

Levi D. Smith CS4451 Computer Graphics

## CS 4451: COURSE SYLLABUS

### SPRING SEMESTER 2000

**TIME:** Mon & Wed 4:30 - 6:00PM

**PLACE:** CCB 101

**INSTRUCTORS:** Prof. Norberto Ezquerro  
GVU Center, CoC 204  
norberto@cc.gatech.edu Phone: 4-4993  
OFFICE HOURS: By appointment

*Graphics  
Visualization  
& User Interface*

**TA:** Brooks Van Horn  
vanhorn@cc.gatech.edu  
OFFICE HOURS: TBD

**TEXT:** COMPUTER GRAPHICS: Principles and Practice,  
by Foley, VanDam, Feiner, and Hughes;  
C-Language Edition

**REFERENCES:** FUNDAMENTALS OF 3D COMPUTER  
GRAPHICS, Alan Watt

COMPUTER GRAPHICS, Hearn and Baker.

**GRADING:** Program assignments: 60%  
Quizzes (app. four): 40%

## PROGRAMS ASSIGNMENTS:

The following are guidelines and rules regarding programming assignments:

- All programming assignments must execute on the SGI workstations.
- Compiling and executing without errors will be considered minimal.
- Late assignments will not be accepted, except for one "late date."
- You will have one late submission reprieve to use at any time during the ~~quarter~~ *semester*

The "late date" option gives a one-day (24 hours) grace period only for one assignment. (Use this option wisely.)

- No incompletes will be given due to late assignment completion.
- You may discuss only high-level questions about assignments with other students. All the assignment work must be your own.
- All algorithm design and coding must be made individually by each student.

## DATES TO REMEMBER

- Friday February 2nd.: Assignment #1 Due
- Friday March 2rd.: Assignment #2 Due
- Friday March 30th.: Assignment #3 Due
- Friday April 27th.: Assignment #4 Due

NOTE: These due dates may change.

- QUIZZES: There will be approximately 4 quizzes spaced equally throughout the semester; dates TBD.

Open GL / C

### CS 4451: OVERVIEW OF TOPICS

The topics to be covered in class and in the programming assignments include, but are not limited to, the following: (The number in parenthesis refers to the corresponding chapter(s) in F&VD):

- Course intro; tests & assignments; 3D Viewing *Assign. #1 - Simple Rotation in Foley Book*
- 3D Viewing (6)
- Visible Surface Determination (VSD) (15)
- VSD; Illumination and Shading (15; 16) *Visual Surface Determination*
- Illumination and Shading (16)
- Illumination And Shading
- Animation *Color Theory / Perception*
- Antialiasing (14, 17, 19)
- Ray Tracing
- Radiosity (16)
- Surface Details: Texture and Bump Mapping (16)
- Volume Visualization (Surface and Volume Rendering)
- Volume Visualization

(The order and length of time associated with topics may vary; some additional topics may be added if there is sufficient interest and/or time.)

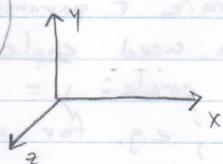
#### Case Studies (tentative)

- VR
- Scientific Visualization
- Animation and Control
- Particle Systems
- Graphics in Medicine

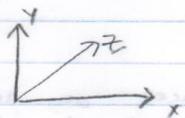
Levi D. Smith  
January 8, 2001  
CS 4451

## Basic Math + Transformations

- Coordinate Systems
- Vector Algebra
- Parametric Representations
- Homogeneous Coordinates



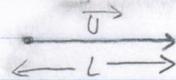
Right Handed System



Left Handed System

Pre-Multiply:  $C [B \cdot A]$  // A times B, times C

Vectors have magnitude + Direction



$$\vec{U} = (U_x, U_y, U_z)$$

$$\|\vec{U}\| = \sqrt{U_x^2 + U_y^2 + U_z^2}$$

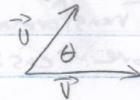
$$\vec{V} = (V_x, V_y, V_z)$$

$$\vec{U} + \vec{V} = ((U_x + V_x), (U_y + V_y), (U_z + V_z))$$

The Dot (Inner) Product of 2 Vectors

$$\vec{U} \cdot \vec{V} = (U_x \cdot V_x) + (U_y \cdot V_y) + (U_z \cdot V_z)$$

$$= \|\vec{U}\| \cdot \|\vec{V}\| \cdot \cos \theta$$



The cross product between  $\vec{U} + \vec{V}$

$$\begin{vmatrix} \hat{i} & \hat{j} & \hat{k} \\ U_x & U_y & U_z \\ V_x & V_y & V_z \end{vmatrix} = \hat{i}(U_y V_z - V_y U_z) - \hat{j}(U_x V_z - U_z V_x) + \hat{k}(U_x V_y - U_y V_x)$$

### Parametric Representation

There are 2 general forms for describing a curve or surface: algebraic + parametric

The Algebraic form can be used explicitly

$$y = f(x) \quad \text{e.g. for circle: } y = (R^2 - x^2)^{1/2}$$

or implicitly:  $F(x, y) = \phi$ ; e.g. for a circle

$$y^2 + x^2 - R^2 = \phi$$

The parametric form introduces an additional variable (or a set of variables)  $t, \tau$

$$x = x(t); \quad y = y(t); \quad t \in [\phi, 1]$$

For a circle,

$$x = R \cos(2\pi t)$$

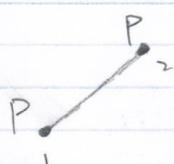
$$y = R \sin(2\pi t)$$

There are several reasons for using parametric representation:

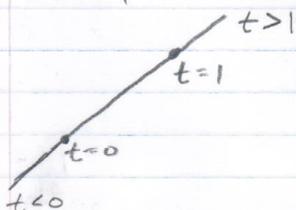
- Points on a curve can be computed sequentially (rather than solving the equations)
- Parametric curves + surfaces are easily transformed
- Most problems involve complex curves/surfaces not described by simple functions
- Transformations (translations, rotations, etc) are more easily done with param. rep.

Levi D. Smith  
January 8, 2001  
CS 4457

Parametric definition of a Line.  
Two points  $P_1 = (x_1, y_1, z_1)$  &  $P_2 = (x_2, y_2, z_2)$



$$\left. \begin{aligned} x &= x_1 + t(x_2 - x_1) \\ y &= y_1 + t(y_2 - y_1) \\ z &= z_1 + t(z_2 - z_1) \end{aligned} \right\} \begin{array}{l} \text{3D} \\ \text{Line} \end{array}$$

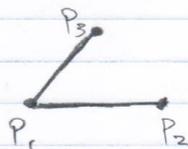


$$\begin{aligned} x &= x_1 + t \vec{V}_x \\ y &= y_1 + t \vec{V}_y \\ z &= z_1 + t \vec{V}_z \end{aligned}$$

$$L = P_1 + t(P_2 - P_1) \quad // \text{ Parametric Equation of a Line}$$

Plane

Given 3 points  $P_1, P_2, P_3$

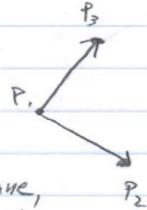


To get the normal to Plane,  
 $(P_2 - P_1) \times (P_3 - P_1) = N$

To see if a point  $P$  is on the Plane,  
 $N \cdot (P - P_1) = 0$   
// then it lies on the plane

January 10, 2001

For a plane,  
defined by  $P_1, P_2, P_3$



To get normal to the plane,  
 $(P_2 - P_1) \times (P_3 - P_1) = N$

Given any point  $P$ , to see if it lies on  
the plane

$$N \cdot (P - P_1) = 0 \Rightarrow \text{on the plane}$$



// Triangles make up the object

Equation of a plane

$$Ax + By + Cz + D = 0$$

// Implicit

// (everything is on

// the left)

$$A'x + B'y + C'z + D' = 0$$

$$A' = \frac{A}{d}, B' = \frac{B}{d}, C' = \frac{C}{d}, D' = \frac{D}{d}$$

$$d = \sqrt{A^2 + B^2 + C^2 + D^2}$$

//  $D$  - how far the plane is from the origin

For a point  $(x, y, z)$ , the distance from  
the point to the plane is

$$A'x + B'y + C'z + D' = \begin{cases} > 0 & // \text{front} \\ = 0 & \\ < 0 & // \text{behind} \end{cases}$$

Given  $Ax + By + Cz + D = 0$

Then  $[A, B, C]$  is a normal vector

### Basic Transformations

Translation

Rotation  
Scaling  
Shearing

// Translation - move object from one location  
// to another

// can also translate the the center of mass  
// for changeable objects

// Rotation - need axis you're rotating along  
// and degree of rotations

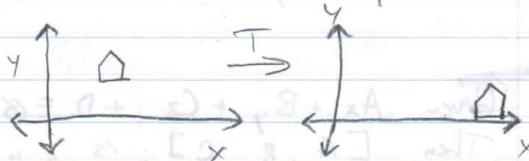
// Scaling - larger / smaller

// Shearing - scale one axis as a function of another

Geometric transformations are procedures for calculating new coordinate positions for points, as required by change in orientation, location, and/or size of object.

Translation: A straight line movement of an object from one position to another. The movement occurs by adding displacement / translation distance to coordinate positions

$$x \rightarrow x+a$$
$$y \rightarrow y+b$$



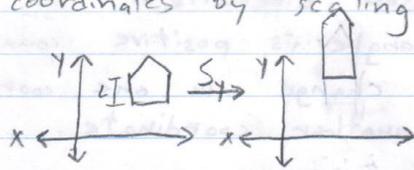
Proportions are preserved

To translate a line segment, translate end points

CS 4451  
January 10, 2001

$$\begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x+a \\ y+b \\ 1 \end{pmatrix}$$

Scaling: Alter the size of an object.  
It's done by multiplying boundary vertex coordinates by scaling factor  $S$ .



Note there has been a shift over all

Scaling is done about the origin.  
First translate, then scale, then translate back. // If you want to keep it in original position

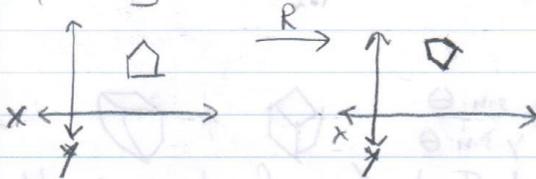
$$\begin{pmatrix} S_x & 0 \\ 0 & S_y \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x \cdot S_x \\ y \cdot S_y \end{pmatrix}$$

// Parallel lines are preserved ~~center~~

// Angles are not preserved

// Length is not preserved

Rotation: a transformation of points along circular paths. This is accomplished by specifying axis of rotation and an angle



Rotation is done about the origin  
To ~~translate~~ with respect to an arbitrary point  
rotate

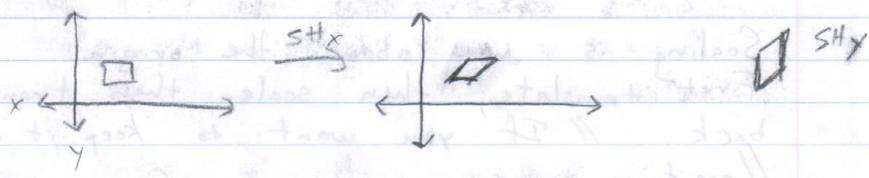
131122  
1005, 01

First translate to origin, then rotate, then translate back

$$T^{-1} R T P = P' \quad // \text{ rotate}$$

$$R \quad T$$

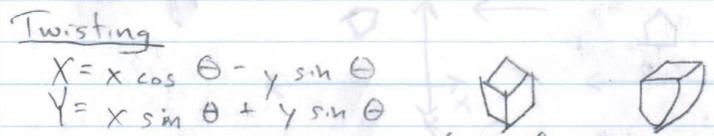
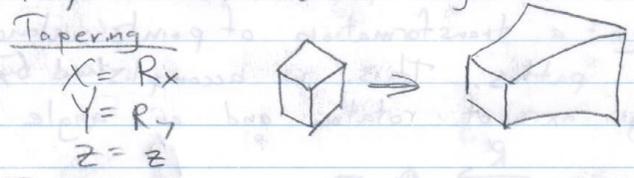
//  $\theta$  is positive counter clockwise  
Sign of rotation angle is positive counter clockwise  
Shear = Proportional change in one coordinate as a function of another coordinate



I causes a distortions of shape  

$$SH_x \begin{pmatrix} 1 & SH_x \\ 0 & 1 \end{pmatrix} \begin{pmatrix} X \\ Y \end{pmatrix}$$

Structure - Deforming transformations  
May be obtained through non-uniform scaling



Barr - at Cal Tech was first to do this

CS 3500 - 00 FF 00  
CS 4000 - FFFF 00  
LCC 3202 - FF 00 00  
CS 4451 - 00 FFFF

CS 4451

January 10, 2001

### Homogeneous Coordinates

It is possible to perform scaling, rotations, & shearing sequentially, starting with initial coordinates and operating on all the intermediate coordinates.

This can be done using matrix multiplication.

But this is inefficient, and usually requires to be added at the end. It's a lot more efficient (and elegant) to calculate the final coordinates directly from the initial coordinates using matrix multiplication & homogeneous coordinates.

In a homogeneous system, a vertex is represented as

$$\begin{pmatrix} X \\ Y \\ Z \\ W \end{pmatrix} \text{ for any scale factor } W \neq 0$$

The 3D cartesian coordinates are

$$X = \frac{X}{W} \quad Y = \frac{Y}{W} \quad Z = \frac{Z}{W}$$

// if  $W=0$ , then the point is at infinity  
// set  $W=1$  when starting

T, R, TP  
MP

Translation

$$TP = \begin{pmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{pmatrix}$$

$$T = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{pmatrix}$$

CS 471

CS 471 - 00 17 17  
CS 471 - 00 17 17  
CS 471 - 00 17 17  
CS 471 - 00 17 17

// last column is always displacement  
 $T_1$  and  $T_2$  are two translations,  
 $T_2 T_1 P$   
LW

$$T = T_2 \cdot T_1 \Rightarrow TP$$
$$T_2 T_1 P = TP$$

• This is called matrix multiplication, concatenation,  
concatenation.

$$T_2 \cdot T_1 = T_1 \cdot T_2 \quad // \text{commutative}$$

The 3D coordinate system is defined by the origin and the axes. The origin is the point where the axes intersect. The axes are the lines that extend from the origin in the positive and negative directions of the x, y, and z coordinates. The origin is the point (0, 0, 0). The x-axis is the horizontal axis, the y-axis is the vertical axis, and the z-axis is the depth axis. The origin is the point where the axes intersect. The axes are the lines that extend from the origin in the positive and negative directions of the x, y, and z coordinates. The origin is the point (0, 0, 0). The x-axis is the horizontal axis, the y-axis is the vertical axis, and the z-axis is the depth axis.

$$T = \begin{pmatrix} x & y & z & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} x & y & z & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} x & y & z & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

January 17, 2001

Assignment 1 tentative due date: Feb 5  
Brooks van Horn  
Office Hours  
Thurs < 10-11  
3-4

$$T \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

$$T = T_2 \cdot T_1 \\ \Rightarrow TP = P'$$

$$T_2 \cdot T_1 = T_1 \cdot T_2$$

// translations are commutative

Scale

$$SP = \begin{pmatrix} S_x \cdot x \\ S_y \cdot y \\ S_z \cdot z \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

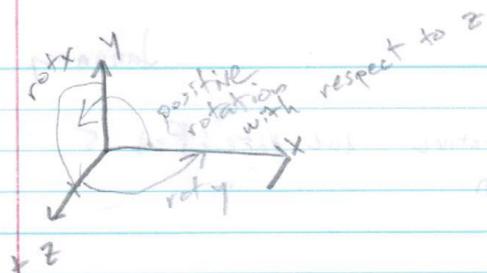
- Scaling is performed with respect to the origin
- Successive scaling operations can be obtained from their product

$$S_3 \cdot S_2 \cdot S_1 = S$$

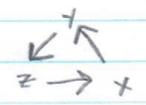
$$S_2 \cdot S_1 = S_1 \cdot S_2$$

Rotation

Positive Rotations are defined as follows:  
When looking from the positive direction along an axis, the rotation is counter-clockwise.



A 90% rotation transforms one axis to another



Rotation about z-axis

$$R_z = \begin{pmatrix} \cos \beta & -\sin \beta & 0 & 0 \\ \sin \beta & \cos \beta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation about x-axis

$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \beta & -\sin \beta & 0 \\ 0 & \sin \beta & \cos \beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation about y-axis

$$R_y = \begin{pmatrix} \cos \beta & 0 & \sin \beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \beta & 0 & \cos \beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- // all rows and cols have amplitude one
- // dot product of two rows or cols is zero

January 17, 2001

3x3 submatrices

→ rows & columns are unit vectors

- normal w.r.t. each other

- Determinant = 1

Preserve Distances & Angles

Shear with respect to  $x_i$

$$SH_{xy} = \begin{pmatrix} 1 & 0 & SH_x & 0 \\ 0 & 1 & SH_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

• Transformations can be combined together  
via matrix multiplication

• Each of them has a "reverse" counterpart

Translation:  $T^{-1}$  is obtained by negating the values of the translation factors

Scaling:  $S^{-1}$  is obtained by replacing  $S_x, S_y, S_z$  by their reciprocals,

Rotation:  $R^{-1}$  is obtained by negating the angles of rotation.

$$\underbrace{(SH \cdot S \cdot T^{-1} \cdot R \cdot T)}_{TR \cdot P} P = \begin{array}{l} // \text{with homogeneous} \\ // \text{coordinates} \end{array}$$

// Transform line segment by transforming

// end points

// Transform plane by transforming 3

// points

This Matrix product is called:

- compounding

- concatenation

- concatenation

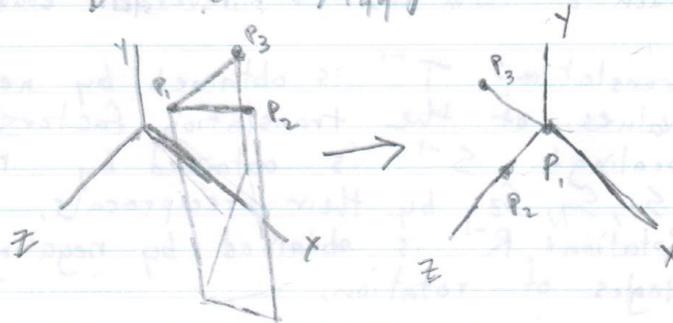
- composition

// parallel lines are preserved

An arbitrary sequence of rotation, translation, and scale matrices produce affine transformations.

In general,

$$\begin{pmatrix} M_{11} & M_{12} & M_{13} & M_{14} \\ M_{21} & M_{22} & M_{23} & M_{24} \\ M_{31} & M_{32} & M_{33} & M_{34} \\ 0 & 0 & 0 & M_{44} \end{pmatrix}$$



// Translate  $P_1$  to origin, rotate  $P_2$ , rotate  $P_3$

1) Translate  $P_1$  to origin

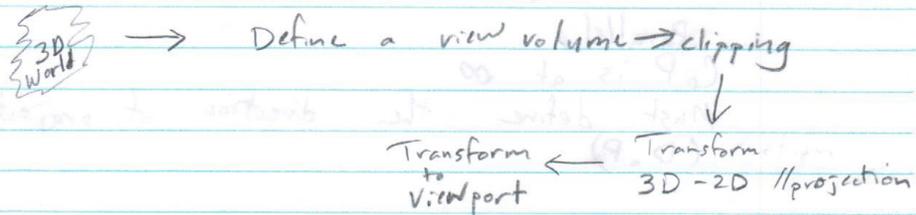
2) Rotate about  $Y$  axis  $\rightarrow$   
 $P_1, P_2$  lies in  $YZ$  Plane

3) Rotate about the  $X$  axis  $\rightarrow$   
 $P_1, P_2$  lies on the  $Z$  axis

4) Rotate about  $Z$ -axis  $\rightarrow$   
 $P_1, P_3$  lies in  $YZ$  plane

TP,

To go from a "3D world" to a 2D plane, a "viewing pipeline" is created. This pipeline can be viewed as consisting of several steps



Terminology

Projections are defined by straight projection rays ("projectors") emanate from a CoP (center of projection) passing through each point of an object and intersecting a Projection Plane

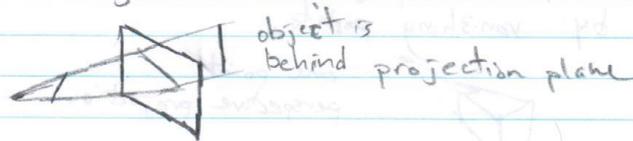


// Planar Geometric Projection

The class of projections are planar geometric projections. There are two classes:

- Perspective
- Parallel

Planar Projections



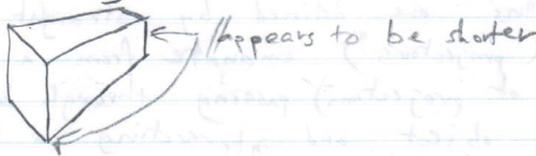
Perspective must define CoP



Parallel  
CoP is at  $\infty$

Must define the direction of projection  
(D.O.P)

Perspective projections give rise to  
foreshortening

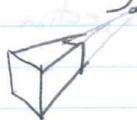


SIZE  $\propto \frac{1}{\text{distance}}$

Objects look realistic, but shape +  
exact dimensions may be lost.

