality of the image depended on the amount of computational resources that you could commit to the task, and the image slowly improved as you went to deeper recursion levels.

- The basic idea, for a 2D mountain profile, is that one starts with a line, and then randomly displaces the midpoint up or down. This gives two lines, and one randomly displaces their midpoints and keeps doing this recursively until the image is good enough.

- An immediate improvement is to use a random point whose mean is the midpoint and displace this point instead. The amount of displacement and the random point are both determined by Gaussian distributions where, for the displacement the variance is of the form $|b-a|^{2H}$, where a and b are the line segment endpoints, which gives a fractal dimension of $2-H$ (for reasons that I haven't looked at for a long time and never want to see again)

- A completely artificial example of this is shown below.



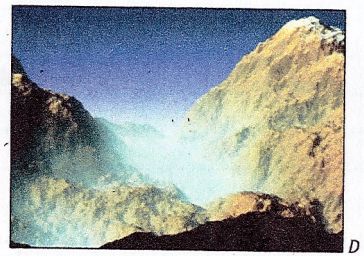
- For 3D mountains or other terrains using midpoint displacement we go from lines to tetrahedral (triangular pyramids). We start with a triangle and displace an approximation of the center, and connect the new point to the other three to give a tetrahedron. This gives three new triangles which we can then recursively divide by displacing their centers, and so on.

- For both Brownian and midpoint displacement we can take more control by initially deciding on a basic shape that we'd like our mountain range (or whatever) to take. For 2D midpoint displacement we initialize the basic profile of our mountain range ~~with an initial set of tetrahedrals. For Brownian we use the intended~~ and then proceed as usual.

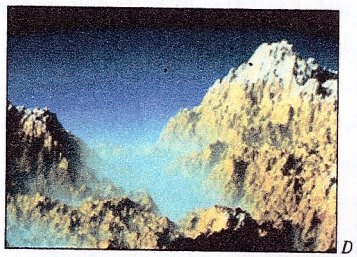
- In 3D we approximate the range with an initial set of tetrahedrals. For Brownian we use the intended shapes to control the mean vector direction for different components of the simulation

- As I discussed, but didn't justify, we can compute fractal dimensions from both Brownian and midpoint displacement fractal generators, based on the properties of the Gaussian distributions of their random generators. A useful rule is that natural objects will usually have a dimension that ends with .15. So, for a mountain profile 1.15 would be a reasonable dimension to aim for.

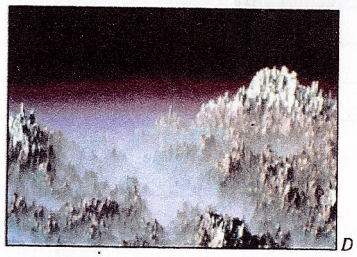
- For a mountain range or the surface of a natural looking planet a value at about 2.15 is likely to be good. Much higher and objects tend to look too rough, and much lower they tend to look too smooth. The three images below attempt to show this.



$D = 2.15$



$D = 2.5$



$D = 2.8$

- The three images were all drawn by Rich Voss with the same program, but for the first the fractal dimension was set to 2.15, for the second to 2.5, and for the third to 2.8

◦ The Mandelbrot Set and Julia Sets

- The Mandelbrot set is the most famous fractal. It has some applications in Chaos theory, where the Mandelbrot ratio (the ratio of the size of the smaller piece to the size of the large piece) has a tendency to keep appearing as a critical number in real chaotic systems, but most of the work on the set has been because of its beauty. As a result it is not great significance to computer graphics, and so my description will be brief.

- David Joyce has a nice page on the Mandelbrot set and Julia sets at http://aleph0.clarku.edu/~djoyce/julia/explorer.html, which you should go to for more details. The images below are from his page. The black values in both pictures are the complex points that are in the set. They are all connected (technically the connection is a julia set) although the connections can be so tiny that they don't appear, as in the image on the right.

- Joyce's page lets you zoom in on the Mandelbrot set. However he has roundoff errors when the image approaches a 10^{10} zoom which results in images at that, and higher, levels of zoom looking increasingly smooth. People who spend time looking at zoomed Mandelbrot images to find different pretty features tend to zoom to at least 10^{25} to search. Obviously this isn't possible with this package.

