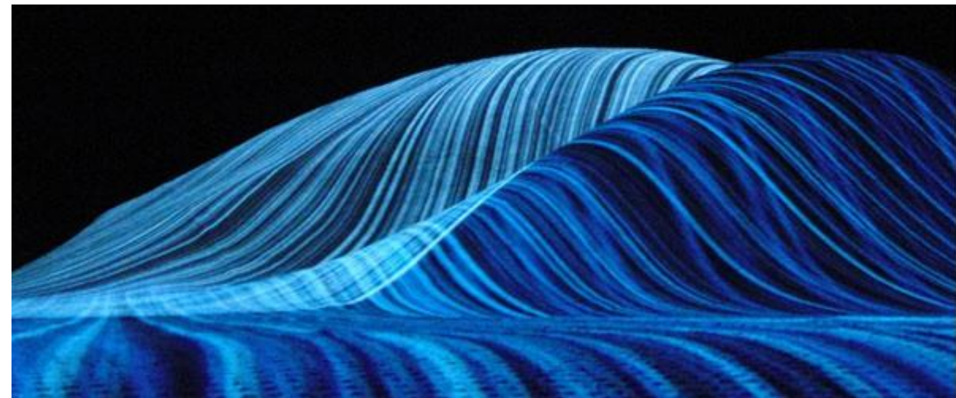
Crash Course In Graphics

In Real Time Interactive Simulations

Take a journey…

- To a land where everything is done 60 times a second or faster
  - Loading resources
  - Unloading resources
  - Receiving input
  - Executing AI to figure out what entities should do about the input
  - Executing logic to manage the simulation state
  - Executing physics to figure out how all the objects should move
  - Drawing everything to the screen

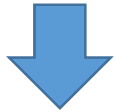


Take a journey

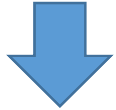
- This is why graphics cards are so important
- Massive parallelism
- The CPU passes a big list of vertexes with associated information (color, normal vector, ect.) and the GPU calculates a color for every pixel on the screen.
- Process is referred to typically as “The Graphics Pipeline”

Graphics Pipeline

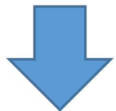
- Model Space



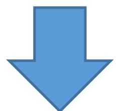
- World Space



- Eye Space



- Projection Space

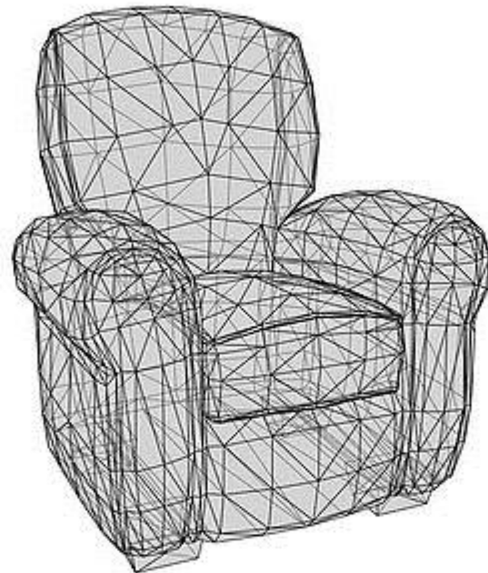


- Screen or CPU Memory

Fancy things are done by manually manipulating these steps in manually written graphics programs. Commonly these programs are written in pairs referred to as the "Vertex Shader" and the "Pixel/ Fragent Shader"