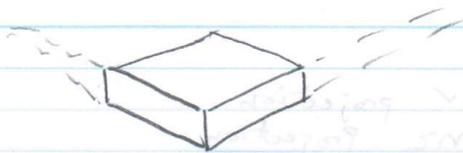
- distances are not preserved
- angles preserved only on faces  $\parallel$   
to projection plane
- parallelism is not preserved

Perspective projections are characterized  
by vanishing points.



one point  
perspective projection

January 22, 2001



two-point



three-point

Parallel projects are less realistic, but  $\neq$  foreshortening.

- Parallel lines remain  $\parallel$
- Angles are preserved only on faces  $\parallel$  to projection plane

There are 2 types:

- Oblique
- Orthographic

Orthographic:  $DoP \perp$  Projection Plane // orthogonal

There are 3 "elevations"

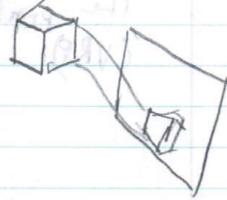
- Front
- Side
- Top (Plan)

There are also

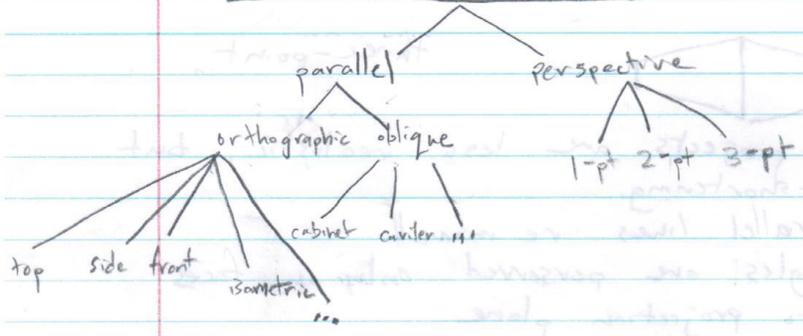
- axonometric
- isometric

Axonometric: Projection Plane is not orthogonal to a principle axis.

Isometric:  $DoP$  makes equal angles with each principal axis.



// 62.7 - cavalier projection  
Planar Geometric Projection



// perspective causes foreshortening

3D viewing

Two coordinate systems

WRC - World reference coordinate System, Representation of objects measured in physical units (e.g., meters)

VRC - Viewing ref. coord. system

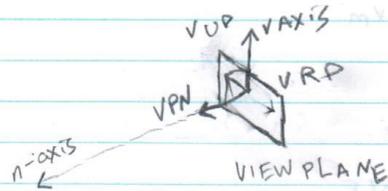
Define how the world objects will be viewed (displayed)

Anywhere w.r.t. WRT

First, specify a projection plane w.r.t. WRC

This is the viewplane and is defined by a point and a vector

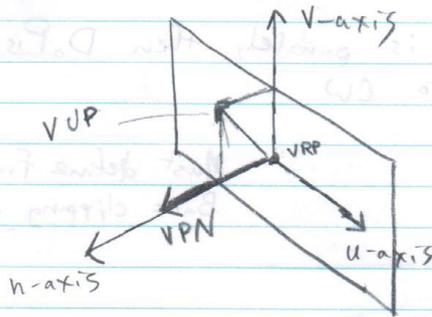
The point is the view reference point (VRP) the origin of the VRC



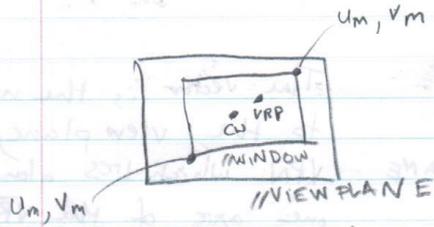
The vector is the normal to the view plane, VPN which lies along one axis of the VRC, the n-axis

defining VRC

A second axis is the "view-up" vector VUP, which helps determine what is pointing up ward, VUP determines the V-axis, the vertical axis on the viewplane. The V-axis is defined as the projection of VUP parallel to VPN onto the viewplane is coincident with the v-axis.



The u-axis is defined as  $u, v$  and  $n$  form a right-handed coordinate system. Next, a window on the view plane is defined to identify what part of the 3D world one wishes to view. The window is a rectangle defined w.r.t. VRP (but it need not be centered about VRP)



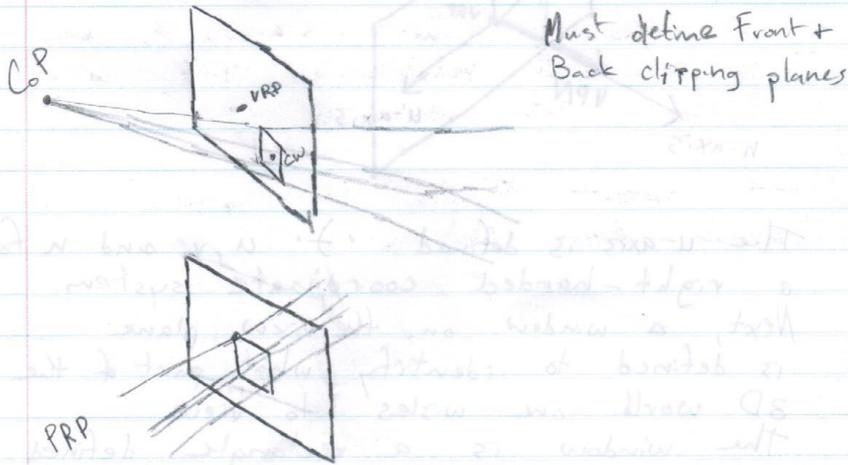
CW = center of window  
 The max and min  $u$  and  $v$  values are specified

Next, the part from which the world will be viewed is defined.

This requires specifying the projection reference point (PRP) and what type of projection.

If projection is perspective, the PRP is CoP

If projection is parallel, then D.o.P is from PRP to CW

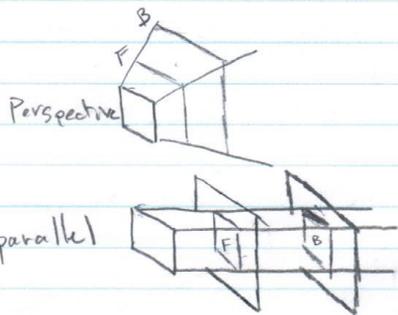


January 22, 2001

Perspective Projection  $\Rightarrow$  Semi-infinite  
pyramid view volume

Parallel Projection  $\Rightarrow$  Semi-infinite  
parallelepiped.

Oblique  $\parallel$  projection  $\Rightarrow$  sides of the  
view volume are not parallel to VPN



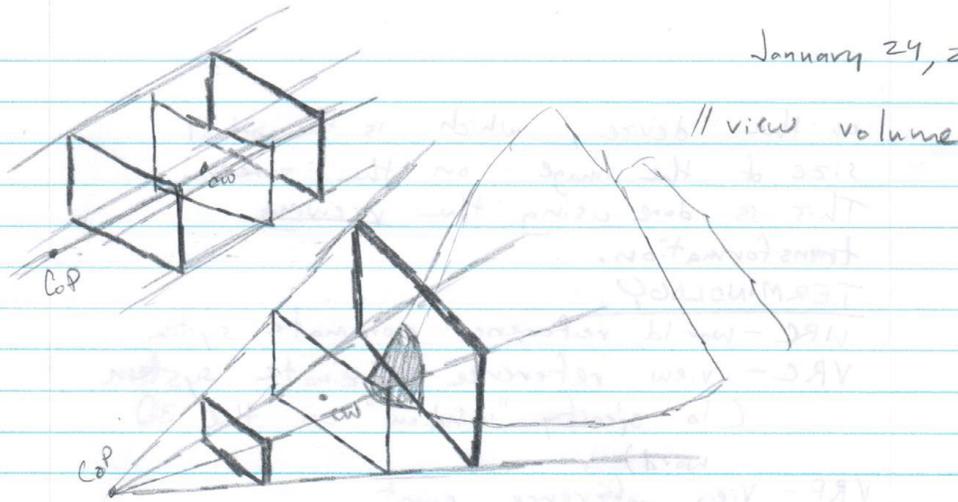
A finite view volume is obtained from from  
the F + B clipping planes.

The F + B planes are defined by the  
signed quantities (front distance F) and  
(back distance) B relative to VRP, with  
+ distance in the direction of VPN

Then convert Finite View Volumes  
Canonical View Volumes

CS 4451

January 29, 2001



// Finite view volumes

// Normalized view volumes

- A finite View Volume reduces the # objects to be displayed and constrains the "world" to the "world"
- The finite  $VV$  is converted into a "standardized"  $VV$  called a canonical  $VV$ , through a set of normalizing transformations
- The objects in the canonical  $VV$  are then clipped (in/out) and projected onto projection plane via the perspective transformation
- The next consideration is to define the 2D space within which the image to be displayed

17/11/20

on the device, which is the actual size of the image on the screen. This is done using the viewing transformation.

TERMINOLOGY

WRC - world reference coordinate system

VRC - view reference coordinate system (to specify "window" on the 3D world)

VRP - View reference point on the view plane (projection plane)

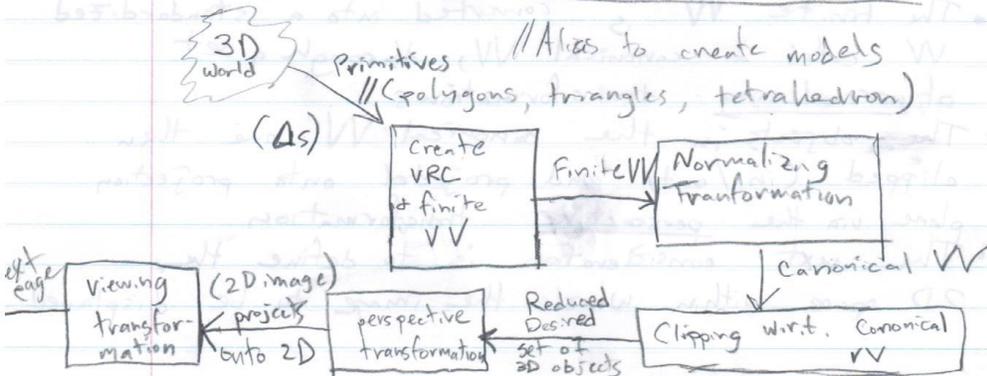
VIEW PLANE -

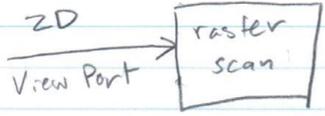
VPN - /view plane normal  $\times$  / (n-axis)

VUP - helps to define v-axis (n, u, v) to form a RH coordinate system

VIEW VOLUME

- CW (Center of Window) + corners
- PRP CoP (Perspective) DoP (Parallel)
- Front and Back Clipping planes





// 3 electron guns  
// 3 cathode ray tubes  
// CMY - printing - non-trivial sets

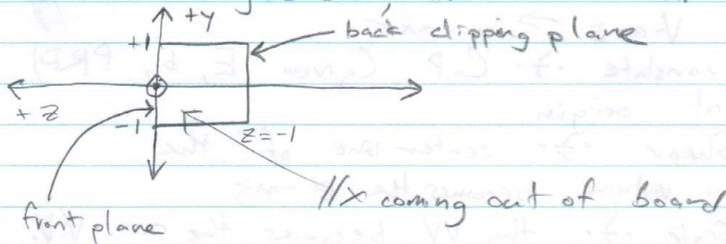
Levi D. Smith  
January 29, 2001

Project 1 Due - February 12, 2001

Quiz - February 7, 2001

- // Computer Vision - start with a photo, and
- // create 3D world from it.
- // interpolate - assume some objects remain still

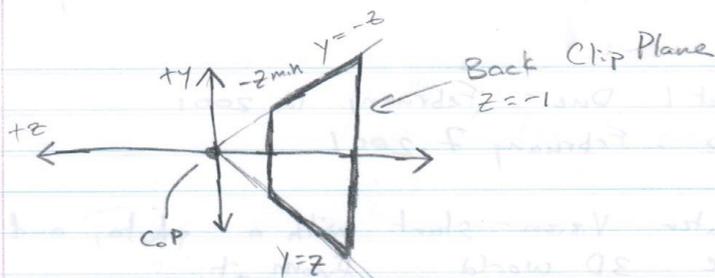
- Normalizing Transformation -  
To avoid clipping against any finite view volume (because intersection calculations can be difficult)
- A "regular" (normalized) view volume is obtained through the Normalizing Transformation. This operation produces the canonical view volume (VV).
- For  $\parallel$  Projections, the canonical VV is



$$\begin{array}{lll} x = -1 & y = -1 & z = 0 \\ x = 1 & y = 1 & z = -1 \end{array}$$

For Perspective Projections,

$$\begin{array}{lll} x = z & y = z & z = -1 \\ x = -z & y = z & z = -z \text{ min.} \end{array}$$



The goal is to perform normalizing transformation ( $N_{per}$  or  $N_{par}$ ) that transforms an arbitrary VV into the corresponding can. V.V.

### Normalizing Transf

- 1.) Translate VRP to origin
- 2.) Rotate the VRC  $\rightarrow$  VPV (v-axis) becomes the z-axis, the u-axis  $\rightarrow$  x-axis, & v-axis  $\rightarrow$  y-axis
- 3.) Translate  $\rightarrow$  CoP (given E, by PRP) is at origin.
- 4.) Shear  $\rightarrow$  center line of the view volume becomes the z-axis
- 5.) Scale  $\rightarrow$  the VV becomes the can. V.V.

Levi D. Smith

1.) Translate VRP to origin

$$\begin{pmatrix} 1 & 0 & 0 & -VRP_x \\ 0 & 1 & 0 & -VRP_y \\ 0 & 0 & 1 & -VRP_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

---

Levi D. Smith  
January 31, 2001  
CS 4451

## 2. Rotate VRC

We want to take  $\mu$  into  $(1, 0, 0)$

$$\begin{aligned} v &\rightarrow (0, 1, 0) \\ n &\rightarrow (0, 0, 1) \end{aligned}$$

First derive  $u \perp v$ :

$$u = \frac{v \times n}{\|v \times n\|}$$

$$\mu = \frac{v \times n \times n}{\|v \times n \times n\|}$$

$$v = n \times \mu$$

Recall unit row vectors of Rotation matrix

$R$  rotate into principle axes

$\Rightarrow$  VPN rotated into  $z$ -axis

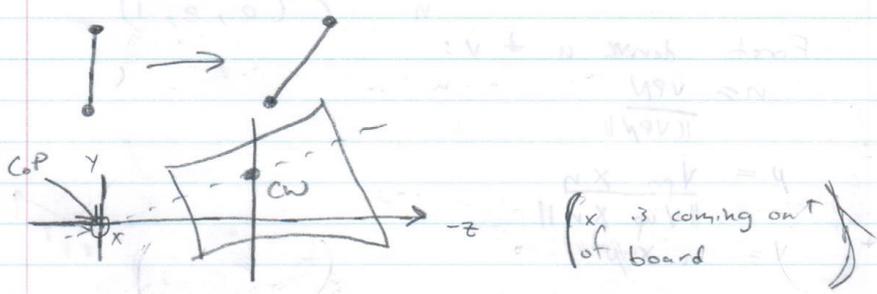
$$\begin{pmatrix} N_x & N_y & N_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

## 3. Translate $COP$ (PRP) to origin

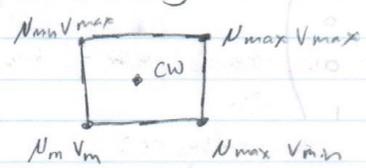
$$\begin{pmatrix} 1 & 0 & 0 & -PRP_x \\ 0 & 1 & 0 & -PRP_y \\ 0 & 0 & 1 & -PRP_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Handwritten notes at the top left of the page, including "Hand", "1000, 1000", and "1000".

4. Shear  $\rightarrow$  center line of the view volume becomes  $-z$ -axis



- Center line lies along the vector (CW - PRP) which is the direction of projection
- Want 'D.P.' along  $z$ -axis



$$CW = \begin{pmatrix} \frac{N_{max} + N_{min}}{2} \\ \frac{V_{max} + V_{min}}{2} \\ 0 \\ 1 \end{pmatrix}$$

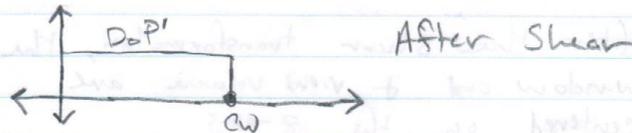
January 31, 2009

$$PRP = \begin{pmatrix} PRP_x \\ PRP_y \\ PRP_z \\ 1 \end{pmatrix}$$

$$D_oP = \begin{pmatrix} (U_{max} + U_{min})/2 & -PRP_x \\ (V_{max} + V_{min})/2 & +PRP_y \\ -PRP_z & \\ 1 & \end{pmatrix}$$

We want  $D_oP \rightarrow D_oP'$

$$SH \cdot D_oP = D_oP'$$



• Shear can be accomplished with (xy)-shear matrix for parallel projections.

• This will not affect z-coordinates

$SH_{xy}$  adds to x- and y-coordinates the term

$$z \cdot SH_x$$

$$z \cdot SH_y$$

• We want  $SH_x$  &  $SH_y$

$$D_oP' = \begin{pmatrix} 0 \\ 0 \\ D_oP_z \\ 1 \end{pmatrix} = SH_{xy} \cdot D_oP$$

$$SH_{xy} = \begin{pmatrix} 1 & 0 & SH_x & 0 \\ 0 & 1 & SH_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & SH_x & 0 \\ 0 & 1 & SH_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} D_o P_x \\ D_o P_y \\ D_o P_z \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ D_o P_z \\ 1 \end{pmatrix}$$

$$D_o P_x + D_o P_z SH_x = 0$$

$$SH_x = -\frac{D_o P_x}{D_o P_z}$$

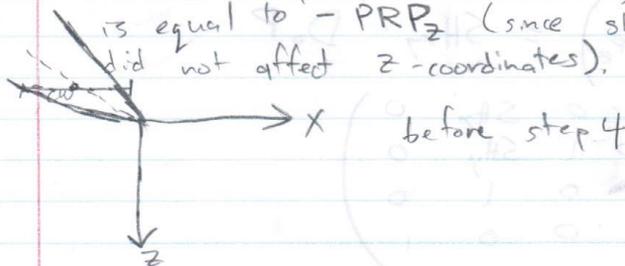
$$SH_y = -\frac{D_o P_y}{D_o P_z}$$

// VRP is not along DoP  
 // CW is along Z.

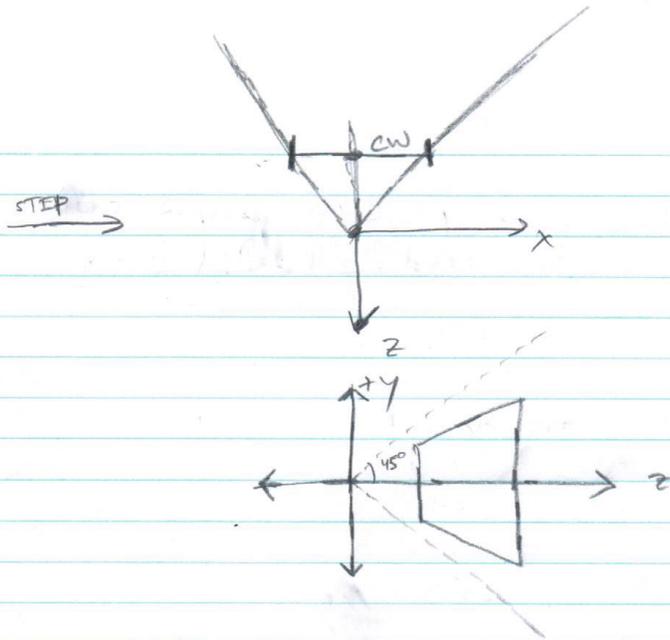
After the shear transformation, the window and view volume are centered on the z-axis also the VRP, which before step 3 (translation of PRP to origin) has now been translated (by -PRP) and sheared (in step 4):  $VRP \rightarrow VRP'$

$$VRP' = SH_{PAR} \cdot T(-PRP) \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \text{ After step 3}$$

The z-component of  $VRP'$ ,  $VRP'_z$  is equal to  $-PRP_z$  (since shearing did not affect z-coordinates).