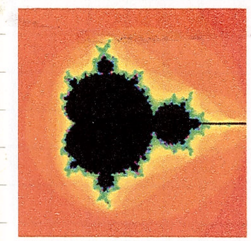
- The basic form of the Mandelbrot set comes from the complex equation

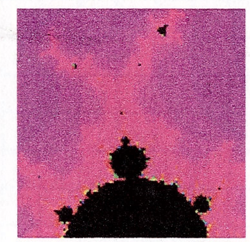
$$x_{i+1} = x_i^2 - c \text{, where } x_0 = 0 \text{ and } c \text{ is a complex number}$$

- For different values of c this iterative sequence will obviously either diverge to infinity at some point, or never diverge to infinity. For any complex number $c = r + is$ in the range $-1 < r < 2$ and $-1 < s < 1$, which will project to a pixel on the display, run the iteration up to, say, 256 times. If it hasn't diverged by then, which we'll interpret as $|x_i| > 2$ since once it reaches this value it must converge, assume that it never will and color that pixel black. If it diverges after, say, x iterations then color the pixel with whatever color is in the x^{th} entry of a 256 element color table

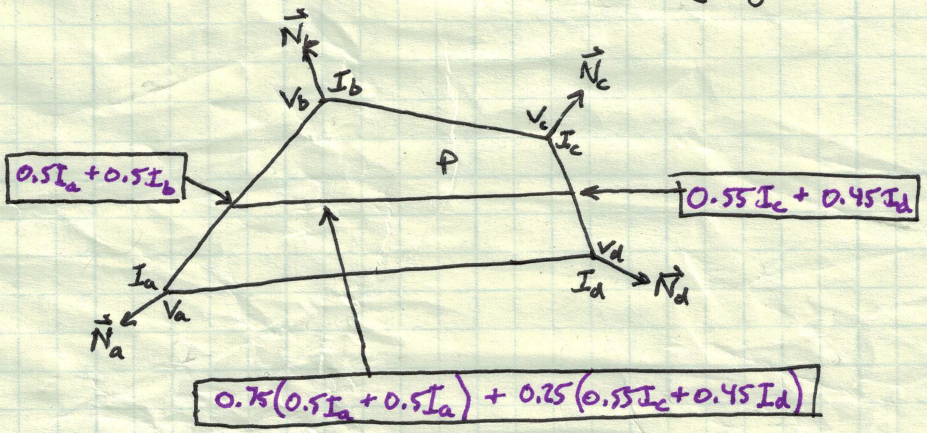
- Typically the colors in the table are selected for dramatic effect and are, for example, shades of reds and oranges.

- For more information on these sets go to David Joyce's page, or many other sites which let you work with these sets.





- This gives intensities I_a, I_b, I_c, and I_d at the vertices as shown below. Using Gouraud shading we now linearly interpolate these intensities down all of the edges of the polygons and then across rows giving, for example, the values shown in the boxes in the following figure.

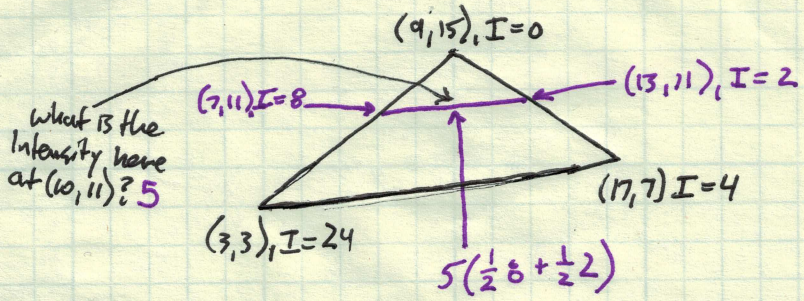


- This approach is that if, for example, we are 2/3 down an edge, then the intensity will be 2/3 of the intensity of the closest edge vertex and 1/3 of the intensity of the farthest edge vertex.

- To get the intensities of the interior points we linearly interpolate across scan lines using the edge intensities as the value on each end.