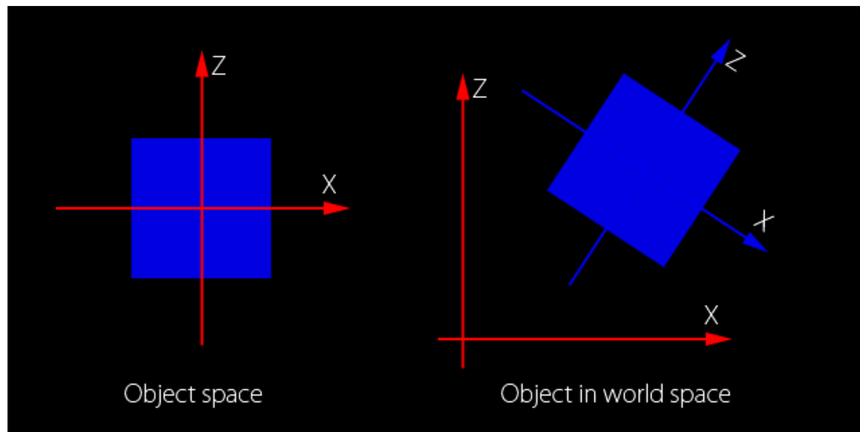
Graphics Pipeline

- Model Space
- Every drawn object in the simulation is at the center of its own coordinate system in isolation.
- This is the format that is output by artists and stored on disk



Graphics Pipeline

- World Space
- Every object produces a transformation matrix that is multiplied by each vertex to compute the corresponding vertex position in World Space



X-Rotation in 3D

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Z-Rotation in 3D

$$\begin{bmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scale in 3D

$$\begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$(4x4)*(4x1) = (4x1)$

Y-Rotation in 3D

$$\begin{bmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Translation in 3D

$$\begin{bmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Matrix Multiplication

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \\ q \end{bmatrix}$$

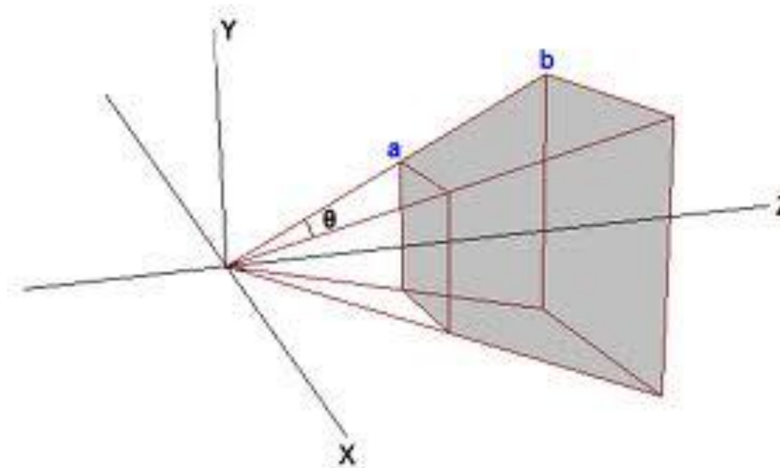
Graphics Pipeline

- Eye Space
- The “Eye” camera in the simulation is placed at the origin of the coordinate system looking straight down the positive Z axis.
- Accomplished with a single matrix that is multiplied by every vertex in every object in world space.

Graphics Pipeline

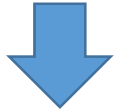
- Projection Space
- Homogeneous Geometry is used to warp eye space vertex’s into pixel locations (X, Y) [0.0 - 1.0], and depth values Z[0.0 – 1.0]
- Homogenous warping is what makes things father away look small
- Like World->Eye space transformation, accomplished with a single matrix multiplied by each vertex.

$$glFrustum(l, r, b, t, n, f) = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