January 31, 2007



Levi D. Smith

Quiz - Next Wednesday, Feb 14

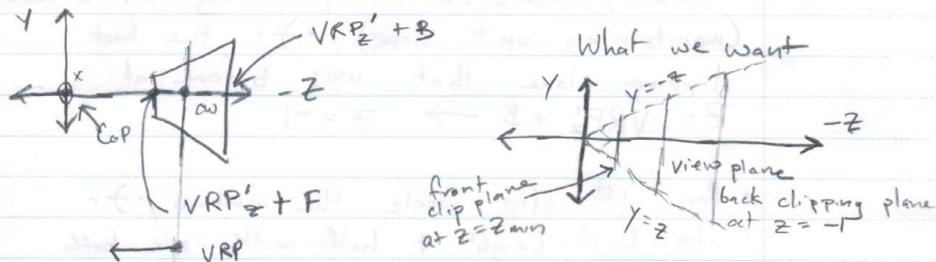
February 7, 2001

- After Shearing, the  $VP$  is centered on  $z$ -axis. Also, the  $VRP$  has now been translated ( $b_y - PRP$ ) and sheared

$$VRP \rightarrow VRP'$$

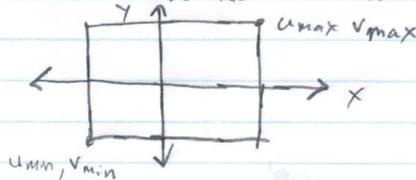
$$VRP' = SH_{PAR} \cdot T(-PRP) \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \quad \text{After Step 3}$$

The  $z$ -component of  $VRP'$  ( $VRP'_z$ ) is equal to  $-PRP_z$  ( $SH_{xy}$  does not affect  $z$  coordinate)



The window is now centered w.r.t  $z$ -axis

- $\therefore$  its bounds on the projection plane are



$$-\left(\frac{M_{max} - M_{min}}{2}\right) \leq x \leq \left(\frac{M_{max} - M_{min}}{2}\right)$$

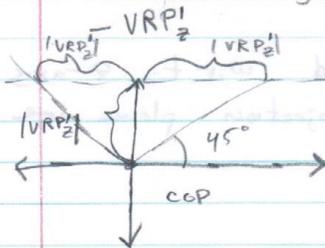
$$-\left(\frac{V_{max} - V_{min}}{2}\right) \leq y \leq \left(\frac{V_{max} - V_{min}}{2}\right)$$

• The final step is scaling along all 3 axes to create the canonical VV.

• This is done in two steps:  
 1<sup>st</sup> Scale differentially in  $x + y$  to give the slope planes bounding the volume (a unit slope).

• Second, scale uniformly along all 3 axes (maintaining unit slopes)  $\rightarrow$  the back clipping plane that was before at  $Z = VRP'_z + B \rightarrow Z = -1$

• for 1<sup>st</sup> step, scale the window  $\rightarrow$  its half-height + half-width are both



$$S_x \left( \frac{M_{MAX} - M_{MIN}}{Z} \right) = -VRP'_z$$

$$S_x = \frac{-Z (VRP'_z)}{M_{MAX} - M_{MIN}}$$

$$S_y = \frac{-Z (VRP'_z)}{V_{MAX} - V_{MIN}}$$

• For step 2 (Scale Uniformly) back clip plane is at  $z = -1$

• Scale by  $\frac{1}{VRP'_z + B} = S_x = S_y = S_z$

• Bringing these 2 steps together:

$$S_{perx} = \frac{2 VRP'_z}{(M_{MAX} - M_{MIN})(VRP'_z + B)}$$

$$S_{pery} = \frac{2 VRP'_z}{(V_{MAX} - V_{MIN})(VRP'_z + B)}$$

$$S_{perz} = \frac{1}{(VRP'_z + B)}$$

The projection plane and  $F + B$  clip planes have translated

$$Z_{MIN} = \frac{-VRP'_z + F}{(VRP'_z + B)}$$

$$Z_{MAX} = \frac{-VRP'_z + B}{(VRP'_z + B)} = -1$$

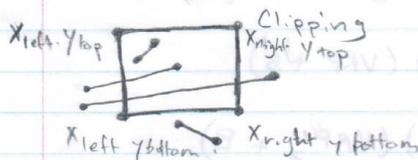
Thus, the normalizing transformation takes perspective projection finite View Volume into perspective projection canonical V.V.

$$N_{per} = S_{per} \cdot S_{H_{per}} \cdot T(-PRP) \cdot R \cdot T(-VRP)$$

Next: Clipping

7 several methods.

1. Brute Force
2. Cohen-Sutherland (2D)
3. Cyrus-Beck (2D)
4. Clip in Homogeneous Coordinates



A point is visible

$$\text{if } X_L < X < X_R \\ \text{and } Y_B < Y < Y_T$$

- consider only line segment endpoints.
- A segment is visible if both endpoints are inside the window  $\Rightarrow$  Trivial Accept
- If one endpoint is outside the window & the other is inside, Find intersection.
- If neither point is in, Find (both) intersections if they exist

// Brute Force find all intersections -  $N$

// don't do this

The Intersection calculations

Use Parametric repr

For a line from  $P_0$  to  $P_1$

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

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

$$z = z_0 + t(z_1 - z_0)$$

To calculate the intersection of a line with  $y=1$  plane,

set variable  $y=1$

Solve for  $t$ :

$$t = \frac{1 - y_0}{y_1 - y_0}$$

• if  $t$  is outside  $[0, 1]$ , the intersection is on the infinite line  $P_1 P_0$  but not between  $P_0$  and  $P_1$ .

• if  $t \in [0, 1]$ , then its value is substituted into the equations for  $x, z$  to find intersection points

$$x = x_0 + \left( \frac{1 - y_0}{y_1 - y_0} \right) (x_1 - x_0)$$

$$z = z_0 + \frac{(1 - y_0)(z_1 - z_0)}{(y_1 - y_0)}$$

The Cohen Sutherland

algorithm uses outcodes to make the  $t \in [0, 1]$  test unnecessary.

1. Build a 6-bit outcode for the endpoints of the line-segments
2. Check endpoint pairs for trivial accept / reject.
3. if trivial accept / reject not possible, then subdivide segments and test resulting outcodes.

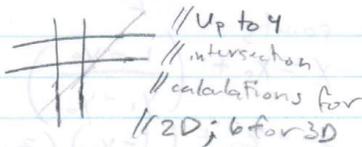
Each endpoint has a 6-bit code  
 A bit is true when condition is satisfied  
 as follows

For  
 Parallel

- Bit 1: point is above  $VV$  ( $y > 1$ )
- Bit 2: " " below  $VV$  ( $y < -1$ )
- Bit 3: " " right of  $VV$  ( $x > 1$ )
- Bit 4: " " left " " ( $x < -1$ )
- Bit 5: " " behind  $VV$  ( $z < -1$ )
- Bit 6: " " in front of  $VV$  ( $z > 1$ )

A segment is trivially accepted if both endpoints have a code of all 0s, or trivially rejected if corresponding bits in the two outcodes are equal to 1.

1001	1000	1010
0001	0000	0010
0101	0100	0010



For  
 Perspective  
 Canonical  
 $VV$

- Bit 1: ( $y > -z$ )
- Bit 2: ( $y < z$ )
- Bit 3: ( $x > -z$ )
- Bit 4: ( $x < z$ )
- Bit 5: ( $z < -1$ )
- Bit 6: ( $z > z_{min}$ )

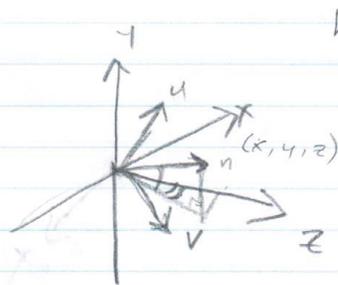
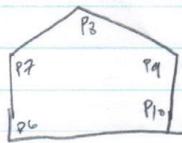
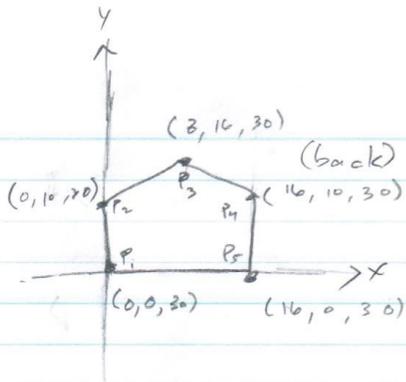
To calculate intersection of lines with (sloping) plane on  $y = z$

$$y_0 + t(y_1 - y_0) = z_0 + t(z_1 - z_0)$$

February 7, 2001

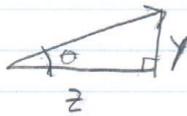
$$x = x_0 + \frac{(x - x_0)(z_0 - y_0)}{(y_1 - y_0) - (z_1 - z_0)}$$
$$y = y_0 + \frac{(y_1 - y_0)(z_0 - y_0)}{(y_1 - y_0) - (z_1 - z_0)}$$

3:30 Monday - Proj 1 Due



$n \rightarrow z$

~~$\theta =$~~



$$= \sin \theta = \frac{y}{\sqrt{x^2 + y^2}}$$

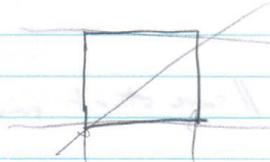
$$\theta = \sin^{-1} \left( \frac{y}{\sqrt{x^2 + y^2}} \right)$$

~~$\theta = \cos$~~

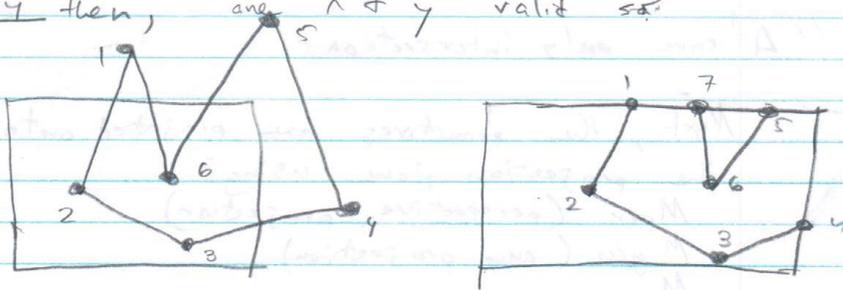


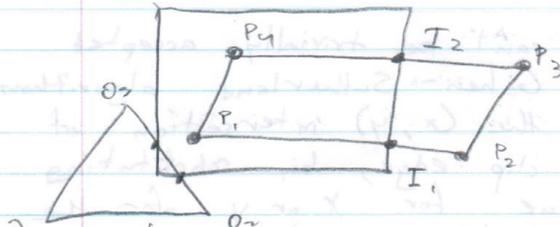
February 12, 2001

- For lines that can't be trivially accepted or rejected, the Cohen-Sutherland algorithm requires finding the  $(x, y)$  intersection of a line with a clip edge, by substituting the known value for  $x$  or  $y$  for the respective (vertical / horizontal) edge.
- For such lines, the Cyrus-Beck (Liang-Barsky) finds the value of the parameter  $t$  in the param. equations of the line for the point at which the segment intersects the infinite line containing the clip edge, then a series of simple comparisons are made



to determine which (if any) of those values correspond to actual intersections, only then, are  $x$  &  $y$  valid so?





$O_1$  polygon edges are traversed in predetermined order:

The edge from vertex  $i$  to  $i+1$  can be one of four types:

- ① Exit visible region:  $P_1 \rightarrow P_2$
- ② Completely outside VR:  $P_2 \rightarrow P_3$
- ③ Enter visible region:  $P_3 \rightarrow P_4$
- ④ Completely inside:  $P_4 \rightarrow P_1$

- ① Save intersection point // save start point
- ② Save nothing
- ③ Save Intersection + end point
- ④ Save both end points

// A save only intersections

Next, the primitives are projected onto the projection plane using =

$M_{per}$  (perspective, projection)

$M_{gen}$  (any projection)

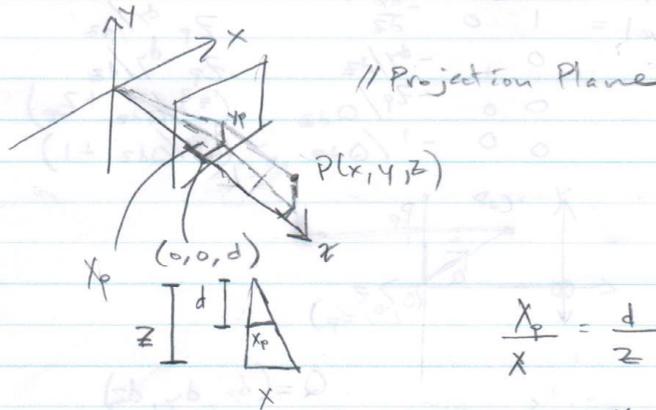
$M_{par}$



February 12, 2001

### 3 D $\rightarrow$ 2D Projection

- Assume CoP is at origin
- Perspective Projection with projection plane normal to  $z$ -axis at  $z=d$



$$\frac{x_p}{x} = \frac{d}{z}$$

$$x_p = \frac{x}{(z/d)}$$

$$y_p = \frac{y}{(z/d)}$$

Expressing the transformation in matrix form =

$$M_{per} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{pmatrix}$$

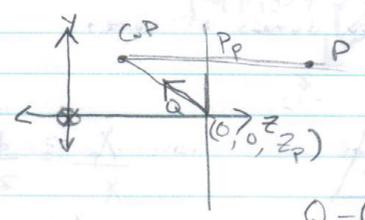
//  $d = -1$   
// for project

$$M_{per} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ z/d \end{pmatrix} = w$$

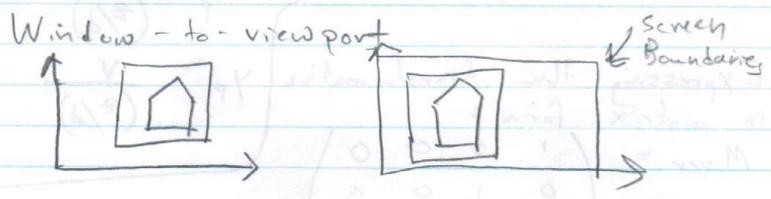


in 3D: 
$$\begin{pmatrix} x/(z/d) \\ y/(z/d) \\ d \end{pmatrix}$$

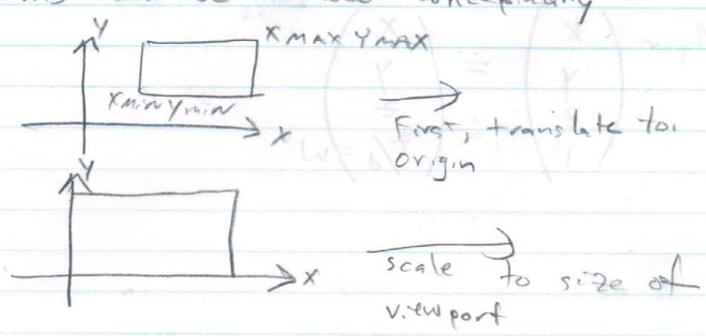
$$M_{\text{general}} = \begin{pmatrix} 1 & 0 & -\frac{dx}{dz} & z_p \frac{dx}{dz} \\ 0 & 1 & -\frac{dy}{dz} & z_p \frac{dy}{dz} \\ 0 & 0 & -\frac{z_p}{Qdz} & (\frac{z_p}{Qdz} + z_p) \\ 0 & 0 & -1/Qdz & (\frac{z_p}{Qdz} + 1) \end{pmatrix}$$

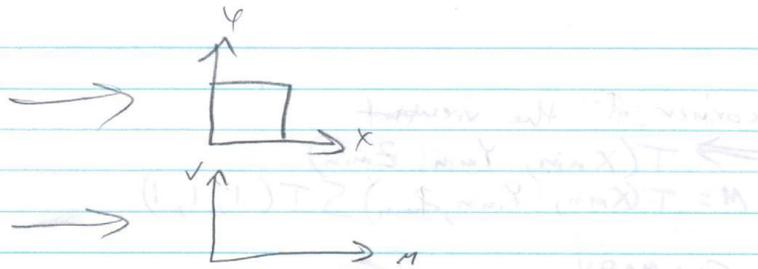


$$Q = (dx, dy, dz)$$

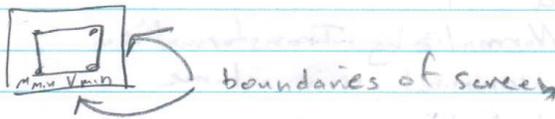


- if Window + viewport have different dimensions, A scaling is done
- This can be viewed conceptually



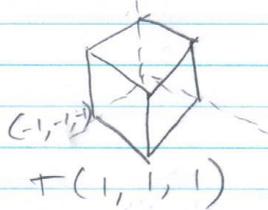


translate by  $(M_{min}, V_{min})$  to find position



to map into a 3D view port, assume a par. view volume, and consider 3-Step process

• first translate  $\rightarrow$  corner  $(-1, -1, -1)$  is at  $0, 0, 0$



Scale into size of view port

$$S_x = \left( \frac{2}{x_{max} - x_{min}} \right)^{-1}$$

$$S_y = \left( \frac{2}{y_{max} - y_{min}} \right)^{-1}$$

$$S_z = \left( \frac{1}{z_{max} - z_{min}} \right)^{-1}$$

Then, the scaled  $VV$  is translated to lower-left

corner of the viewport

$$\Rightarrow T(X_{min}, Y_{min}, Z_{min})$$

$$M = T(X_{min}, Y_{min}, Z_{min}) \cdot T(1, 1, 1)$$

### SUMMARY

- 1.) 3D coordinates  $\rightarrow$  homogeneous rep  
// add a "1"
- 2.) Apply Normalizing Transformations  
// get a canonical view volume
- 3.) Divide by  $w$
- 4.) Clip
- 5.) Back to homogeneous coordinates
- 6.) Project onto 2D
- 7.) Translate and scale to viewport  
// (device coordinates)
- 8.) Divide by  $w$



$$\begin{aligned} & \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \\ & \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & w \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \\ & \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \\ & \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \end{aligned}$$

that would be the device coordinates

February 19, 2001

## Visual Surface Determination

// (Hidden Surface Removal)

• Once the viewing specification is made, next step is to determine which  $\Delta$ 's, lines, and surfaces are visible.

•  $\exists$  2 families of approaches:

1. Determine which of  $n$  objects is visible at each pixel,

$\Rightarrow$  Image-Based (Image precision)

2. Compare Objects directly w.r.t. each other & determine which is closest to viewpoint

$\Rightarrow$  Object-Based (Object precision)

At a very high level:

1. Image-based  $\Rightarrow$  device resolution;  $O(m^2)$

For each pixel

- determine object closest to viewer pierced by a pixel projector.

- draw pixel in the appropriate color

2. Object-based

For each object

- determine which part(s) of object whose view is unobstructed by other "parts" of objects.

- draw those parts in the appropriate color

$\Rightarrow$  object resolution // looks better

$\sim O(n^2)$



Discrete Sampling



Continuous

VSD is done in 3D.  $\therefore$   
depth relationships are preserved.

Given 2 pts.  $P_1$  +  $P_2$ ,

Does either  
point obscure  
the other

Yes  $\Rightarrow$  Compare z-values  
to determine which is  
closest to viewport

No  $\Rightarrow$  Depth info not  
used directly

VSD is done after the normalizing  
transformation.