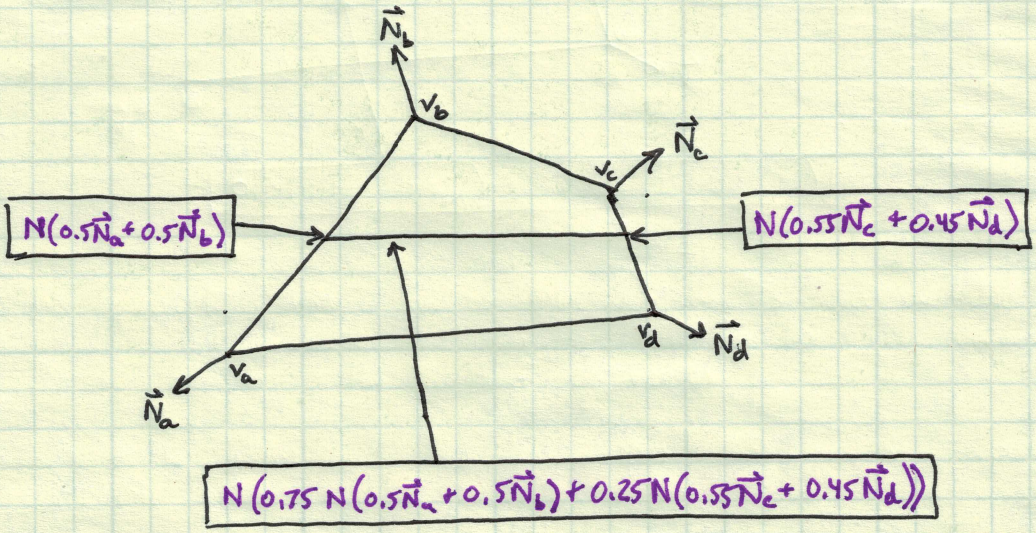
° Gouraud Example

- Consider the triangle below, where the vertex coordinates and intensities are shown

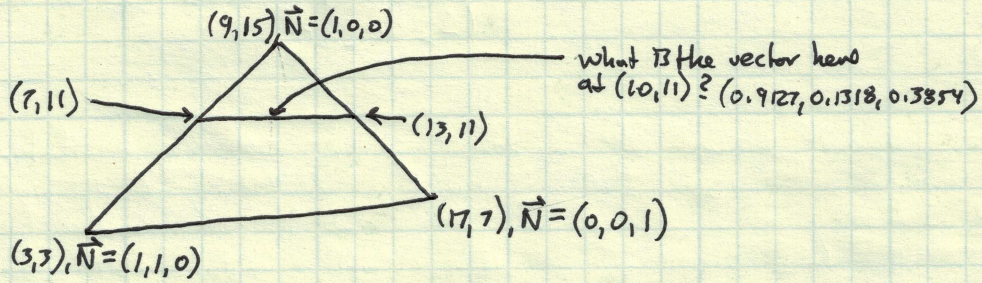


- Simple calculations say that the endpoints of the scanline through (10,11) are (7,11) and (13,11). (7,11) is 1/3 down the left edge, so has intensity $8(\frac{2}{3}0 + \frac{1}{3}24)$ and (13,11) is 1/2 down the right edge, so has intensity $2(\frac{1}{2}0 + \frac{1}{2}4)$. The midpoint of this scanline is (10,11) with intensity $5(\frac{1}{2}8 + \frac{1}{2}2)$

- Phong Shading
  - With Phong shading we interpolate the normals, instead of the intensities, down the edges of the polygon. E.g., in the figure that we used for Gouraud, the normal halfway down the left edge would have the value $0.5\vec{N}_a + 0.5\vec{N}_b$, which now needs to be normalized by dividing by its length
  - To prevent the figure from becoming too messy, I've added a function $N(\overrightarrow{vector})$ which returns the normalized vector. I.e., $N(\overrightarrow{vector}) = \overrightarrow{vector}/|\overrightarrow{vector}|$



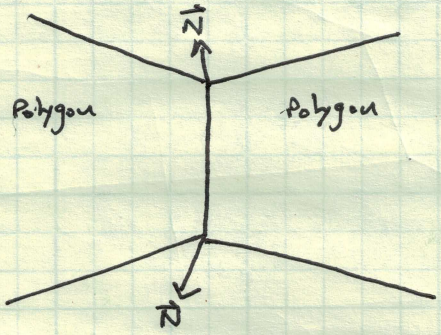
  - So apart from the fact that we are averaging vectors instead of intensities, and that we need to ensure that all of our vectors are unit by normalizing them as soon as they are created, the interpolation procedure is similar to the procedure that we used for Gouraud.
  - Phong Example:
    - Things rapidly get messy as we interpolate vectors, so we will use very simple vertex normals on the same figure that was used for the Gouraud example



    - Before we do anything else, remember that the vertex normals need to be unit vectors. Two of them are, but we need to normalize the vector at (3,3) to get (0.7071, 0.7071, 0.0) before we begin the interpolation

- The normal at (7, 11) is $N(\frac{2}{3}(1,0,0) + \frac{1}{3}(0.7071, 0.7071, 0))$, which is $N(0.9024, 0.2357, 0)$ or $(0.9675, 0.2527, 0)$
- The normal at (13, 11) is $N(\frac{1}{2}(1,0,0) + \frac{1}{2}(0,0,1))$, which is $N(\frac{1}{2}, 0, \frac{1}{2})$, or $(0.7071, 0, 0.7071)$
- The midpoint of the line between them has vector $N(0.8373, 0.1264, 0.3536)$ or $(0.9127, 0.1378, 0.3854)$

- **Why do Gouraud and Phong make the Object Appear Smooth?**
  - To see why the two smooth shading techniques work well, look at the shared edge between two polygons, after vertex normals have been computed.