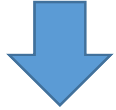


Graphics Pipeline

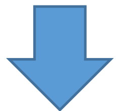
- Model Space



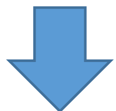
- World Space



- Eye Space



- Projection Space

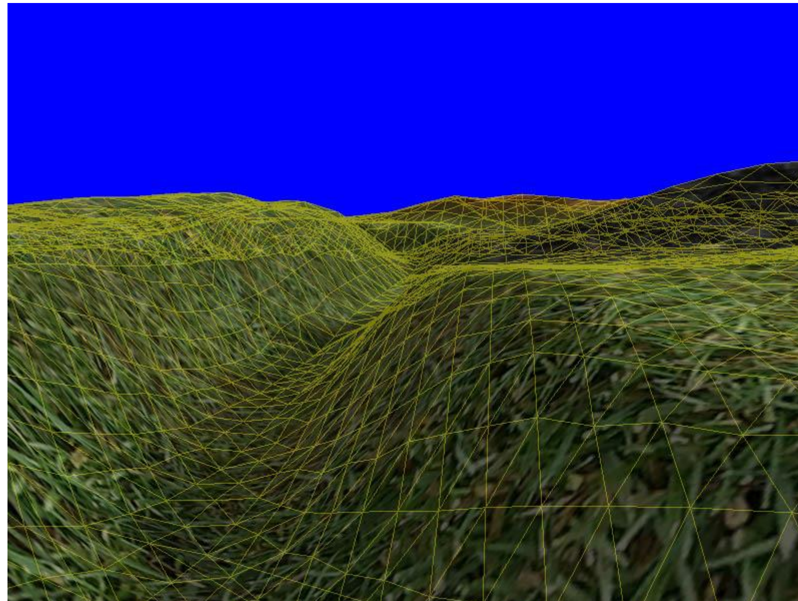


- Screen or CPU Memory

Fancy things are done by manually manipulating these steps in manually written graphics programs. Commonly these programs are written in pairs referred to as the "Vertex Shader" and the "Pixel/ Fragent Shader"

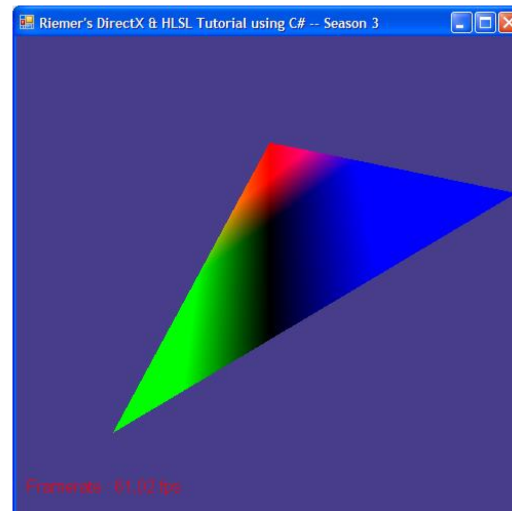
Graphics Programing

- Vertex Shader
- A program that executes all needed operations on each vertex
- Receives all the needed transformation matrices as input, and can receive all sorts of other inputs (i.e. surface normal).



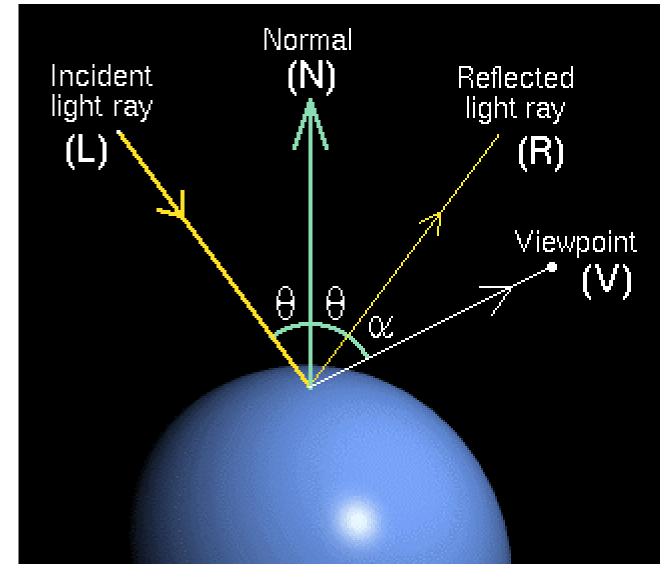
Graphics Programming

- Pixel Shader
- Calculates the color of each pixel that falls between vertex's output by the vertex shader
- Receives interpolated values of variables output for each vertex in the Vertex Shader (graphics card knows which vertex's make up which triangles)