It is common to transform from  
perspective to parallel canonical VV  
(to preserve relative depth, straight lines,  
and planes)

### APPROACHES

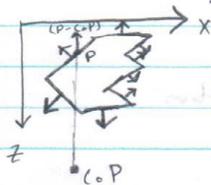
- 1. Back-Face Removal (Culling)
  - 2. Z-Buffer
  - 3. Depth-Sort
  - 4. Binary Space Partition (BSP) tree
  - 5. Scanline Algorithms
- Image Based  $\left\{ \begin{array}{l} 1. \\ 2. \end{array} \right.$
- Object Precision  $\left\{ \begin{array}{l} 3. \\ 4. \\ 5. \end{array} \right.$

February 19, 2001

Culling is an object based approach.  
 If an object can be approximated by a polyhedron:



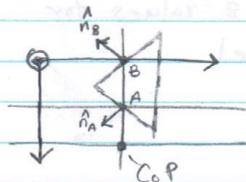
then its faces completely enclose its volume.  
 Assume each face has an outward-facing normal-vector.  
 If none of object's interior is intersected by front clip plane  
 then those  $\Delta$ 's whose normals point away from  
 observer lie on a part of the object  
 whose visibility is blocked by other  $\Delta$ 's



$\Rightarrow$  the back facing  $\Delta$ 's can be eliminated from further processing

Dot product  $\vec{n}$  and  $(P - C_oP) \geq 0$

$\Rightarrow$  back-facing plane of  $\Delta$



$$n \cdot (P - C_oP) : \begin{cases} \geq 0 \Rightarrow \text{Back} \\ < 0 \Rightarrow \text{Front} \end{cases}$$

$$C_oP = \begin{pmatrix} 1 \\ 0 \\ z \end{pmatrix}$$

$$\hat{n}_A = \begin{pmatrix} -\sqrt{2}/2 \\ 0 \\ \sqrt{2}/2 \end{pmatrix}; \quad \hat{n}_B = \begin{pmatrix} -\sqrt{2}/2 \\ 0 \\ -\sqrt{2}/2 \end{pmatrix}$$

$$A = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}; \quad B = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

$$\hat{n}_A \cdot (A - C_oP) = \begin{pmatrix} -\sqrt{2}/2 & 0 & \sqrt{2}/2 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}$$

$$= -\sqrt{2}/2 \Rightarrow \text{Front}$$

$$\hat{n}_B \cdot (B - C_oP) = \begin{pmatrix} -\sqrt{2}/2 & 0 & -\sqrt{2}/2 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ -2 \end{pmatrix}$$

$$= \sqrt{2} \Rightarrow \text{Back}$$

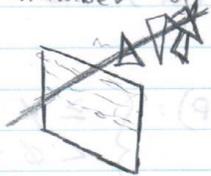
// for large objects

Z-Buffer (Catmull '74)

The Z-buffer (depth buffer)

is an image precision algorithm that requires 2 buffers

- $B_z$  that stores Z values
- $B_f$  (Frame) that stores color with same number of entries



compare z-values for each pixel

1.) Initialize

$B_z$  to  $\infty$  (or "largest" value of z-buffer represents z-value of front clipping plane)

- BF to BACKGROUND COLOR,  $\Delta + 58$
- 2.)  $\Delta$  are scan-converted in any order
  - 3.) Compare z-values and update the buffers.

```

For y = 0 to y = y_max //screen
  for x = 0 to x = x_max do //x_max screen
    begin //resolution (dense
      write BF(x, y, BACKGROUND COLOR) //coordinates
      write Bz(x, y, 0)
    end
  end

```

```

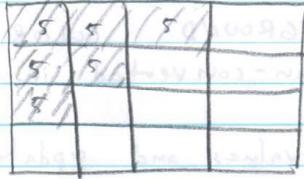
for each  $\Delta$  do
  for each pixel in  $\Delta$ 's (x, y) projection do
    begin
       $P_z = \Delta$ 's z-value at (x, y)
      IF  $P_z \geq B_z$  then
        new point is not further
        BF(x, y,  $\Delta$ -color)
        Bz(x, y,  $P_z$ )
      end
    end
  end

```

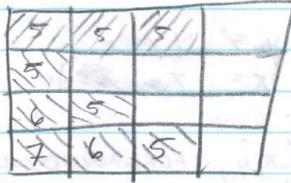
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

$\Delta_1$		$\Delta_2$	
5	5	5	
4	4		
3			
		5	
		4	5
		7	6
			5

$B_2 + A_1$



$+ A_2$



- + commonly implemented in hardware
- + fast  $O(n^2)$
- + doesn't require that objects be polygonal
- + somewhat independent of number of  $A$ 's
- + no object comparisons
- + only  $z$ -comparisons
- lots of memory required
- aliasing effects
- + any order of  $A$ 's

// BSP-preprocessing ; VR navigation  
// textile engineering - color theory

February 21, 2001

- // time & memory drawbacks of Z buffer
- // determine if the ray intersects with
- // triangle plane (div by zero error)



- // Draw vectors - if angles equal 360, then
- // the point is in the triangle

$$Ax + By + Cz + D = 0$$
$$R_0 + Dt = Ray$$

$$Ray_x = R_{0x} + RD_x t$$
$$Ray_y = R_{0y} + RD_y t$$
$$Ray_z = R_{0z} + RD_z t$$

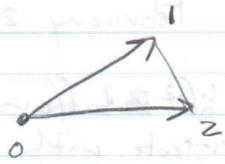
$$Ray = (R_{0x} + RD_x t, R_{0y} + RD_y t, R_{0z} + RD_z t)$$

$$A(R_{0x} + RD_x t) + B(R_{0y} + RD_y t) + C(R_{0z} + RD_z t) + D = 0$$

// solve for t

$$t = \frac{-(P_N \cdot R_0 + D)}{P_N \cdot RD}$$

$$t = \frac{-(AR_{0x} + BR_{0y} + CR_{0z} + D)}{AD_x + BD_y + CD_z}$$



- // D - distance between plane and origin
- // D can be negative
- // based on plane normal

// Ray Plane Intersection - search

- // if cross of signs are different,
- // then the point is not in the triangle

- // if  $\text{vect} \geq 0$ , then it's in triangle
- // really doesn't matter.

// D should be normalized

- //  $R_0$  is a point - don't normalize it

// RD

// screen =  $z = -1$

// for (45, 75) - eyepoint

// intersection function takes most time

February 26, 2001

// No class Wednesday → SWEET!!

- Binary Spanning Partition (BSP) Tree

FUCHS

// UNC

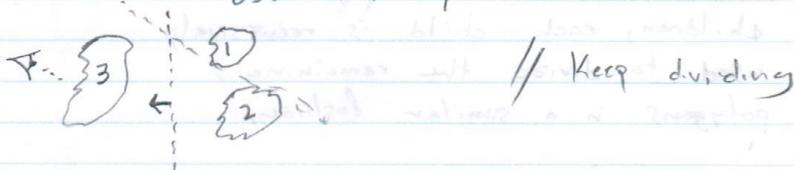
// Pixel Planes - massive graphics hardware system

// each pixel has a CPU

// draw back halfspace first - painter's algorithm

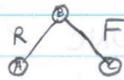
// (paint objects in the back first.

- The work is done in a pre-processing step prior to display.
- It's well suited for cases where viewpoint changes frequently but objects are static
- Main goal is to find clusters of objects ( $\Delta$ 's and planes that partition the object space  $\rightarrow$  clusters on the same side of a particular plane as the viewpoint can obscure, but not be obscured, by other clusters



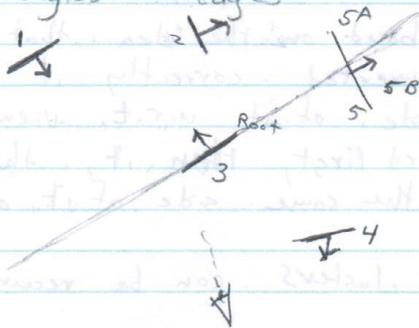
- BSP tree is based on the idea that a  $\Delta$  will be seen converted correctly if all  $\Delta$ 's on the other side of it w.r.t. viewpoint are seen converted first, then it, then all other  $\Delta$ 's on the same side of it as viewpoint
- Each of the clusters can be recursively

subdivided if other suitable planes  
can be found



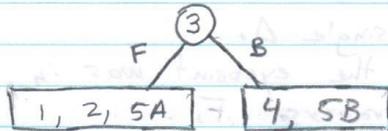
- choose a particular polygon as a root  
(any  $\Delta$ )
  - compute the plane equation that  
defines it + normal vector  $\vec{n}$
  - This partitions the scene into  
two half-spaces
    - a) half-space containing  $\Delta$ s  
in front of  $\Delta_{root}$   
(w.r.t.  $\vec{n}$ )
    - b) half-space containing  $\Delta$ s behind  
 $\Delta_{root}$
- ( $\Delta$ s spanning half-plane are split)
- One  $\Delta$  from each half-space as  
children, each child is recursively  
used to divide the remaining  
polygons in a similar fashion.

// triangles on edges

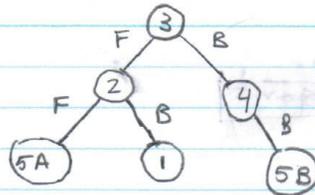


February 26, 2001

- Select a  $\Delta_{root}$ , say 3
- ⇒ eyepoint is in rear half-space



- Select the next child, say 2



then select 4

- Now, the BSP tree is constructed.
- Next, we traverse it.
- Traversal of BSP tree
- Traverse the branch descended first is the side of the tree away from viewpoint
- This is determined by substituting the viewpoint into the plane equation for  $\Delta_{root}$   
e.g. if eyepoint is in front half-space
  - scan-convert  $\Delta$ s in rear  $\frac{1}{2}$ -space
  - scan-convert root
  - scan-convert  $\Delta$ s in front  $\frac{1}{2}$ -space
- When there is no first branch to descend or that branch has been completed, then render the  $\Delta$  at this node.
- After the current node's  $\Delta$  has rendered

descend the branch that is closer to the viewpoint

- Algorithm terminates when each node contains a single  $\Delta$ .

• In the example, the viewpoint was in  $B \frac{1}{2}$  space  $\rightarrow$  traverse F.

$\rightarrow$  Go to node 2. Viewpoint is in the  $B - \frac{1}{2}$   $\rightarrow$  traverse F.

$\rightarrow$  5A

• Then 2, then 1,

$\rightarrow$  Then consider viewpoint

$\rightarrow$  Then root  $\Delta_R$

viewpoint w.r.t 4.

Viewpoint in F w.r.t. 4.

$\rightarrow$  Render 5B, then 4.

• Rendering order is

5A }  
2 } F  $\frac{1}{2}$ -space  
1 }

3 --- Root

5B }  
4 } B -  $\frac{1}{2}$  space

// select the one with the fewest branches,

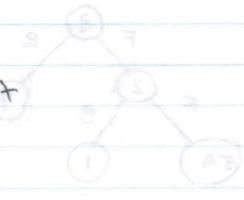
// to speed up the algorithm

// Developed after Z-buffer

// Depth-Sort - sort out the depth.

// order of polygons w.r.t. each

// other



February 26, 2001

// z-buffer, scanline z-buffer, BSP,  
// depth sort

March 12, 2001

// Think of light source as a point

### Illumination Models

(Lighting + Shading Models)

- The goal is to model how light interacts with objects, and to create "renderings" that look realistic
- A shading approach may use certain illumination models for some pixels, and other models for other pixels

Sources: Ray Optics and Thermal Radiation

// Ray Optics - light travels in straight lines

- Most models are really approximations,

There [are] 3 kinds of approaches

- Ⓐ Local Models: take into account interactions between light and a point on a surface // 95%
- Ⓑ Global Models: take into account the interactions between light and all surfaces.
  - RAY TRACING (Backward tracing of rays)
  - RADIOSITY (Heat exchange)

Ⓒ Hybrid // Uses both local and global models

### Local Illumination Models

(1) Light - source independent

- Depth cueing
- Depth Shading
- Ambient

(2) Light - source dependent

- Diffuse
- Specular

### Depth Shading

- Color or intensity of image

determined solely by depth of  $\Delta$

- Darker colors at lower elevations, for instance

- Effective in modeling terrain data.

Depth-Cueing

- Reduce "clarity" of pixel as a function of depth

Ambient Illumination

Ambient light is the illumination of an object caused by reflected, non-directed light emanating from other surfaces.

A simple model assumes ambient light is uniform in the scene

- Let  $I_a$  = ambient light intensity (same for all objects)

$k_a$  = reflection coefficient (for each object)

$$0 \leq k_a \leq 1$$

$I = I_a k_a \Rightarrow$  An object's own "reflectance" of background light.

- Light-Dependent Models.

What an object looks like depends on:

- Viewing orientation

- material properties of an object

- location + properties of light source

March 12, 2001

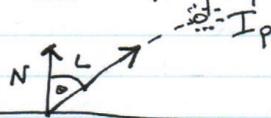


- reflection  $\left\{ \begin{array}{l} \text{diffuse // at an angle} \\ \text{specular // head-on} \end{array} \right.$
- absorption
- refraction (transmission)
- combinations

### Diffusion Reflection

Using Lambert's Law

- The Intensity of Light reflected from a surface is proportional to the cosine of the angle between the vector  $L$  to the light source and the normal to the surface vector  $N$  perpendicular to the surface.



- The amount of reflected light is dependent on position of light source, the orientation of surface, but independent of observer's position

$$I = I_p k_d \cos \theta$$

$$0 \leq \theta \leq \pi/2$$

$$0 \leq k_d \leq 1$$

$$I = I_p k_d \frac{\hat{N} \cdot \hat{L}}{\|\hat{N}\|}$$

- This gives dull, matte surfaces

March 14, 2001

$$I = I_p k_d (\hat{L} \cdot \hat{N})$$



- Diffuse model gives rise to "harsh" looking objects. (they contrast too much with background)
- This can be remedied by adding ambient light term:

$$I = I_a k_a + I_p k_d (\hat{L} \cdot \hat{N})$$

- What about attenuation as a function of distance?
- for instance, for surfaces that are farther from light source.

$$I = I_a k_a + I_p k_d \frac{(\hat{L} \cdot \hat{N})}{R^2} \quad \begin{array}{l} // R \text{ is distance from} \\ // \text{light source} \end{array}$$

- However, this results in too rapid a fall-off of intensity.

Instead of  $1/R^2$ , use

$$f_{att} = \frac{1}{c_1 + c_2 R + c_3 R^2}$$

- $c_1$ ,  $c_2$ , and  $c_3$  are determined empirically.
- So far, the model is monochromatic.
- What about R, G, + B?

⇒ Separate equations are written for each RGB

e.g.,

$$I_R = I_{aR} k_a O_{dR} + f_{att} I_{pR} k_d O_{dR} (\hat{N} \cdot \hat{L})$$

$O_{DR}$  is used for scaling  $\rightarrow$  the user can control the amount of reflection (ambient / diffuse) without altering objects coefficients,

• In general, for any  $\lambda$   
 $I_{\lambda} = I_{a\lambda} k_a + f_{att} I_{p\lambda} k_d (N \cdot L)$

$$\Rightarrow I = \sum_{\lambda} I_{\lambda}$$

This leads to the inclusion of transparency,

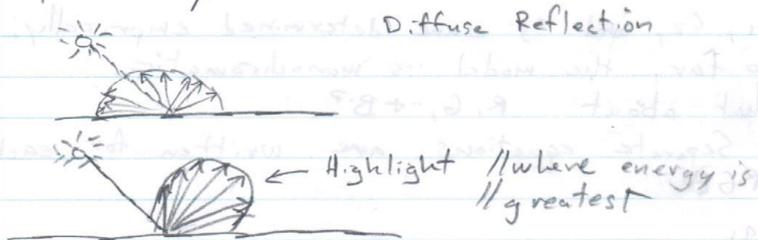
$$I = \sum_{\lambda} I_{\lambda} + f(a)$$

• Light Source Dependent Models:

Highlights and Specular Reflections.

• In general, surfaces are not perfectly reflective. This means that there are zones or areas of relatively higher degree of reflection than others.  $\Rightarrow$  The degree of reflection will depend on the viewpoint

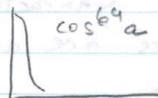
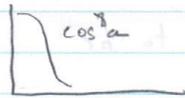
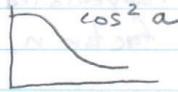
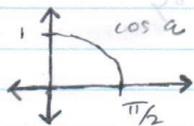
• In practice, reflection is not concentrated along a single ray, but has a specular zone and then falls off in intensity from specularly.

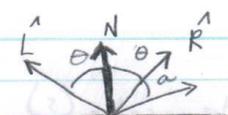


- In most reflection models, the color(s) of the highlight are those of the light source,
- While the color(s) of diffuse reflection are those of the object.



- Specular reflection can be modeled by considering  $\vec{L}$ ,  $\vec{N}$ , and  $\vec{V}$
- Following the model of Bui-Tuong Phong (1975) this behavior is approximated by a term of the form:
  - $k_s \cos^n \alpha$
  - $k_s$  = coefficient of specular reflection
  - $n$  = glossiness factor
- This is empirically derived,
  - $n \in (1, \infty)$
  - $k_s \in [0, 1]$
- This term expresses how much (source) light to add to the color of the object.
- For perfect reflector,  $n \rightarrow \infty$



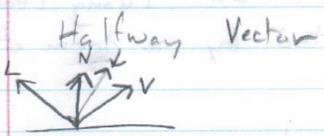


Phong Illumination Model:

$$I = I_a (k_a + f_{att} I_p) [k_d \cos \theta + k_s \cos^2 \alpha]$$

$$[k_d (L \cdot N) + k_s (R \cdot V)^n]$$

//  $k_d$  - diffusivity  
 //  $k_s$  - specular



• Rather than calculating  $R$ , the halfway vector is used:  $H$  is a unit vector normal to a hypothetical surface oriented halfway between  $L$  and  $V$ .

$$H = \frac{L + V}{\|L + V\|}$$

• Although  $\text{angle}(R, V) = \angle(N, H)$  this can be compensated for by the glossiness factor  $n$ .  
 $(N \cdot H)^n$

Phong Model Characteristics:

- Light sources are assumed to be point sources.
- In general, most of the geometry is

March 14, 2001

- ignored except for the surface normals,
- Diffuse + specular terms are modeled as local components
  - Highlights are modeled as a function of  $n$ .
  - Ambient light term is modeled as a constant.

// ambient, diffuse, specular - 3 components

March 19, 2001

Second Quiz - March 28

Assignment #3 - Posted Tonight - Due March 30

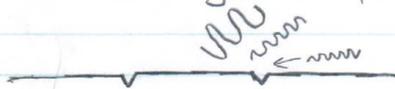
// Shading Project

// Extra Credit - use Open GL libraries

// Assignment #4 - possibility - 10 groups - pick a topic in graphics

In optics, light interactions depend on:

- wavelength(s)
- angle of incidence
- roughness of surface
- electromagnetic properties



In CG, surface roughness is simulated via surface detail effects (e.g. texture mapping),

EM properties are ignored.

$\lambda$  is approximated by RGB values.

- The interaction is approximated -  $\rightarrow$  there is an ambient component, diffuse component, and a specular component interpolated over the surface of  $\Delta$ 's.

// specular component - has highlight

// attenuation - fall-off of energy

### Incremental Shading

For  $\Delta$ 's, the information that is available are the vertices and  $N$ . Therefore, to obtain a shade for the entire polygonal surface,

the reflection model is applied at vertices and subsequently interpolations are made.

• Interior points are evaluated in scan-line order

(can be integrated into a Z-buffer algorithm.)

FLAT

scan line



GOURAUD

// interpolate normals

PHONG

Flat (constant, faceted) Shading

One calculation for entire  $\Delta$

Every interior pixel is assigned the same color.

+ Fast

+ Easy to implement

- Valid if light source is at  $\infty$

- Valid if observer is at  $\infty$

- faceted appearance.

Flat Shading ignores its neighbors

Gouraud Models light interactions

in meshes + takes advantage of the

info provided by neighbors

## Gouraud (Interpolative) Shading

1. Approximate the normal at a vertex by averaging all normals of abutting  $\Delta$ 's

$$\left\{ \begin{array}{l} x_1 \\ + \\ x_2 \\ \hline 2 \end{array} \right\} \begin{array}{l} y_1 \ z_1 \\ y_2 \ z_2 \end{array}$$

2. Calculate the intensity at each vertex using the selected illumination model.
3. Interpolate intensity along each  $\Delta$  edge.
4. Interpolate along a scan line to compute intensity at each interior point.

March 27, 2001

## • The Rendering Equation (Kajiya)

- Models light being transferred from one point ( $x'$ ) to another ( $x$ ) in terms of intensity of light emitted from first point ( $x'$ ) to second ( $x$ ) and the intensity of light emitted from all other points ( $x''$ ) that reach first point ( $x'$ ) and is reflected from  $x'$  to  $x$ .

- This is expressed recursively

$$I(x, x') = g(x, x') \left[ \varepsilon(x, x') + \int p(x, x', x'') dx'' \right]$$

$x, x', x''$  = points in the environment.

- $I(x, x')$  is related to intensity passing from  $x''$  to  $x'$

- $\varepsilon(x, x')$  is related to intensity that is emitted from  $x'$  to  $x$ .