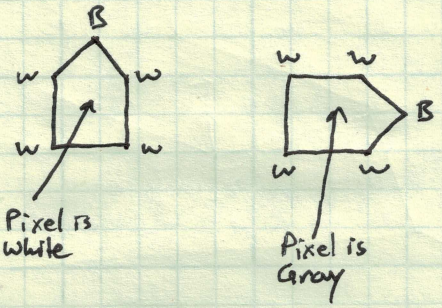
  - The vertex normals, $\vec{N}_1$ and $\vec{N}_2$, will be the same for both polygons, and so the intensities (directly for Gouraud or indirectly computed from normals for Phong) will be the same at any point for both polygons.
  - Given any scan line that passes through the edge, the two polygons linearly interpolate the intensities on either side of the point where the scan line intersects the edge, and so values on either side of the edge are almost identical.
  - Since there are no color discontinuities at the edge, it doesn't show up. In fact the only change at the edge is that the rate of change of the colors can change on opposite sides of the edge.

- **Comparison of Gouraud and Phong Shading**
  - Both methods share a problem, which is that although they do a pretty good job of smoothly shading the interior of the object, the silhouette of the object will still have jaggies.
  - E.g., consider the cylinder from the beginning of this section. Even though it may appear smooth in its interior, the bottom will still have three straight edges shown in the polygonal diagram.

- One solution is to use much smaller polygons near the outside edge of an object than we use in the interior, but increasing the number of polygons will, of course, increase the computational cost.

- Another problem with both methods is that they aren't invariant under rotation. I.e., as you rotate a polygon the internal colors are not always the same.

- Fortunately, this is usually not obvious, and it takes a relatively pathological case to demonstrate it. Eg., consider the two cases below, where the second figure is a 90° rotation of the first. Assume B is black and W is white.



- To see why this happened, remember that we scan down the edges and then across the scan lines. In the left figure the pixels on both ends of the scan line are white, so that on the scan line is white. In the right figure the scan line is interpolating from white to black, and so all pixels on the scan line are grays, darkening from white to black across the scan line.

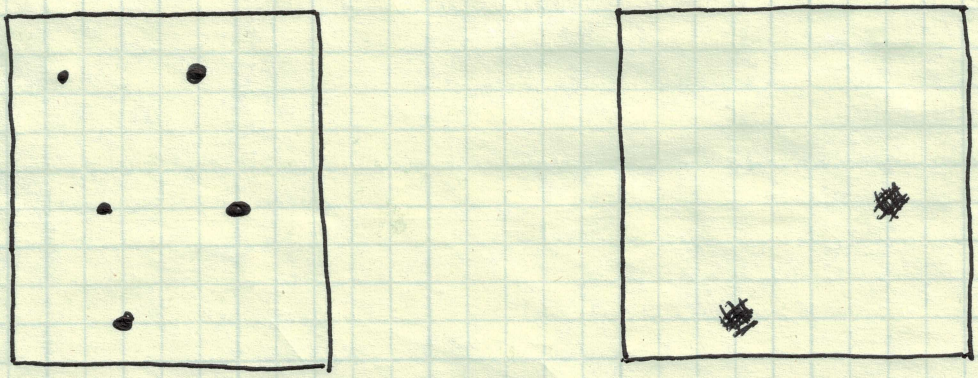
- Gouraud's only advantage over Phong and it is a very significant advantage, is that it is much faster since:

  - A.) It only has to compute the illumination model once for each vertex, and Phong has to do it for every displayed pixel.

  - B.) Interpolating and normalizing the vectors is much more work than interpolating the intensities.

- One disadvantage of Gouraud compared to Phong is that it doesn't deal well with hot spots, in particular if the size of the hotspots are small relative to the size of the polygon mesh which is defining the object

- E.g., say that we have the polygonal mesh shown below which has 81 polygons, bending around in 3D, where the black circles are places where there are supposed to be relatively small hotspots.



- With Phong, the picture will look just like it is supposed to. However with Gouraud the picture will look more like the picture on the right.