Lighting

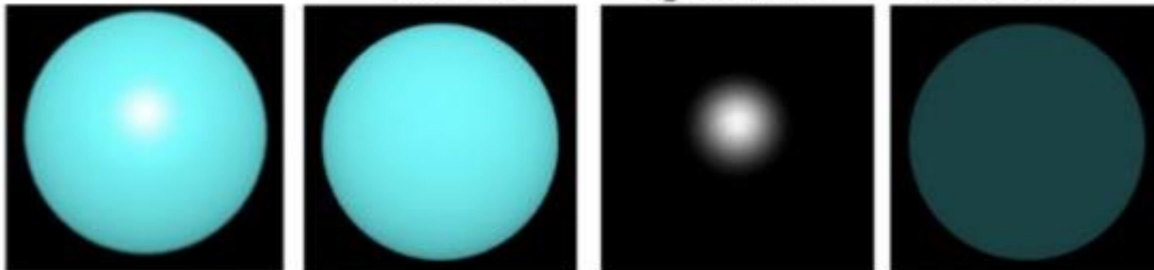
- Foundation of modern lighting was made with a model called “Phong” shading. Named after the inventor.
- Implementing this is the activity I sent earlier (easier than it looks)

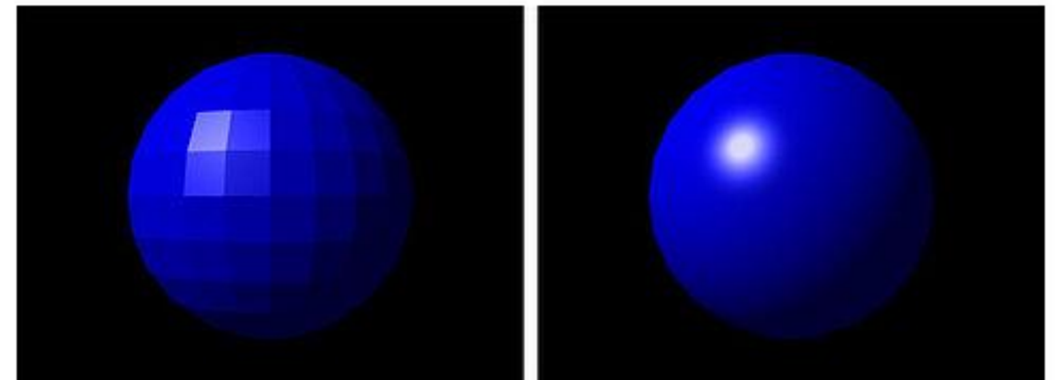


Phong Lighting Equation

$$I = K_d L_d (l.n) + K_s L_s (r.v)^{\alpha} + K_a L_a$$

diffuse specular ambient

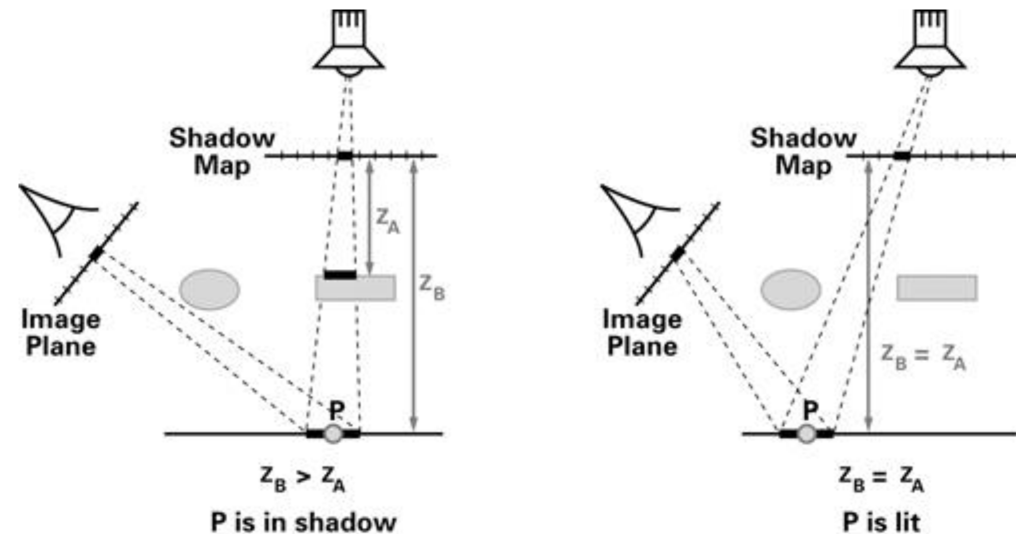




FLAT SHADING PHONG SHADING

Shadows (map technique)

- Uses two passes through the graphics pipeline