- Initial evaluation of  $g(x, x')$  &  $\varepsilon(x, x')$  for  $x$  at some view point does USD.

- Integral is over all points on surfaces

$p(x, x', x'')$  is related to intensity of light reflected (includes specular & diffuse reflection) from  $x''$  to  $x$  from surface at  $x'$ .

$g(x, x')$  is a geometric form =

$$\begin{cases} \emptyset & \text{when } x \text{ \& } x' \text{ are occluded from each other} \\ 1/R^2 & \text{when } x \text{ \& } x' \text{ are visible to each other} \end{cases}$$

- the goal is to solve the Rendering Equation.
  - a) RAY TRACING
  - b) RADIOSSITY
  - c) combinations of (a) + (b)
- are solutions to the Rendering Equation.
- // Image-Based Rendering

### RAY TRACING

- RT is one of the most complete simulations of light interaction models + a possible solution of the Rendering Equation
  - The basic assumption is that an observer sees a point on a surface as a result of the interaction of the surface with rays emanating from elsewhere (everywhere else) in the scene (rather than local interaction).
  - RT was originally used by Appel (1968) as a solution to VSD, the 1<sup>st</sup> use of RT incorporating reflection, refraction + shadows developed by Kay (79) and Whitted (1980).
  - Global reflection + transmission replace local (diffuse + specular) reflection.
  - The simplest RT model is VSD
- Ray Tracing  
A CoP and a window selected  
(window is pixel plane)

For each pixel, an "eye" ray is cast through pixel (from CoP). The pixel's color is set to that of the object at closest point of intersection.

Select CoP and window

for each scan line in image do  
for each pixel in scan line do

begin

determine ray from CoP through pixel

for each object in scene do

if object is intersected and is  
closest considered thus far, then

record intersection and object name

set pixel's color to that of closest  
object intersection

end

This can be extended to include:

- ① Reflection
- ② Refraction
- ③ Shadowing
- ④ Transparency  
(non-refractive transmission)

Trace backwards from CoP to 1<sup>st</sup> intersection. At that intersection, calculate the total contribution to the pixel as a sum of 3 terms:

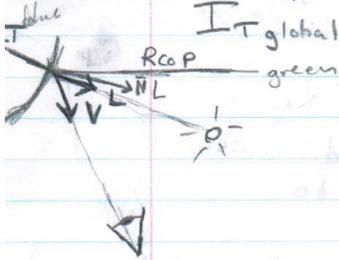
- 1) Local contribution  $I_{local}$  (ambient + diffuse + specular)  
Phong Illum model

2) Global reflection term

$I_{Rglobal}$  (recursively)

3) Global transmission

$I_{Tglobal}$



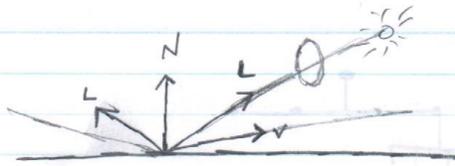
$$I = I_{local} + I_{Rglobal} + I_{Tglobal}$$

$$I_{local} = I_a k_a + I_p f_{att} \{ k_d (L \cdot N) + k_s (R \cdot V)^2 \}$$

$I_{Rglobal}$  (trace the "green" ray & calculate reflection for objects intersected)

$I_{Tglobal}$  similarly, trace transmitted ("blue") ray & find intersections.

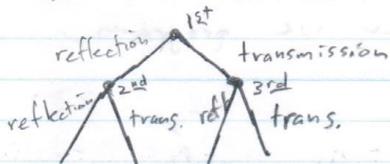
April 2, 2001



$$I = I_{\text{LOCAL}} + I_{R_{\text{GLOBAL}}} + I_{T_{\text{GLOBAL}}} \text{ // transmission}$$

To perform recursions, a ray-tree is constructed which represents the hierarchy of intersections:

a 1<sup>st</sup> intersection



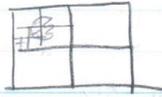
Each reflected / transmitted ray spawns other reflected / transmitted rays.

At each intersection, four quantities are evaluated:

- + shadow ray (to find intervening objects)
- +  $I_{\text{local}}$  (PHONG)
- +  $I$  (reflection) global
- +  $I$  (transmission) global

Tree Traversal is terminated by reaching a user-defined depth.

(NB) or calculate the relative contribution due to  $n$  recursions until contribution is below a pre-determined threshold.



octree

### Project 4

- it's up to us whether we program or not
- or text / diagrams

- Abstract - What our project will (planned) to do
- include what our project will do
- (2 paragraphs)

Tomorrow # 4:30

April 4, 2001

- In RT, hard part is to determine intersections with objects. Generally, a bounding volume - typically a sphere.
- Consider the case of a ray and a sphere (with center  $(a, b, c)$  and radius  $r$ ).

$$(x-a)^2 + (y-b)^2 + (z-c)^2 - r^2 = 0.$$

For a ray:

$$x = x_0 + t(x - x_0) \quad \Delta x = x - x_0$$

$$y = y_0 + t(y - y_0) \quad \Delta y = y - y_0$$

$$z = z_0 + t(z - z_0) \quad \Delta z = z - z_0$$

Substitute  $x, y, z$  into equation for sphere:

$$x^2 - 2xa + a^2 + y^2 - 2yb + b^2 + z^2 - 2zc + c^2 = r^2$$

$$\begin{aligned} & [(x_0 + t \Delta x)^2 - 2a(x_0 + t \Delta x) + a^2 + (y_0 + t \Delta y)^2 \\ & - 2(y_0 + t \Delta y)b + b^2 + (z_0 + t \Delta z)^2 \\ & - 2(z_0 + t \Delta z)c + c^2 = r^2 \end{aligned}$$

$$\begin{aligned} & (\Delta x^2 + \Delta y^2 + \Delta z^2)t^2 \\ & + 2t [\Delta x(x_0 - a) + \Delta y(y_0 - b) + \Delta z(z_0 - c)] \\ & + (x_0 - a)^2 + (y_0 - b)^2 + (z_0 - c)^2 - r^2 \end{aligned}$$

Quadratic in  $t \Rightarrow$  use the quadratic formula

No real roots  $\Rightarrow$  no intersection

One root  $\Rightarrow$  ray grazes sphere

Two roots  $\Rightarrow$  ray pierces the sphere  
take smallest (closest to viewer)

Next, get surface normal at point of intersection,

$$\frac{x-a}{r} \quad \frac{y-b}{r} \quad \frac{z-c}{r}$$

at  $(x, y, z)$

Next find intersection with a  $\Delta$ .

First, find the plane containing  $\Delta$ , then determine whether ray intersects  $\Delta$ .

$$Ax + By + Cz + D = 0$$

Using  $x = x_0 + t\Delta x$  ... etc ( $y + z$  also)

$$A(x_0 + t\Delta x) + B(y_0 + t\Delta y) + C(z_0 + t\Delta z) + D = 0$$

$$t = \frac{-Ax_0 - By_0 - Cz_0 - D}{A\Delta x + B\Delta y + C\Delta z}$$

If denominator = 0  $\Rightarrow$  Ray + Plane are ||



Send out a ray (on the plane):

If odd # intersections  $\Rightarrow$  in

If even # intersections  $\Rightarrow$  outside

This is one of several ways of determining intersection with  $\Delta$ 's.

(Reminder)

At intersection point:

- 1.) Send out a shadow ray,  
(to light source)

If  $\exists$  intervening object:

- use background color
- use a fraction of object's color
- continue tracing of ray.

- 2.) Calculate  $I_{LOCAL}$  (Phong Model)

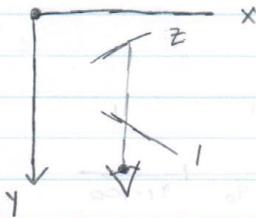
- 3.) Calculate  $I_{GLOBAL}$  (reflection)

- 4.) Calculate  $I_{GLOBAL}$  (transmission)

To include transparency:

- a) non-refracted transmission
- b) refractive transmission

For a): light is not redirected in space  
 $\Rightarrow$  not physically realistic but visually effective



Simple model is called interpolated transparency which determines color of the pixel along the ray as:

$$I = (1 - k_1) I_1 + k_2 I_2$$

Where  $0 \leq k \leq 1$  is "transmission coeff" associated with the surfaces, and  $(1 - k)$  is the opacity of the surface.

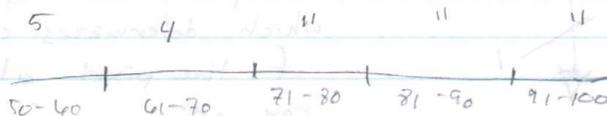
For a more realistic effect, use ambient + diffuse components of 1 with full color of 2 using this equation, then add specular component

• One solution is the "screen door" transparency technique, where only some of the pixels associated with a transparent object are used, resulting in adjacent pixels acquiring the color of different objects.

• The low-order bits of a pixel's (x,y) address are used to index into a bit mask. If the indexed bit

$\left\{ \begin{array}{l} 1 \Rightarrow \text{object color is used} \\ \%w \Rightarrow \text{object color is suppressed + next closest } \Delta \text{ color is used} \end{array} \right.$

\* all emails are posted



### Buffer Initialization

I) For  $y=0$  to  $y=\max$   
for  $x=0$  to  $x=\max$

begin

set  $F_B(x, y, \text{BACKGROUND\_COLOR})$

set  $F_G(x, y, 0)$

end

II) For each polygon

For each pixel that projects at  $x, y$

calculate  $P_z$  ( $z$  value)

If  $P_z \geq B_z$  at  $x, y$

set  $B_z(x, y, P_z)$

set  $B_F(x, y,$

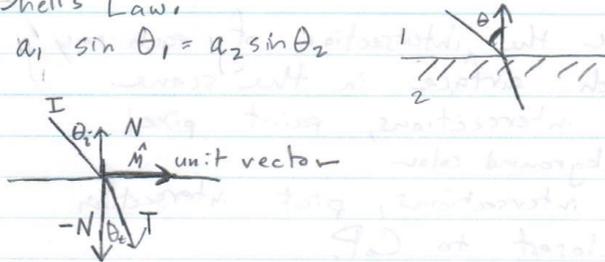
end

$$2) I_R = I_{AR} k_{AR} + f_{ATT} I_{PR} [k_{DR} (\hat{N} \cdot \hat{L}) + k_{SR} (R \cdot \hat{V})^n]$$

April 9, 2001

// Finish Ray Tracing  
RT: refractive transparency  
refractive index depends on wavelength,  
Snell's Law:

$$n_1 \sin \theta_1 = n_2 \sin \theta_2$$



$T = \sin \theta_t \hat{M} - \cos \theta_t \hat{N}$  in terms of  $\hat{I}$  and  $\hat{N}$

$$\Rightarrow \vec{T} = \alpha \vec{I} + \beta \vec{N}$$

$$\alpha = \frac{n_i}{n_t} \quad // \text{ratio of indices of refraction}$$

$$\beta = \left( \frac{n_i}{n_t} \right) (\hat{N} \cdot \hat{I}) - \left[ 1 - \left( \frac{n_i}{n_t} \right)^2 \{ 1 - (\hat{N} \cdot \hat{I})^2 \} \right]^{1/2}$$

$$n = \frac{\text{speed of light in vacuum}}{\text{speed of light in medium}}$$

Water: 1.33

glass: 1.52

air: 1.0003

$n_i$  - refractive index

// Best approximation of light going from  
// one medium to another

## RT:

1. Cast a ray from CoP // viewpoint  
through a pixel, for all  
pixels

2. Compute the intersection of each ray  
with each surface in the scene.

If no intersections, paint pixel  
the background color

If  $N$  intersections, pick intersection  
point closest to CoP.

3. At each intersection, calculate

a) shadow ray

b) local  $I_{\text{local}}^{(\text{refl})}$

c)  $I_{\text{global}}^{(\text{refl})}$

d)  $I_{\text{global}}^{(\text{trans})} \approx k_T$

4. Determine termination criteria.

a) define number of recursions

$$N \leq 7$$

b) rays leave the scene w/out  
intersection

c) use an adaptive depth control

• for secondary interactions

consider

$$I + C_1 R + C_2 R^2$$

attenuation

as well as refraction coefficients:

$$k_1 \cdot k_2 \cdot \dots \cdot k_n \leq \epsilon$$

April 9, 2001

RT "signatures"

- spherical objects
- very sharp surfaces.
- infinitely thin beams,  
(may be overcome with multiple rays.)

⇒ Photo realistic Images

// Radiosity

Whitted's algorithm can be written as:

$$I_{\lambda} = I_{a\lambda} k_a + \sum_{1 \leq i \leq m} S_i f_{att_i} [I_{p_i} [k_d (\hat{N} \cdot \hat{L}) + k_s (\hat{N} \cdot \hat{H}_i)^n] + k_r I_{r\lambda} + k_t I_{t\lambda}]$$

//local  
//halfway vector  
//global

This is a solution to kajiya's rendering equation integrating:

- Perspective projection
- VSD
- shadowing
- reflection
- transmission

Assign #4

Grading Criteria

	# points
1. Effectiveness of Tutorial //can understand it	//how well one (0-40)
2. Level of Effort	(0-30)
3. Originality <u>OR</u> Revelance	(0-20)
4. In class presentation	(0-10)



April 11, 2001

## ANIMATION

Literally, it means to bring to life,

In CG,  $\exists$  several types:

- Motion Dynamics: time-varying position of objects
- Update Dynamics: changes in shape, color, transparency, size, texture & structure.
- Environmental changes: lighting, camera position, orientation, focus, and rendering technique.

Animation is used widely in videogames, entertainment, education, advertising, et cetera.

### • Conventional Animation

Has a well-defined sequence.

- A story is written.
- A storyboard (high-level sequence of sketches)
- A soundtrack is recorded.
- Detailed layout is produced (with every scene)
- Key frames are produced (defining extreme or characteristic positions / situations).
- Intermediate frames are filled in (inbetweening)
- Trial film ("pencil test") is done
- Transfer to cels. (acetate).
- cels  $\rightarrow$  film

• Many steps in conventional animation can exploit the computer-based methods.

• In computer-based animation, there's no predetermined sequence. Typically, start out with storyboard, although some of the steps are highly interactive.

Lumiere - // first movie 1900's

Daguerre - // first silver screen

### - Keyframe Animation

- Inbetweening
- Interpolation
- Quaternions

### - Constrained-Based Control

- Geometric constraints  
\*floors + walls\*
- Linkages

### - Dynamic Control (Tracking Live Action)

- Rotoscoping
- Indicators/Reflectors
- Data glove, VR environments

### - Physically-Based or Model-Based Animation

- Kinematics  
\*friction, gravity, etc\*
- Dynamics (\*equations\*)

### - Procedural Control

- Actors (\*object oriented\*)
- Particle Systems

- Choreography
  - Control decor, cameras, etc. /\*post processing\*/
- Aliasing
- Reactive Control. /\* AI \*/

Keyframe animation is typically used for rigid bodies. Positions (and perhaps orientation) of objects are initially specified at key frames and the system derives in-between frames:

Inbetweening can be done a number of ways:

- linear interpolation (lerping)
- non-linear interpolation
- hybrid techniques

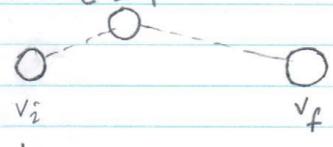
Given some attribute

$V$  (eg, color, size, position, ...),

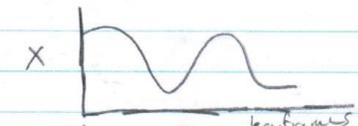
the goal is to find  $V$  in between initial and final states.

$$V_t = V_i (1-t) + t V_f$$

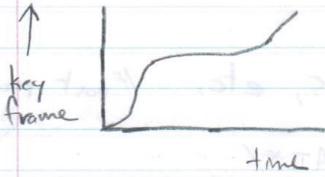
$$0 \leq t \leq 1$$



// can add more keyframes



Double-Interpolant method



• Given these  
 // spline function used to interpolate

10,000 polygons



Trigonometric method

April 16, 2001

- Rigid Body Interpolation
- Euler Angles + Quaternions
- Euler angles allow orientation to be described by specifying rotations w.r.t. 3 principal axes. (roll, pitch, + yaw).
- However, twisting can also occur  $\Rightarrow$  Quaternions
- Based on Euler's theorem stating that any displacement of a rigid body with a single point fixed can be represented by a single rotation about some axis.
- Quaternions are four component algebraic representations:

$$a + b\bar{i} + c\bar{j} + d\bar{k}$$

$\bar{i}$ ,  $\bar{j}$ , and  $\bar{k}$  are basis vectors for 3D space

(a b c d) are real and  $a^2 + b^2 + c^2 + d^2 = 1$   
key-framing can be used to interpolate parameters (a b c d) of the rigid-body transform

- To perform quaternion arithmetic, group components into real part (scalar) and an imaginary part (vector).
- Multiplication is a combination of cross-product + complex multiplication.

Given 2 quaternions:

$$(S_1, \bar{V}_1) + (S_2, \bar{V}_2)$$

$$(S_1, \bar{V}_1) (S_2, \bar{V}_2) = \left[ \underbrace{S_1 S_2 - (\bar{V}_1 \cdot \bar{V}_2)}_{\text{scalar}}, \right]$$

$$\left[ \underbrace{S_1 \bar{V}_2 + S_2 \bar{V}_1 + \bar{V}_1 \times \bar{V}_2}_{\text{vector}} \right]$$

To rotate a vector  $[0, \mathbf{v}]$  by a quaternion  $q$ , the rotated

vector  $\mathbf{v}'$  is

$$\mathbf{v}' = q^{-1} \mathbf{v} q \quad \text{when } q^{-1} = \left( \frac{S_1 - \mathbf{V}}{S^2} \right) + \mathbf{V} \cdot \mathbf{V}$$

// Masses, velocities, + accelerations: kinematics  
 // (trajectories)

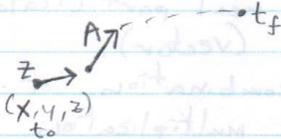
// i.e.  $F = ma$  - dynamics

// <sup>realism</sup> visual, physical, interaction, collaborative  
 // (conceptual)

// <sup>senses</sup> temperature, orientation,

Kinematics: positions and velocities of objects

"Object  $Z$  at  $(x, y, z)$  with velocity  $\mathbf{V}$  at time  $t_0$  and acceleration  $\mathbf{A}$ "



Inverse kinematics: To reach a final position  $(x', y', z')$  at  $t_f$ , what must  $\mathbf{V}$  and  $(x, y, z)$  have been?

Dynamics - take into account physical laws (Newton's laws, ...):

"Object  $\vec{z}$  at  $(x, y, z)$  with mass  $M$  and experiencing force  $F$ "

$\Rightarrow$  Equations of motion ( $\vec{F} = m\vec{a}$ )

Inverse Dynamics: "What force(s) must have been applied to reach final state given initial state?"

~~\*/ Interpolate Rigid bodies // - use quaternions deformed body during animation: \*/~~

23 April:

- Presentations

- Assignment #4 due by 4 PM

25 April:

- Quiz #3

27 April:

- Grade each other's assignments

April 18, 2001

## Particle Systems

This is an approach to simulate behavior over time for objects that are difficult to model polygonally.

Ex: Trees, grass, fire, smoke, ...

A PS is defined by a set of particles, each of which is created, transformed, and annihilated over a period of time.

The transformation process is defined by rules and/or probabilistic framework.

Individual particles will change:

- color

- size

- transparency

• Both global & individual behavior can be controlled by rules.

• A starting point is to determine the number of particles created at a particular time,

$N(t)$

// number of particles at time,  $t$

$$N(t) = N(\text{mean}) \pm \text{Random } V$$

$N(\text{mean}) = \text{mean \# particles}$

Random = RN function

$V = \text{variance of the particle population}$

Individual particles have a number of attributes:

• Initial Position

• Initial Velocity Vector

• Initial Size

• Initial Color

- Initial Transparency
- Shape
- Lifetime

-  $\exists$  two types of rules:

- Creation
- Transformation

### Creation Parameters

- A volume for creating particles
- A distribution of position
- The mean initial velocity & its variance
- $N(t)$  (number of particles created)

For each transformation parameter,  
define mean value and a variance.

$$X (\text{mean}) \pm V (\text{variance}) \cdot RN$$

For annihilation a particle, it is  
possible to define

- a max # of frames
- a max # of a parameter
- explicitly incorporate a "lifetime" factor
- the factor =  $\emptyset$   
(+ multiplies the set of parameters)

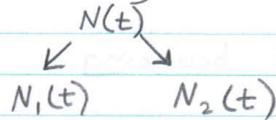
• It is also possible to constrain particle  
movement within some subspace  
(e.g. to a cylinder, or a trajectory)

in general to geometrical constraints

- In addition, a particle can spawn other  
particles leading to a hierarchy

April 18, 2001

or multi-generation PS,



One can create global effects by defining regions of influence in which some parameters change in a specific manner (e.g. adding a larger component to  $V_x$  to simulate the effects of forces).