- I.e., some hot spots will vanish completely, and others will lose their definition and take on some of the shape of the polygons. This is because Gouraud is based on interpolating from the intensities at the vertices.

- If the hotspot doesn't cover at least one vertex, it can't appear in the averages over the vertex-colors. Also, when it is on a vertex the intensity will usually be very bright as compared to other vertices, so its effects, under interpolation, will be felt through most of the containing polygons

- To avoid these problems you need polygons that are small relative to the hotspot size, but that can be computationally expensive as it requires more polygons.

- Gouraud can also have a problem with mach banding, which doesn't show up with Phong. This is the appearance of light parallel stripes in the image, which can be partially blamed on the human visual system, which does a very good job of enhancing very subtle changes in intensity across an edge.

Computing Interpolations Efficiently

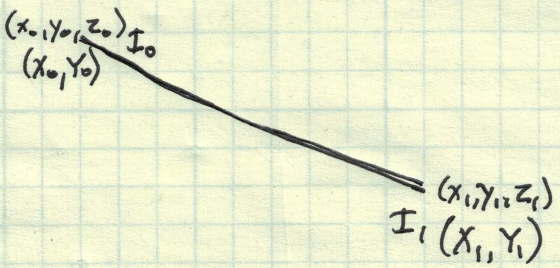
- There are two kinds of interpolations that go on, in either Gouraud or Phong:

  - First we interpolate along all the polygon edges
  - Second we interpolate across scan lines.

- Since the scan lines are on the screen this means we are only interested in edge values that are also pixel locations on the screen. So we will have some polygon with 3D vertices $(x_0, y_0, z_0), (x_1, y_1, z_1), \ldots$, which we will have to first project into screen pixel locations $(x_0, y_0), (x_1, y_1), \ldots$, and it is these values that will be used in the interpolation.

- Throughout this section I will assume that we are using Gouraud and interpolating intensities. This is just to simplify the notation. Phong will follow the same interpolation procedures.

- The figure below shows a typical edge for a polygon:



- We have an edge going from the 3D point (x_0, y_0, z_0) to (x_1, y_1, z_1), which when projected on the screen will have coordinates (X_0, Y_0) and (X_1, Y_1). Since we are wanting endpoints for scan lines, we need to interpolate at each integer Y value between Y_0 and Y_1, finding the appropriate X values and intensities

- We can do this with a simple loop like:

```
sort (X0,Y0) and (X1,Y1) so that Y0 > Y1
X = X0; Y = Y0; I = I0
N = Y0 - Y1;
δX = (X1 - X0)/N; δI = (I1 - I0)/N;
storePixel(X, Y, I);
while Y > Y1 do
    Y--;
    X += δX;
    I += δI;
    storePixel(round(X), Y, I);
end while
```

- The basic idea is that once we've computed the δX and δI by dividing the changes in X's and I's by the difference in Y's, we can just add in these X and intensity changes for each scan line (new Y)

- Unfortunately we still need to compute round(X) to get the nearest integer X, (this can be improved on by using Bresenham line drawing techniques to get the new X without any float operations)

- The function storePixel will store scan line endpoints with corresponding intensities for subsequent interpolation across the line. This interpolation technique is much easier since the Y values are constant. Now N becomes the difference in the X's, δI is computed as before, and a simple while loop on the X's adds in δI each time to the intensity value.

OpenGL Light and Material Properties

- Properties of light sources are set by statements of the form

  glLight*(light, parameter, value) * - fv...

  - light: a light source - GL_LIGHT0
  - parameter: GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_POSITION
  - value: 4-vector representing RGBA (in case of first 3) or a position vector in homogeneous coordinates

- Global Ambient light is set with

  glLightModelfv(GL_LIGHT_MODEL_AMBIENT, globamb)

- Material Properts at a vertex are set by statements of the form:

  glMaterial*(face, parameter, value) * fv....

  - face: GL_FRONT, GL_BACK, GL_FRONT_BACK
  - parameter: GL_AMBIENT_AND_DIFFUSE, GL_SPECULAR, GL_SHININESS GL_EMISSION
  - value: 4-vector for RGBA

- You can set the normals with

  glEnableClientState(GL_NORMAL_ARRAY)

  ~~glNormal3fv(type,~~

- You can set the Shading Model with

  glShadeModel(GL_FLAT)

  glShadeModel(GL_SMOOTH) → Default

For Next Time

- Read Ch. 11
- Review past lectures
- There will be no class (I will be out of town)
- There will be a video lecture
- Enjoy!