- First pass pretends like the eye is at the light's position and outputs to CPU memory the distance of the closest object per pixel on the would be screen from the light.
- Second pass draws the scene like normal but uses the information from the first pass to darken pixels in shadow.

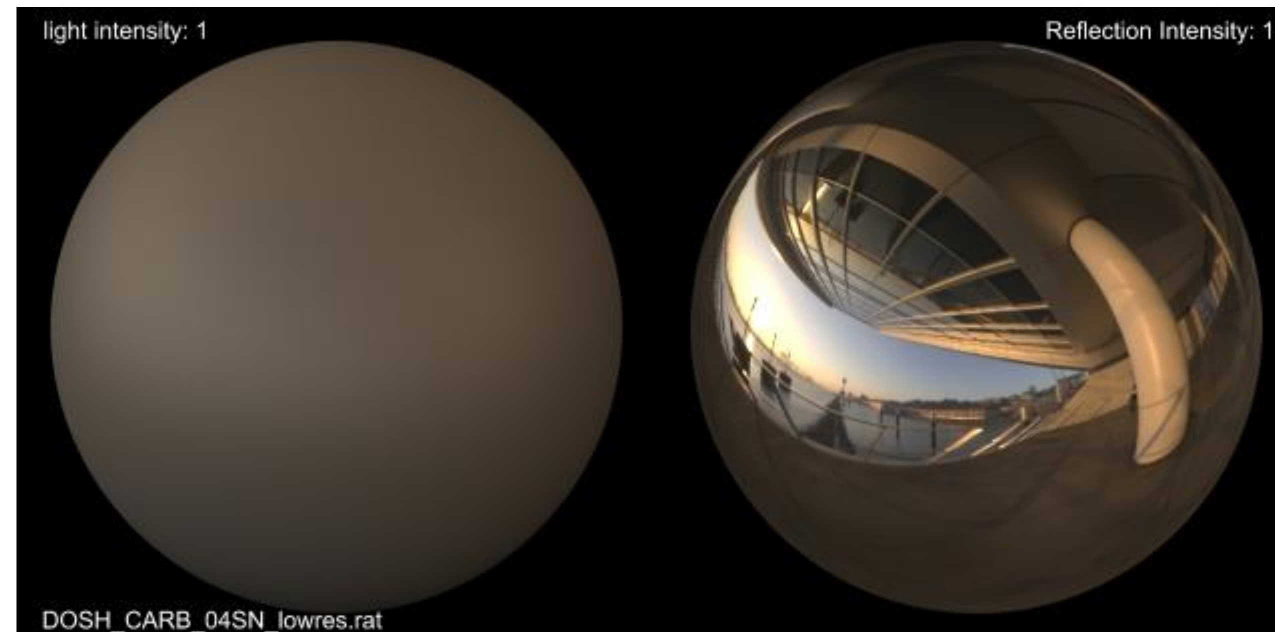


Reflections

- Similar to shadows
- Place object at the origin and produce a screen map (frame buffer) in CPU memory from the color at the direction of the surface normal
- One map for each side of object
- Second pass: When looking