William Reeves (85) // explosion of a star

$$X(t) = X_{\text{mean}} \pm V \cdot RN$$

© No Comprehensive Exam

Rendering How much light be bouncing

Equation -

Shadow Ray

$I_{local}$  -

$I_{global}$  - figure out where else the ray goes

$I_{transmission}$  -

Ways of interpolation for animation

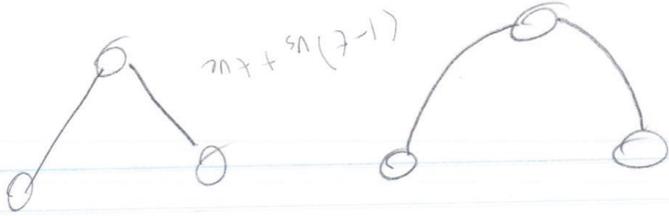
- linear, non-linear, hybrid techniques
- try to find out where it was

April 23, 2001

• Grade Ourselves -

---

9 Groups - 9



$$I(x, x') = g(x, x') \left[ \epsilon(x, x') + \int_p(x, x', x'') I(x', x'') dx'' \right]$$

Kajiya

$$I(x, x') = g(x, x') \left[ \epsilon(x, x') + \int_p(x, x', x'') I(x', x'') dx'' \right]$$

$$I(x, x') = g(x, x') \left[ \epsilon(x, x') + \int_p(x, x', x'') I(x', x'') dx'' \right]$$

$$I(x, x') = g(x, x') \left[ \epsilon(x, x') + \int_p(x, x', x'') I(x', x'') dx'' \right] \text{ // Rendering Eq.}$$

for each scanline in image do  
 for each pixel in scanline do  
 determine ray from CoP to pixel  
 for each object in scene  
 if ray intersects object and  
 object is closest seen thus far,  
 record object name and intersection  
 set pixel color to that of closest  
 intersection  
 end

$I_{\text{local}}$  // Phong Illumination Model  
 $I_{\text{global}}$  Reflectance  
 $I_{\text{global}}$  Transmittance  
 Shadow Ray

Ray Tree  $\rightarrow$  Recursions  
 RT - Hard to determine intersections  $\rightarrow$  use  
 bounding volume (sphere)

$$X(t) = X_{\text{mean}} \pm \text{Variance} \cdot \text{Random Number}$$

Whitted's algorithm

$$I_{\lambda} = I_{\lambda} k_a + \sum_{1 \leq i \leq m} S_i f_{att_i} I_{p_{\lambda i}} [k_d (\hat{N} \cdot L^{\wedge}) + k_s (\hat{N} \cdot H^{\wedge})^n] + k_r I_{r_{\lambda}} + k_t I_{t_{\lambda}}$$

$$I_{\lambda} = I_{\lambda} k_a + \sum_{1 \leq i \leq m} S_i f_{att_i} I_{p_{\lambda i}} [k_d (\hat{N} \cdot L^{\wedge}) + k_s (\hat{N} \cdot H^{\wedge})^n] + k_r I_{r_{\lambda}} + k_t I_{t_{\lambda}}$$

kittys run into red green knives  
to mifate the genocide

$$k_r I_{r_{\lambda}} + k_t I_{t_{\lambda}}$$

Whitted's

$$I_{\lambda} = I_{\lambda} k_a + \sum_{1 \leq i \leq m} S_i f_{att_i} I_{p_{\lambda i}} [k_d (\hat{N} \cdot L^{\wedge}) + k_s (\hat{N} \cdot H^{\wedge})^n] + k_r I_{r_{\lambda}} + k_t I_{t_{\lambda}}$$

$$k_r I_{r_{\lambda}} + k_t I_{t_{\lambda}}$$

### Gouraud

- 1) Approximate the normal at a vertex, by averaging the normals of all adjacent  $\Delta$ 's
- 2) Calculate Intensity at each vertex
- 3) Interpolate across  $\Delta$  edges
- 4) Interpolate across scan lines

$$I = I_a k_a + I_p f_{att} [k_d \cos \theta + k_s \cos^n \alpha]$$

Phong Illumination

$$I = I_a k_a + I_p f_{att} [k_d \cos \theta + k_s \cos^n \alpha]$$

$$(x-a)^2 + (y-b)^2 + (z-c)^2 - r^2 = 0$$

$$\begin{aligned} x &= x_0 + t(x-x_0) \\ y &= y_0 + t(y-y_0) \\ z &= z_0 + t(z-z_0) \end{aligned}$$

3/27 Rendering Equation

4/2 Ray Trees

4/4 RT Object Interactions

4/9 Refractive Transparency

4/11 Animation

4/16 Rigid Body Interpolation / Euler Angles

4/18 Particle Systems

Interpolation - Linear / Non Linear / Hybrid

Key Framing - Inbetweening, Interpolation, Quaternions

Constrained Based - Geometric Boundaries, Linkages

Physically Based - Kinematics, Dynamics

Choreography -

Aliasing -

Reactive Control - AI

Procedural Control - Actors, Particle Systems

Radiosity

Position, Velocity  
Size  
Color  
Shape  
Transparency  
Lifetime

$$B_i = E_i + p_i \sum_{1 \leq j \leq n} B_j F_{i \rightarrow j}$$

Refraction - Snell's Law

$$n_1 \sin \theta_1 = n_2 \sin \theta_2$$

Radiosity

Form Factors, Substructuring, Progressive Refinement

$$B_i = E_i + p_i \sum_{1 \leq j \leq n} B_j F_{i \rightarrow j}$$

Animation

Keyframing - Inbetweening, Interpolation, Quaternions  
Constrained Based - Geometrically constrained Linkages

Physically Based, Model Based - Kinematics, Dynamics

Choreography, Aliasing, Procedural Control - Particle Systems, Actors

Reactive Control - AI

Whitted's Algorithm

May '79

$$I_{\lambda} = I_{\lambda} k_{\lambda} + \sum_{1 \leq i \leq m} S_i f_{\lambda i} I_{p_i}$$

$$[k_d (\dot{N} - \dot{L}) + k_s (N \cdot H)^n] k_r I_{\lambda} + k_t I_{\lambda}$$

Kajiya's Rendering Equation

$$I(x, x') = g(x, x') + [E(x, x') + \int_{\Omega} p(x, x', x'') I(x', x'') dx'']$$

Radiosity

$$B_i = E_i + \rho \sum_{1 \leq j \leq n} B_j F_{j-i} \frac{A_j}{A_i}$$

### Whitted's Algorithm

$$I_{\lambda} = I_{a, \lambda} + \sum_{1 \leq i \leq m} S_i + f_{att, i} + I_{p, i} [k_d (\hat{N} \cdot \hat{L}) + k_s (\hat{N} \cdot \hat{H}_i)^n] k_r I_{r, \lambda} + k_t I_{t, \lambda}$$

### Rajiv's Rendering Equation

$$I(x, x') = g(x, x') + \left[ \epsilon(x, x') + \int_{\Omega} \rho(x, x', x'') I(x', x'') dx'' \right]$$

## Project Conclusion

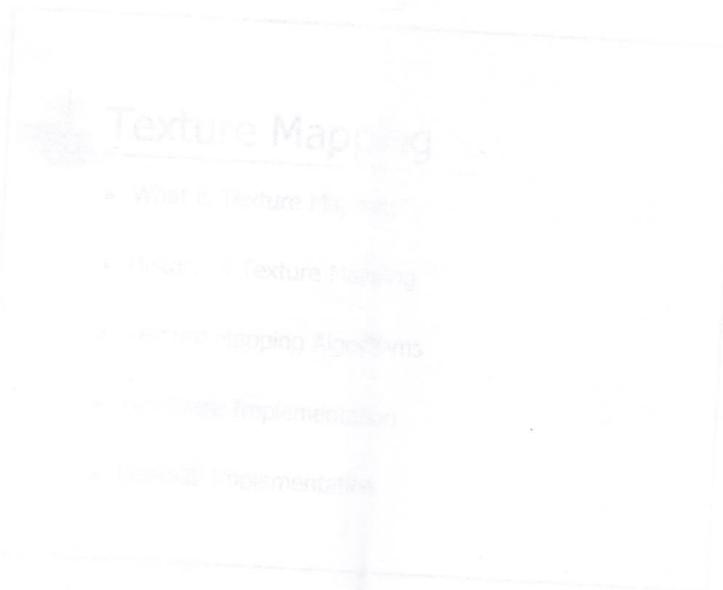
The project that we have created covers the topic of texture mapping.

After our study of shading models, we felt that texture mapping is the next logical step in the study of polygon detail. Texture mapping is a significant topic in computer graphics, and we thought that it should be covered in our class in the form of our project.

In our project, we have created a basic tutorial so that novices to the subject will be able to read and understand many of the concepts presented in texture mapping. We have created slides that can be viewed over the Internet and a simple texture mapping demo. Our research focused on the areas of history, algorithms, hardware and code implementation of texture mapping. Source code is provided with the demo so that people who are interesting in learning the actual software implementation of texture mapping can have a resource to aid them. While the texture mapping demo uses preexisting OpenGL code as a framework, all drawing routines were written completely by our team's developer. All research done by our team used resources that are noted at the end of the tutorial.

Our team has faced a few problems during the production of our project. Since all team members have differing schedules, finding an appropriate time when everyone could meet was sometimes a complicated task. Fortunately, the use of an on-line system, which keeps track of important dates, relative web links, and our completed documents, allowed us to effectively share our work with other members. Our developer also faced challenges porting our project code between system platforms for our presentation.

The research and work by our team has broadened our knowledge on the topic of texture mapping. Hopefully, others will be inspired by our presentation to further investigate this topic.



Group #

What is Texture Mapping?

## Texture Mapping Tutorial

CS4451 Group 4  
Project 4

Carpenter, Gelsomini, Huynh, Lee, and Smith

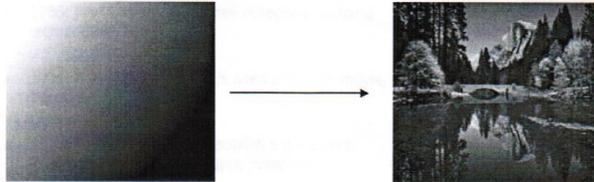
Clad, Anthony, Hoang, Albert, + myself

## Texture Mapping History

- What is Texture Mapping?
- History of Texture Mapping
- Texture Mapping Algorithms
- Hardware Implementation
- OpenGL Implementation

## What is Texture Mapping?

- Texture mapping is a technique that allows a texture (image) to be painted onto the surface of a polygon.
- This results in a more detailed polygon surface.
- This technique can help reduce the number of polygons used in objects.



## Texture Mapping History

- 1974: Texture mapping was first described in Ed Catmull's Ph.D dissertation. Catmull texture mapped a mickey mouse image onto a curved surface patch.
- 1976: Jim Blinn created the first environment mapped object, the Utah Teapot.
- 1982 - 83: The team of Gene Miller and Ken Perlin and the team of Michael Chon and Lance Williams independently created the first reflection mapped objects.
- 1983: The MIP Mapping technique, which is a method for avoiding aliasing in texture mapping algorithms, is introduced in Lance William's SIGGRAPH paper entitled "Pyramidal Parametrics".
- 1984: Miller and Hoffman made great advancements in computer graphics animation.



• A texture is made from the environment and then mapped onto the object

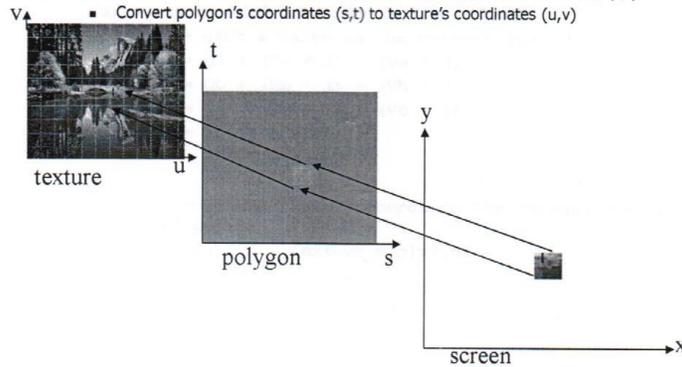
## Texture Mapping History (continued)

- 1985: At the New York Institute of Technology, reflection mapping is used in a moving scene with animated CG
- 1986: The first movie to use reflection mapping, Flight of the Navigator, is released
- 1986: Ned Greene formalizes reflection mapping techniques
- 1991: Reflection mapping is used in the hit movie, Terminator 2
- 1993: Haeberli and Segal publish a document illustrating the uses of texture mapping



## Basic Algorithm Overview

- Convert pixel from screen coordinates  $(x,y)$  to polygon's coordinates  $(s,t)$
- Convert polygon's coordinates  $(s,t)$  to texture's coordinates  $(u,v)$



## A Texture Mapping Algorithm

Given: a 256x256 texture map with horizontal and vertical axes  $u$  and  $v$ , respectively

First, we define some vectors for the texture map:

Let vector  $P$  be the texture origin (usually the top left)

Let  $M$  and  $N$  be vectors for the  $u$  and  $v$  axes, respectively

Now, we have the equations:

We compute 9 numbers which remain constant while texture mapping the polygon.  
 $O$  stands for Origin,  $H$  for Horizontal, and  $V$  for vertical.

$$\begin{aligned}O_a &= N_x * P_z - N_z * P_x \\H_a &= N_z * P_y - N_y * P_z \\V_a &= N_y * P_x - N_x * P_y\end{aligned}$$

$$\begin{aligned}O_b &= M_x * P_z - M_z * P_x \\H_b &= M_z * P_y - M_y * P_z \\V_b &= M_y * P_x - M_x * P_y\end{aligned}$$

$$\begin{aligned}O_c &= M_z * N_x - M_x * N_z \\H_c &= M_y * N_z - M_z * N_y \\V_c &= M_x * N_y - M_y * N_x\end{aligned}$$

- This algorithm is not the fastest, but it is the most reliable.
- Other algorithms are available.

## Algorithm Pseudocode

```
for every j which is a row in the polygon
  for i = each x value in the current row
    a = Oa + (Ha * i) + (Va * j)
    b = Ob + (Hb * i) + (Vb * j)
    c = Oc + (Hc * i) + (Vc * j)
    u = 256 * a / c
    v = 256 * b / c

    //copy the texture coords to the current screen
    // coords
    screen[i] = texture_map[v][u]
  endfor
endfor
```

# Hardware Implementation

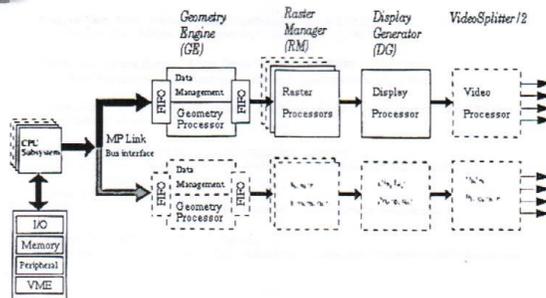
Q: Why?

A: Hardware provides cost-effective real-time computer graphics capacities

Hardware implementation:

- Design a memory system for texture mapping
- Support real-time texture mapping
- *Hardware can use*  
Using multiple fragment generators for texture mapping fragments
- Data in the graphics pipeline is processed in stages, the process begins in the CPU, where the application sends data to the graphics subsystem across the bus to the graphics pipeline
- Graphics pipeline: geometry processing, fragment generation, hidden surface removal, and frame-buffer display

# Hardware (continued)



▪ Reality Engine System Architecture Showing Single and Dual Pipeline Configurations

- Data on bus to pipeline.
- Data is converted to pixel data.
- Pixels are processed
- Optional - optimises data capacity

## OpenGL Implementation

- How do you texture map a polygon in OpenGL?
  - First you need an image to use as a texture.
  - Must be a square power of 2, eg; 64x64, 128x128, 256x256.
  - Use `glGenTexture()` to get texture maps into program, shown in demo code.
- Once you have textures in OpenGL, the rest is easy
  - To texture map a quad all that you need is to match the corners of the texture to the corners of the quad.
  - At the beginning of each quad that will use a different texture, call `glBindTexture()` to tell OpenGL to use that texture.
  - Each `glVertex3f()` call is preceded by a `glTexCoord2f()` call which specifies which corner of the texture to map to this vertex.

Example:

```
glTexCoord2f( float x, float y );  
glVertex3f( xcoord, ycoord, zcoord );
```

## sources

- Foley, van Dam, Feiner, Hughes: Computer Graphics: Principles and Practice  
Reading, Mass. Addison-Wesley Publishing Company: 1997. Pg.741-744.
- Barrett, Sean. Texture Mapping. N. pag. Online. Internet. April 20 2001.  
Available [http://www.toise.com/en/computers/tech\\_specs\\_graphics/texture.html](http://www.toise.com/en/computers/tech_specs_graphics/texture.html)
- Hakura, Ziyad S. and Gupta, Anoop. The Design and Analysis of a Cache Architecture for Texture Mapping.  
Stanford, California.
- Mapleson, Ian. Reality Engine in Visual Simulation Technical Overview.  
N. pag. Online. Internet. April 20 2001. Available <http://www.futuretech.yourwerk.nl/re.html>
- Molofee, Jeff. OpenGL Windows Tutorials.  
N. pag. Online. Internet April 20 2001. Available <http://nehe.gamedev.net/opengl.asp>
- Debevec, Paul. The Story of Reflecion Mapping.  
N. pag. Online. Internet April 20 2001. Available <http://graphics3.isi.edu/~debevec/ReflectionMapping/>

Smith, Levi D.  
NAME (Print last name first)

February 14, 2001

83/100

2001

SPRING  
CS 4451 ~~WINTER SEMESTER 2000:~~

QUIZ

*Relax..*

This is a closed-book quizz consisting of 3 (three) problems or questions with the point distribution given below:

#1: 15 Points  
#2: 35 Points

#4: 50 Points  
Total: 100 Points

PRINT your last name and first-name initial in the space provided on the this page.

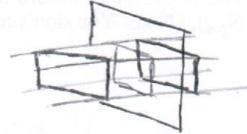
Lev D. Smith

13

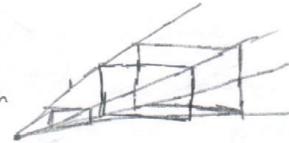
1. (15 Points)

(a) What are the two basic types of planar geometric projections, and what is the major difference between them?

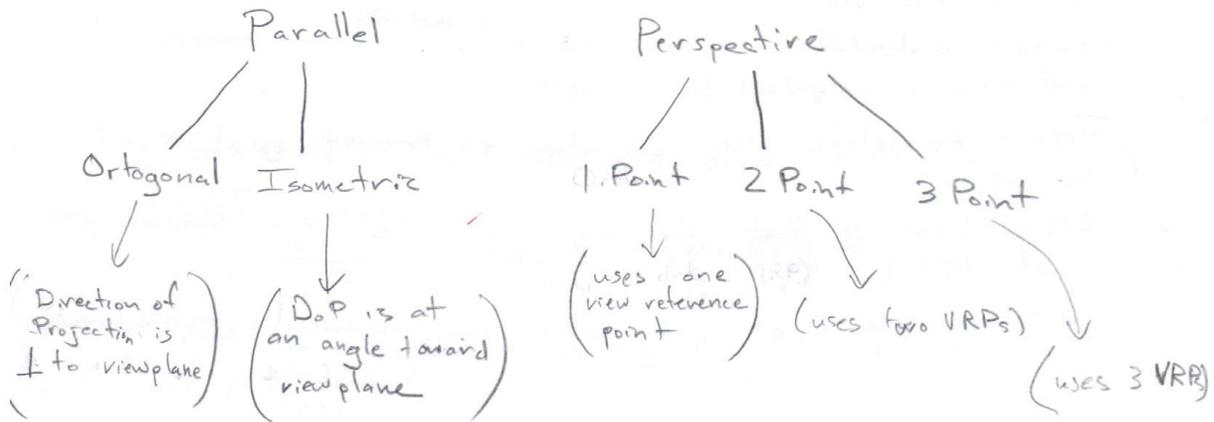
Parallel - Uses the direction of projection to define the view volume, *CoP at  $\infty$*



Perspective - Uses a center of projection to define the view volume, *CoP at finite distance*



(b) Under these two types of projections, there are several possible projection subclasses. Draw and label a hierarchical tree diagram that summarizes the subclasses of planar geometric projections.



35  
2. (35 Points)

(a) Write down an expression for  $N_{\text{per}}$ , the normalizing transformation for perspective projections. The expression should show the correct order of steps (i.e., the transformations that compose  $N_{\text{per}}$ ). (Note: You don't need to write actual matrix expressions for the individual transformations.)

$S SH T_{\text{PRP}} R T_{\text{VRP}} N_{\text{per}}$   
(using reverse order notation)

S = scale  
SH = shear  
T = translate  
R = rotate  
T = translate

(b) Briefly explain what each step is, and the reason for introducing each step.

$T_{\text{VRP}}$  - translate the view reference point to the origin.

R - rotate view reference coordinates such that u-axis is aligned with the z-axis, v-axis is aligned with the x-axis, and w-axis is aligned with the y-axis.

$T_{\text{PRP}}$  - translate the projection reference point to the origin.

SH - shear so that the angles of the view volume are at  $45^\circ$ .

S - scale so that you obtain the canonical view volume.

(c) What is the overall objective of performing this transformation?

To create the canonical view volume, which makes the clipping process much easier.

35

Levi D. Smith

3. (50 Points)

(a) What is the sequence of steps necessary to implement the overall 3D viewing transformation, beginning with a set of primitives defined in 3D space and ending with a 2D image displayed on part of the CRT screen?

- Apply perspective transformation
- Convert points to homogeneous coordinates

### NORMALIZING TRANSFORMATION

- Clip everything outside of the view volume using a clipping algorithm, like Cohen-Sutherland which uses a 6-bit code to compute which lines should be clipped.
- Convert back to view coordinates.

- Scale such that the view port fills the

2

$$S\left(\frac{u_{\max} - u_{\min}}{2}, \frac{v_{\max} - v_{\min}}{2}, \frac{z_{\max} - z_{\min}}{1}\right)$$

- Translate the view volume by  $(1, 1, 1)$

2

$$T(1, 1, 1)$$

7

- 2D image is obtained by using only the x and y coordinates of the points.

Smith, Levi D.  
NAME (Print LAST name first)

55

## CS 4451 SPRING SEMESTER 2001: QUIZ

Relax..

This is a closed-book quiz consisting of 3 (three) problems or questions with the point distribution given below:

#1: 15 Points  
#2: 40 Points

#4: 10 Points  
Total: 100 Points

1. (50 Points)

Describe the Z-Buffer algorithm. Provide high-level pseudocode.

30

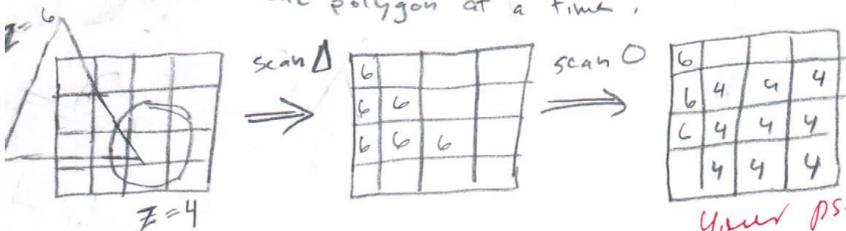
The Z-Buffer algorithm is an imaged-based algorithm as opposed to the object-based methods.

The Z-buffer algorithm draws a ray from the CoP to the view plane. One polygon is analyzed at a time. If the ray intersects the polygon, then the value of the distance at the intersection is copied into the buffer. Then, another polygon is analyzed, and if the ray intersects it, then it compares the distance

to the distance currently stored in the buffer. If the distance is smaller than the current value for that pixel, or there is no data for that pixel, then the value is copied into the buffer. An image can then be extracted from the buffer once all objects are analyzed.

- Pros:
- Commonly Implemented in hardware
  - $O(np)$
  - Objects do not have to be polygons
  - Can examine one polygon at a time.

- Cons:
- Requires a lot of memory
  - Aliasing Effects



Pseudo code

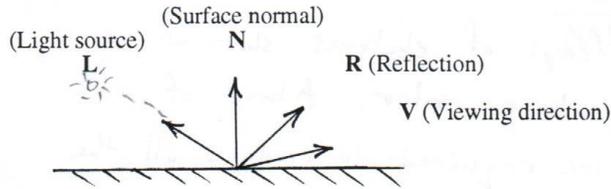
for each  $\Delta$   
 scan  $\Delta$  for intersections  
 if intersects at a point  
 compare current value in buffer  
 if current value is greater  
 - copy new value into the buffer

Your pseudocode must have 2 Buffers!

Initialize buffers  
 Make comparison of Z-value  
 Write onto buffers  
 (if appropriate).

Levi D. Smith

2. (40 Points) Consider the geometry shown in the diagram below, representing the interaction of a light ray with a glossy, reflective surface. The vectors  $L$ ,  $N$ ,  $R$ , and  $V$  are unit vectors.



(a) Assuming the light source is a point source with intensity  $I_{pR}$ , write an expression for the illumination  $I_R$  associated with the red component  $R$  of light based on the Phong Illumination Model.

25

$$I_R = I_{aR} k_a + I_{pR} f_{att} [k_d + k_s \cos^n \alpha] (\vec{N} \cdot \vec{L})$$

$n = \text{glossiness factor}$   
 $I_a = \text{ambient light intensity}$   
 $k_a = \text{ambient light reflection constant}$

$$I_R = I_{aR} k_a O_R + I_{pR} f_{att} [k_d O_R + k_s \cos^n \alpha] (\vec{N} \cdot \vec{L})$$

Diffuse Reflection

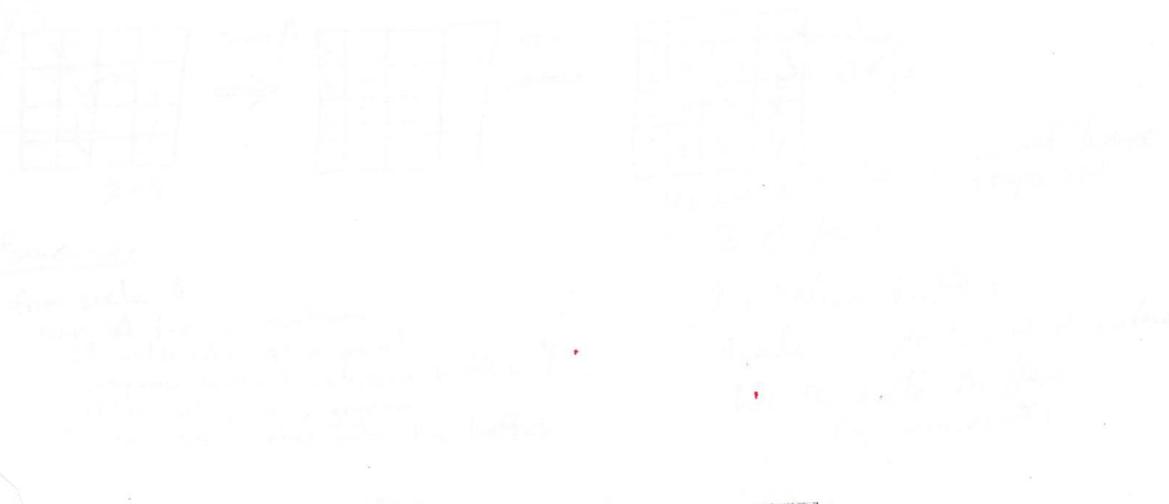
$$I = I_a k_a + I_p k_d (\vec{N} \cdot \vec{L})$$

(b) Give an explicit expression for the attenuation of light as a function of the distance  $R$  from the light source.

$$f_{att} = \frac{1}{R^2}$$
~~$$= \frac{1}{c_1 + c_2 + c_3}$$~~

3. (10 Points) What is Mach banding (or what are Mach Bands)?

Mach bands are the rays of electrons shot at a monitor screen to produce color. A band of red, green, and blue are required to produce all the colors of the visible spectrum.



## CS 4451 Computer Graphics

## Assignment #1

Due: Monday, February 5, 2000, 3:30 PM

February 12, 2001

Purpose: To demonstrate understanding of 3D viewing.

Discussion: This assignment is Foley, van Dam, Feiner, Hughes (FVFH) Exercises 6.1 and 6.2 for perspective projections ( $N_{per}$ ), with clipping in 3D (not in homogeneous coordinates), implementing FVFH Figure 6.46. You are to write a program that displays a (clipped) wireframe model of the house in FVFH Figure 6.24 from any number of different views. Your program will essentially be transforming, clipping and drawing 3D lines.

## Implementation Requirements:

(Some of these requirements are for ease of grading)

1. Your program must accept from stdin the parameters defining a perspective projection view volume, where a set of viewing parameters consists of six lines of text. The following input format and order must be used, and constitutes one set of view parameters (remember, the program must be able to read any number of these sets of parameters and produce the corresponding view each time):

Define the View Reference Coordinate (VRC) System (and thus the view plane):

~ View Reference Point (VRP) - three reals:  $VRP_x, VRP_y, VRP_z$

~ View Plane Normal (VPN) - three reals:  $VPN_x, VPN_y, VPN_z$

~ View Up Vector (VUP) - three reals:  $VUP_x, VUP_y, VUP_z$

Define a window on the view plane:

~ Viewplane Window Bounds - 4 reals:  $u_{min}, u_{max}, v_{min}, v_{max}$

Define the semi-infinite perspective projection view volume:

~ Projection Reference Point (PRP) - 3 reals:  $PRP_u$ ,  $PRP_v$ ,  $PRP_n$

Define a finite view volume:

~ Front, Back Clipping Planes (F, B) - 2 reals:  $F_n$ ,  $B_n$

An example set of parameters:

0 0 54

0 0 1

0 1 0

-1 17 -1 17

8 6 30

1 -23

Your program will be graded by redirecting to stdin from a file, e.g: `%prog1<grade.data` where `grade.data` is the name of a file containing sets of view parameters. Code accordingly!

- The VRC system is right-handed: your 3D transformation matrices should reflect this (e.g., a translation matrix should have  $t_x$ ,  $t_y$ ,  $t_z$  in the rightmost column, not the bottom row.)
- Your program will be tested with view parameters corresponding to FVFH Figures 6.29, 30, 34, and 41, and possibly others. Your program **must** pause between each view, waiting for a left mouse button press. Use either the OpenGL aux library *libaux.a* or GLUT library *libglut.a* to provide this mouse functionality.

Test data can be assumed to be in the correct format and order, but your program should provide an error message (but not terminate) for input that results in an invalid or an unreasonable view, such as the following:

- VPN is degenerate (zero)
- VUP is parallel to VPN
- PRP is on the view plane
- View plane window is reasonable (eg,  $u_{max} \neq u_{min}$ )
- B is in front of F

- Please make the window on the display screen (viewport) 500 x 500!
- The program must be written in C and compile and run using OpenGL on an SGI Indy workstation in the CoC SGI lab. YOU must write the code that implements 3D perspective viewing and clipping, and any matrix math functions. You MAY use *gluOrtho2D()*, but YOU MAY NOT use the following OpenGL calls: *glFrustum()*, *gluPerspective()*, *gluLookAt()*, *gluProject()*, *glTranslate()*, *glRotate()*, *glScale()*. *glClipPlane()*.

3. **ALL source code you submit must be well documented (documentation is an indicator of understanding!)**
4. **You MUST include a REAMDE file that details both how well your program works, any quirks it might have, and a statement that you are trying for the extra credit (if you are doing so.)**

#### Submission:

1. Ensure your name and login name appear in *each file you submit*.
2. Put the files you are submitting (including the Makefile you used) in a directory on a College of Computing UNIX machine, then execute the following command: (The "\_1" corresponds to the assignment number and the <Name> is your LAST NAME.

```
% tar -cvf <Name>_1.tar <Name of file(s)/directory to be tarred>
```

#### For example:

```
% tar -cvf VanHorn_1.tar .
```

This would tar everything in the current directory and put it into the file named VanHorn\_1.tar.

Then mail the tar file to [vanhorn@cc.gatech.edu](mailto:vanhorn@cc.gatech.edu).

3. You can submit as many times as you want, but only the submission closest to the due date/time will be examined and graded.

#### Hints and help:

1. FVFH Chapters 6.5.2 (implementing perspective projection) and 6.5.6 (steps in the overall viewing transformation), Figures 6.55 (clipping to the 3D canonical perspective projection view volume) and 6.59 (matrices involved in the viewing transformation).
2. *IRIS InSight* on-line documentation for OpenGL and the auxiliary and utility libraries; UNIX man pages for OpenGL; OpenGL programming texts; [http://reality.sgi.com/mjk\\_asd/spec3/spec3.html](http://reality.sgi.com/mjk_asd/spec3/spec3.html) for the GLUT library.
3. The diagrams at the end of this assignment may prove useful when debugging your view transformation code.

#### Grading criteria:

FVFH Figure 6.24 wireframe house

Perspective view pipeline of FVFH Figure 6.46

Clipping in 3D (NOT homogeneous coordinates, i.e.,  $N_{per}$  not  $N^2_{per}$ )

1. Submission and Compilation (and following directions) \_\_\_\_\_

- 5 name and login name not on each submitted file
- 5 does not accept view parameters in the order specified
- 5 no README file (and the required information within)
- 50 does not accept view parameters input from stdin
- 100 prohibited calls used
- 100 does not compile on SGI Indy

2. Documentation \_\_\_\_\_

- 5 to -25 insufficiently documented (code not commented, functions not explained, etc.)

3. Code \_\_\_\_\_

+10  $T_{vtp}$ 

+10 R

+10  $T_{ppp}$ +10  $SH_{par}$ +10  $S_{per}$ 

+10 clipping in 3D

+10  $M_{per}$ +10  $M_{VV3DV}$ 

+10 divide by W

+10 matrices constructed and multiplied correctly, and used in the correct order

## 4. Run-time \_\_\_\_\_

- 40 aborts without display

- 10 invalid views not checked and handled with an error message

- 5 viewport not 500 x 500

- 5 does not use left mouse button press for each new view

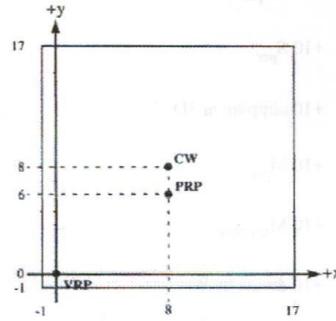
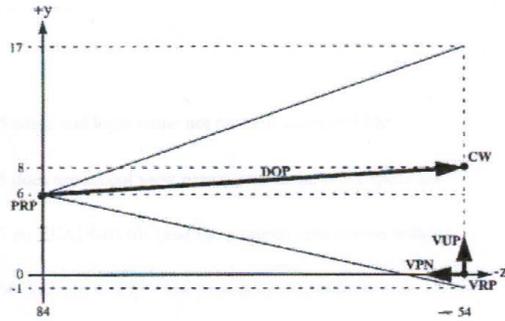
**Extra Credit (+25 points, maximum):**

Add the following capability:

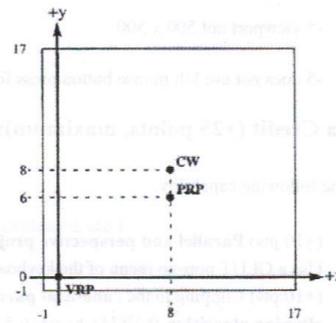
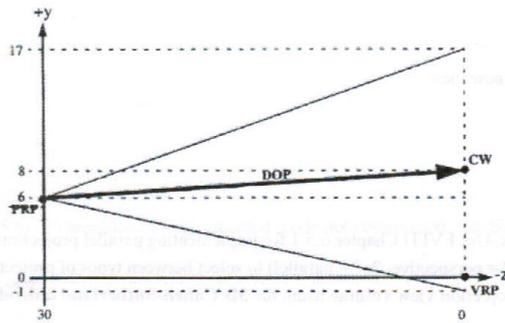
- (+10 pts) **Parallel and perspective projections.** Use FVFH Chapter 6.5.1 for implementing parallel projection viewing. Use a GLUT pop-up menu of the keyboard (?p? = perspective, ?o? = parallel) to select between types of projections.
- (+10 pts) Clipping to the **canonical parallel projection view volume** using the **3D Cohen-Sutherland outcode clipping algorithm** (FVFH Chapter 6.5.3).
- (+5 pts) GLUT pop-up menu is used to select between types of projections.
- 

NOTE: The following equation is wrong: the correct placement of VRP is [ 0 0 -54], not [ 0 0 +54]!

$$\begin{aligned}
 VRP &= \begin{bmatrix} 0 & 0 & 54 \end{bmatrix}_{VRC} & u &= \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \\
 VPV &= \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}_{VRC} & v &= \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \\
 VUP &= \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}_{VRC} & n &= \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \\
 (u_{min}, u_{max}) &= (-1, 17)_{VRC} & CW &= \left[ \frac{17 + (-1)}{2} \quad \frac{17 + (-1)}{2} \quad 0 \right] = [8 \ 8 \ 0]_{VRC} \\
 (v_{min}, v_{max}) &= (-1, 17)_{VRC} & DOP &= CW - PRP = [(8-8) \ (8-6) \ (0-30)] = [0 \ 2 \ -30] \\
 PRP &= \begin{bmatrix} 8 & 6 & 30 \end{bmatrix}_{VRC} \\
 F &= 10_{VRC} \\
 B &= -50_{VRC}
 \end{aligned}$$

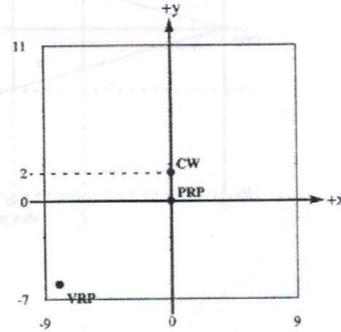
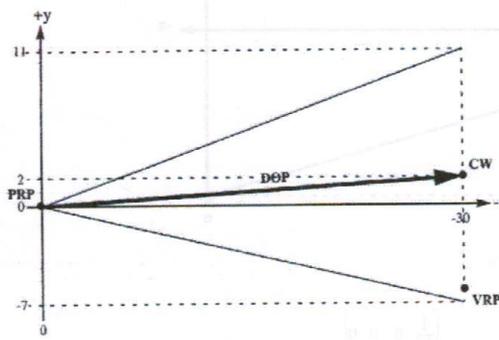


$$T_{vrp} = \begin{bmatrix} 1 & 0 & 0 & -VRP_x \\ 0 & 1 & 0 & -VRP_y \\ 0 & 0 & 1 & -VRP_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -54 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

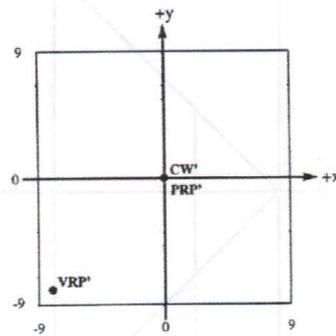
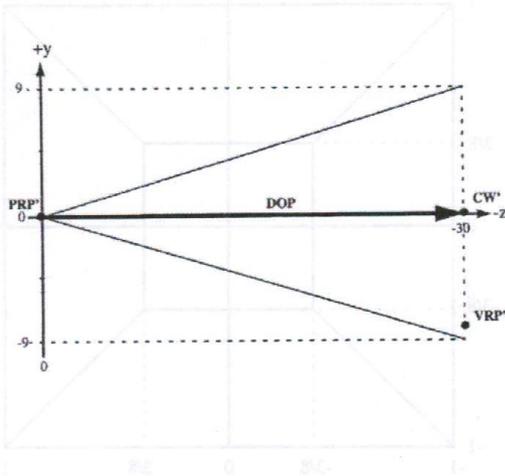


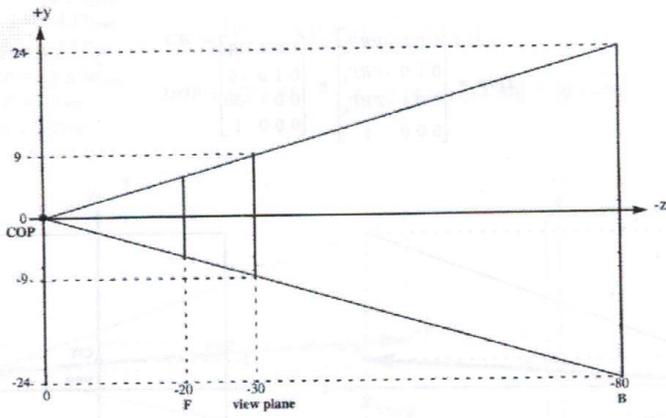
$$R = \begin{bmatrix} u_x & v_x & n_x & 0 \\ u_y & v_y & n_y & 0 \\ u_z & v_z & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{prp} = \begin{bmatrix} 1 & 0 & 0 & -PRP_u \\ 0 & 1 & 0 & -PRP_v \\ 0 & 0 & 1 & -PRP_n \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & -8 \\ 0 & 1 & 0 & -6 \\ 0 & 0 & 1 & -30 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

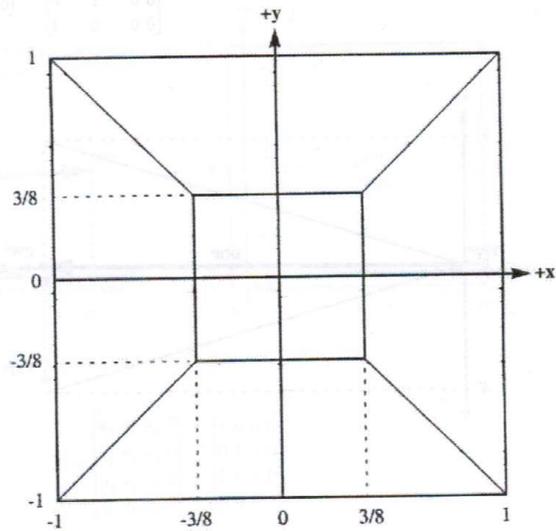
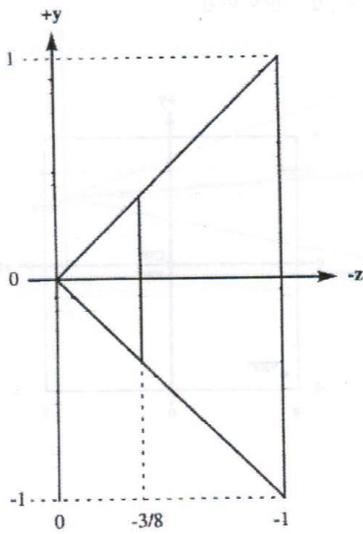


$$SH_{par} = \begin{bmatrix} 1 & 0 & \frac{DOP_u}{DOP_n} & 0 \\ 0 & 1 & \frac{DOP_v}{DOP_n} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & -\frac{2}{-30} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & \frac{1}{15} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$





$$S_{per} = \begin{bmatrix} \frac{1}{24} & 0 & 0 & 0 \\ 0 & \frac{1}{24} & 0 & 0 \\ 0 & 0 & \frac{1}{80} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Assignment #1

Due: Monday, February 5, 2001, 3:30 PM

Purpose: To demonstrate understanding of 3D viewing.