at a reflective object, use the first

pass color

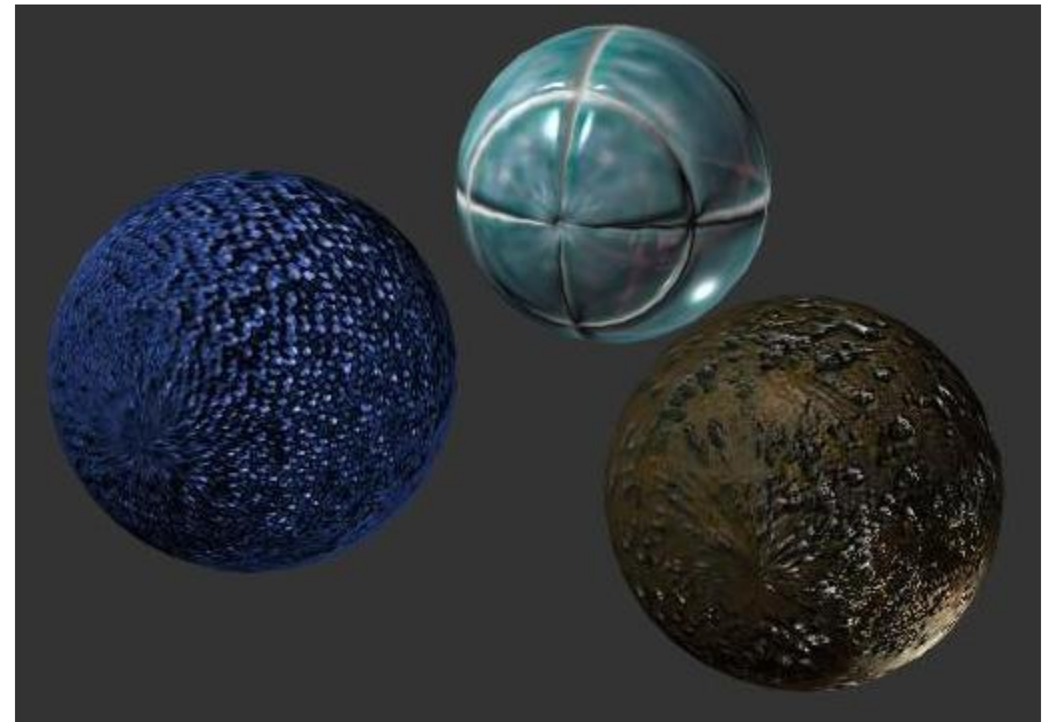


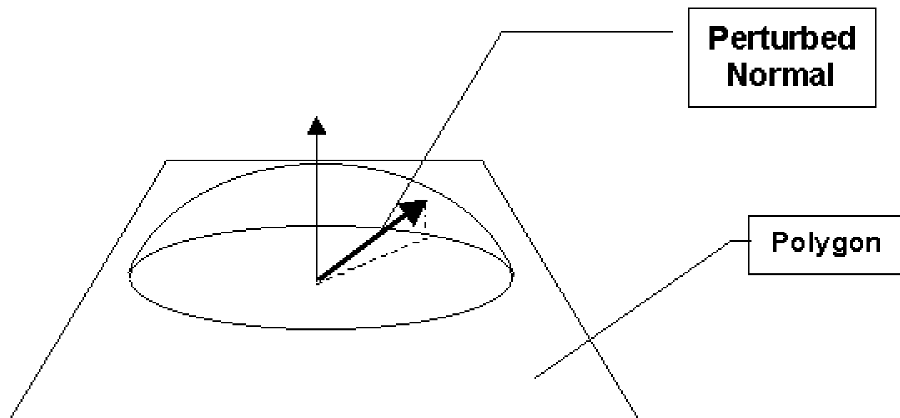
Bumps

- Input to the pipeline is both a color map for the object and a height map. Both resources on disk.
- Use partial derivatives across the

height map to artificially change the

normal vector used in the lighting.





Activity!

- Check out:
- OpenGL implementation of very basic Phong lighting.
- See Instructions.docx in the root directory for instructions.
- IM me for help.