Discussion: This assignment is based on Dan Fagan's Higher (FVH) exercises for perspective projection ( $N_{200}$ ), with clipping in 3D (not in homogeneous coordinates) implementing FVH Figure 6.46. You are to write a program that displays a (copy of) wireframe model of the house in FVH Figure 6.11 from any number of different views. Your program will essentially be transferring, clipping and drawing 3D lines.

Implementation Requirements:

(Some of these requirements are for ease of grading)

- 1. Your program must accept from the user parameters defining a perspective projection view volume, where a set of viewing parameters consists of six floating point. The following input format and order must be used, and constitutes one set of view parameters (remember, the program must be able to read any number of these sets of parameters and produce the corresponding view each time):

Define the View Reference Coordinate (VRC) System (and thus the view plane):

View Reference Point (VRP) - three reals:  $VRP_x, VRP_y, VRP_z$

View Plane Normal (VPN) - three reals:  $VPN_x, VPN_y, VPN_z$

View Up Vector (VUP) - three reals:  $VUP_x, VUP_y, VUP_z$

Define a window on the view plane:

Viewplane Window Bounds - 4 reals:  $w_{min}, w_{max}, y_{min}, y_{max}$

Define the non-infinite perspective projection view volume:

# CS 4451 Computer Graphics

## Assignment #2

**Due: 2 March, 2001 5:00 p.m.**

### Problem Statement

The purpose of this assignment is to write a scan-line polygon renderer that uses the *z-buffer* algorithm for hidden surface computations. The *z-buffer* algorithm is described in **FVFH, Chapter 15.4** (pp. 668-672). You can find a description of the scan-line rendering algorithm in **FVFH, Chapter 15.6** (pp. 680-685).

Since this is a shortened amount of time, *and to help you if you did not finish assignment #1*, there will be code made available to you after the first assignment is turned in. You will be required to fill in the appropriate areas marked as `/* CODE: .... */`.

### Startup Code and Sample Output

The startup code does not implement the *z-buffer* or shading. Sample views will be made available next week.

### Grading Criteria

- +40 points: the program compiles and runs without a core dump.
- +60 points: for test cases *view1*, *view2*, and *view3* (+20 points per test case).

### Extra Credit

- +20 points: implement *Gouraud Shading*. See **FVFH, Chapter 16.2.4** (pp. 736 - 737) for details.
- +30 points: implement *Phong Shading*. See **FVFH, Chapter 16.2.5** (pp. 738 - 739) for details.

### Notes

- For shading you are to use normals of the faces -- you do not have to calculate normals at the vertices.
- You have to add light sources in order to showcase the shading capabilities of your renderer.
- In order to receive any of the extra credit, you need to fill in the code marked as `/* Extra Credit Code: .... */`.

### Submission Rules

- The program must be written in C or C++ and compile and run using OpenGL on an SGI Indy workstation in the CoC SGI lab. YOU must write the code that implements the *z-buffer* (and the shading, if you want extra credit.)
- All source code (that you write) must be well documented (documentation is an indicator of understanding!)

- You MUST include a README file that details both how well your program works, and quirks it might have, and a statement that you are trying for the extra credit (if you are doing so.)
- Ensure your name and login name appear in *each file you submit*.
- Put the files you are submitting (including the Makefile you used) in a directory on a CoC UNIX machine, then execute the following command (The <Name> is your LAST NAME):  
`% tar -cvf <NAME>_2.tar <Name of file(s)/directory to be tarred>`

**For example:**

**atlanta> tar -cvf VanHorn\_2.tar .**

This would tar everything in the current directory and put it into a file named VanHorn\_2.tar

**Then mail the tar file to [vanhorn@cc.gatech.edu](mailto:vanhorn@cc.gatech.edu).**

- You can submit as many times as you want, but only the submission closest to the due date/time will be examined and graded.
- No late submissions at all!!

## Binary Spanning Partition

### Z-buffering

- + commonly implemented in HW
- + objects do not have to be AAs
- +  $O(N^2)$
- + no object comparisons
- requires lots of memory
- aliasing effects

### Illumination Models

Flat -

Phong - interpolate normals

Govard - calculate average vector, then interpolate across 4 edges, then interpolate across scan lines

Reflection - Lambert's Law  $I = I_a k_a + I_p k_d (\bar{N} \cdot \bar{L})$

Specular Reflection -

### Illumination

Phong -  $I_r = I_a k_a O_d + \int_{att} I_p [k_d O_d \cos \theta + W(\theta) \cos^n \alpha]$

Govard  
- calculate the normal of a vertex by averaging the normals of all vectors  
- calculate intensity

# CS 4451 Computer Graphics

## Assignment #3

**Due: 30 March, 2001 5:00 p.m.**

### Problem Statement

The purpose of this assignment is to write a shader that implements *Gouraud, Phong, and Flat shading* as found in **FVFH, Chapter 16.2** (pp. 734-741). This uses the z-buffer algorithm as described in **FVFH, Chapter 15.4** (pp. 668-672) and **Chapter 15.6** (pp. 680-685). Notice that this z-buffer algorithm is the second version of the algorithm that I taught you in class.

This again is a shortened amount of time, so code has been provided for you. You are not required to use it, but I *highly suggest using it*...it is sufficiently written well enough for our needs (You will note that aliasing occurs, but I am not bothered by that for now.) You should be able to implement everything just by filling in the areas marked `/* CODE */`.

### Startup Code and Sample Output

The startup code does not implement any kind of shading. Sample views will be made available soon (along with the test data I will be using to grade you.) The extra credit code will be made available no later than Friday night.

### Grading Criteria

- 05 points: no acceptable README file
- 05 points: no Makefile
- 05 points: viewport not 500x500
- 05 points: insufficient documentation
- 10 points: does not display correctly

- 10 points: incorrect Flat shading
- 30 points: incorrect Gouraud shading
- 30 points: incorrect Phong shading

### DealBreakers:

- 100 points: uses OpenGL to display the shadings.
- 75 points: does not accept input
- 60 points: the program does not compile and/or does not run without coredumping.

### Extra Credit

- +25 points: implement all types of shading using OpenGL (You must use the extra credit code provided!)

+2 points: For each *true* bug in the code provided: the first person to find and post on the newsgroup that bug, I will award two points of extra credit. There is a maximum of ten bonus points from this method allowed. The post in the newsgroup must be labeled "Bug Found?" and contain a clear and concise description of the bug. The decision that the bug is *true* will be made solely by the T.A. He will reply to the post either "Affirmative" or "Negative" in the heading along with reasons why. I hope that no bugs exist, but c'est la vie of a coder.

### Notes

- The hardest type of shading to do correctly is Phong. Therefore, try Gouraud and Flat first. Get them working and then go to Phong.
- Even though you have less than 200 lines of code to write, you must account for both coding AND getting used to my code. It is still C++, so start right now (And remember, those that find bugs first, get credit for it!)
- Make use of the newsgroup to help each other out!

### Submission Rules (NEW!!)

- The program must be written in C or C++ and compile and run using OpenGL on an SGI Indy workstation in the CoC SGI lab. YOU must write the code that implements the shading.
- All source code (that you write) must be well documented (documentation is an indicator of understanding!)
- You MUST include a README file that details both how well your program works, and quirks it might have, and a statement that you are trying for the extra credit (if you are doing so.) Also, you must tell me how to run your program. Finally, let me know if you are using a late token.
- Ensure your name and login name appear in *each file you submit*.
- Put the files you are submitting (including the Makefile you used, *but NO executables or object files!*) in a directory on a CoC Solaris machine, where the name of the directory is your last name followed by an "\_" and your first name. Thus, the directory would be for me: **Vanhorn\_Brooks**. Then execute the following command (The <dirname> is the name of the directory):

```
% tar -cvf <dirname>_3.tar <dirname>
```

For example:

```
atlanta% tar -cvf VanHorn_Brooks_3.tar Vanhorn_Brooks
```

This would tar everything in the Vanhorn\_Brooks directory.

Then mail it to the T.A. at: [vanhorn@cc.gatech.edu](mailto:vanhorn@cc.gatech.edu) with the subject heading "**Project # 3, Submission # [X]**" where [X] is the number of the current submission.

- You can submit as many times as you want, but only the last submission will count. So, if you turn in you last submission an hour after the deadline, then you've just used your late token (if you have one. If you don't have one, then you missed the assignment completely and will receive a zero for the third assignment.)

Name: Levi Smith

Submission and Compilation:

- 5 name and login not on each submitted file
- 5 does not accept view parameters in order specified
- 5 no README file
- 50 does not accept view parameters input from stdin
- 100 prohibited calls used
- 100 does not compile on SGI Indy

Documentation:

- 5 to -25 insufficiently documented

50  
100

Code:

- ~~-10~~ incorrect Tvrp
- 10 incorrect R
- 10 incorrect Tprp
- 10 incorrect SHpar
- 10 incorrect Sper
- 10 incorrect clipping in 3D
- 10 incorrect Mper
- 10 incorrect Mvv3dv
- 10 incorrect divide by W
- 10 incorrect matrix operations/construction

Sper: Didn't use  $v_{rp}^t$  - instead you used  $v_{pp}$ .

Run-time:

- 40 aborts without display
- 10 invalid views not checked
- 5 viewport not 500x500
- 5 does not use left mouse for each new view

Extra Credit:

- +10 Parallel projection
- +10 Clipping to canonical parallel projection
- +5 GLUT pop-up menu to select types of projections

NOTES:

=====

Name: Levi Smith

Assignment # 2

Submission and Compilation:

- 5 No acceptable README file
- 5 No acceptable Makefile
- 5 Insufficient Documentation
- 60 Does not compile and/or run without core-dumping.

60  
100

Code:

- 35 Incorrect Intersection Code
- 35 Incorrect Z-Buffer Code
- 20 Didn't try to find intersection point between ray and triangle
- 5 incorrect distance function
- 5 incorrect way to find angle

Other:

- 5 Viewport is not 500x500
- 10 Does not display correctly
- 70 Does not accept input

75  
85  
70  
-230

Extra Credit:

- +20 Gouraud Shading
- +30 Phong Shading

Other:

Name: Levi Smith

Assignment # 3

Submission and Compilation:

- 05 points: No acceptable README file
- 05 points: no makefile
- 05 points: viewport not 500x500
- 05 points: insufficient documentation
- 10 points: does not display correctly.

GPH

55  
100

Code:

- 10 points: incorrect Flat Shading
- 30 points: incorrect Gouraud Shading
- 30 points: incorrect Phong Shading

→ -25 little phong code.  
-10 incorrect vertex intensity interpolation

Other:

- 100 points: uses OpenGL to display the shadings
- 75 points: does not accept input
- 60 points: the program does not compile and/or run w/o core dumping.

Extra Credit:

- +35 points: Uses OpenGL to display the shadings
- +2 -> +10 points: Found a bug in program and posted it on newsgroup

Hi, everyone.

Just a reminder that this coming Monday (the 24th.) we will hear the presentations on your projects.

To make the most out of your team's presentations, please observe the following suggestions:

1. Each team will have a max of eight minutes. (There \*might\* be time for questions but I'll start applauding at the eight minute mark to drown out over-runs - an old and efficient practice).
2. Have one speaker only (two or more speakers make adhering to the time limit difficult).
3. For the presentation itself, make sure the spokesperson presents clearly:
  - (a) The topic of the project (a title would be useful)
  - (b) The relevance of this topic to the class - *next step in polygon detail*
  - (c) What was done by the team: what was either written, and/or programmed, and/or implemented from another source, etc. *Tutorial + code*
  - (d) Point out WHAT WAS THE TEAM'S CONTRIBUTION (i.e., what is new and/or what the main effort/work was). *used original Open GL code, but wrote our own draw methods*
  - (e) Challenges or problems encountered, or technical limitations inherent in the technique discussed. *For research, sources are cited in lecture*
  - (f) Summary: What was done, what was learned, what additional research may be possible, etc.

\*\* In general, your presentation should motivate others to take a look at the actual work done. \*\*

NOTE THAT IF YOU FOLLOW THIS OUTLINE (AND YOU SHOULD), each point (b) through (f) would be assigned about 1.5 minutes - you don't have to give equal time to each point, but you should address them all.

4. Rehearse the presentation.

5. As you listen to the presentations, you should formulate a grade based on the criteria discussed: relevance or originality; level of effort,

overall quality.

6. For those requiring OpenGL, Brooks has asked me to forward the following:

- >
- > Please send out an announcement that we will have opengl, but only on an
- > 02 machine - tell them to make sure that their programs work on the 02
- > machines.

Finally, on Wednesday we'll have a final quiz.

\* GRADUATING SENIORS: For graduating seniors, I will look at your grades for the semester and indicate whether you have the option of not taking the final quiz (if you have a stellar record or if your grades will not change as a result of the quiz's grade).

Cheers,

Prof. Ezquerro

```

import java.awt.*; import java.awt.event.*; import javax.swing.*;

/* For the purpose of simplifying this code, the view window is
fixed and stretches from the origin to botRight.*

public class J3DPanel extends JPanel
{
    /* Viewing Data */
    private Point topLeft = new Point(0.0, 0.0, 0.0);
    private Point botRight = new Point(100, -100, 0.0);
    private Point lightSource = new Point(50, -50, 50.0);
    private Point viewPoint = new Point(50.0, -50.0, 50.0);

    /* Viewing State */
    public static final int PHONG = 0;
    public static final int BUMP = 1;
    private int renderMode = BUMP;
    private int bumpMap = 0;

    /* GUI */
    private Bump mainWindow;

    public void switchMap() // this switches to the next bump map
    {
        bumpMap++; this.repaint();
    }

    public void switchType() // this turns bump mapping on and off
    {
        if (renderMode == PHONG)
            renderMode = BUMP;
        else
            renderMode = PHONG;
        this.repaint();
    }

    public J3DPanel (Bump mainWindow) // GUI stuff constructor
    {
        this.mainWindow = mainWindow;
        this.addMouseListener(new EventHandler(this));
        this.repaint();
    }

    public void paintComponent(Graphics g) // zBuffering here
    {
        int height = this.getHeight();
        int width = this.getWidth();

        float intensity = (float) 0.0;
        Point current, normal; // go pixel
        for (int x = 0; x < width; x++) // by pixel
            for (int y = 0; y < height; y++)
            {
                /* This method for obtaining the point and its
                normal is not generalizable since it's using
                the simplifications we made. */
                current = new
                Point(((double)x/(double)width
                    * botRight.x,
                    ((double)y/(double)height)
                    * botRight.y,
                    0.0);

                normal = new Point(0.0, 0.0, 1.0);

                if (renderMode == PHONG)
                    intensity =
                    phongIntensity(current,normal);
                else if (renderMode == BUMP)
                    intensity =
                    bumpIntensity(current,normal);

                intensity = (float) (0.3 + 0.7*intensity);
                g.setColor(new Color(
                    (float)0.0,
                    (float)0.0,
                    intensity));

                g.fillRect(x,y,x+1,y+1);
            }

        public float bumpIntensity(Point P, Point N)
        {
            double epsilon = 0.0001;
            /* pick 2 nearby points */
            Point Pp = P.plus(N.times(bump(P.x,P.y)));
            Point P1 = P.plus(epsilon,epsilon,0.0);
            plus(N.times(bump(P.x+epsilon,P.y+epsilon)));
            Point P2 = P.plus(epsilon,-epsilon,0.0);
            plus(N.times(bump(P.x+epsilon,P.y-epsilon)));

            /* interpolate the normal */
            Point v1 = P1.minus(Pp);
            Point v2 = P2.minus(Pp);
            Point Np = (v2.cross(v1)).normalize();

            return phongIntensity(Pp,Np); // calculate the intensity
        }

        public double bump (double x, double y)
        {
            switch (bumpMap)
            {
                case 0:
                    mainWindow.setLabel(
                        "B(x,y) = sin((x^2 + y^2)^0.5)");
                    return Math.sin(Math.pow(x*x + y*y,0.5));

                case 1:
                    mainWindow.setLabel("B(x,y) = sin(x) + cos(y)");
                    return Math.sin(x) + Math.cos(y);

                case 2:
                    mainWindow.setLabel(
                        "B(x,y) = sin(x) + Math.sin(y)");
                    return Math.sin(x) + Math.sin(y);
            }
        }
    }
}
J3DPanel.java - 2

```

```

        "B(x,y) = sin(((x-50)^2 + (y+50)^2)^0.5)");
        return
        Math.sin(Math.pow((x-50)*(x-50) + (y+50)*(y+50),0.5)) ;
    }

    /* ADD YOUR OWN BUMP MAPS HERE!
    * JUST ADD A CASE STATEMENT, SET THE LABEL IF YOU WANT
    * AND RETURN THE FUNCTION OF THE BUMP MAP YOU WANT TO TRY.
    * KEEP ITS RETURN VALUE SMALL, AND REMEMBER THAT THE CENTER
    * OF THE WINDOW IS AT (50,-50) ON THE X-Y AXIS. */

    default:
        bumpMap = 0 ;
        return Math.sin(Math.pow(x*x + y*y,0.5)) ;
    }
}

public float phongIntensity(Point p, Point N)
{
    double diffuse = 0.0 , specular = 0.0 ;
    N.normalize() ;

    Point L = (lightSource.minus(p)).normalize() ;
    diffuse = N.dot(L) ;
    if (diffuse < 0.0)
        diffuse = 0.0 ;

    Point V = (viewPoint.minus(p)).normalize() ;
    Point R = (N.times(2*diffuse)).minus(L) ;
    specular = R.dot(V) ;
    if (specular < 0.0)
        specular = 0.0 ;

    return (float)(diffuse+(1-diffuse)*specular) ;
}

}

class Point
{
    public double x ;
    public double y ;
    public double z ;

    public Point () { this(0.0,0.0,0.0) ; }

    public double magnitude ()
    {
        return Math.pow(x*x + y*y + z*z,0.5) ;
    }

    public Point normalize ()
    {
        double mag = magnitude() ;
        x /= mag ; y /= mag ; z /= mag ;
        return this ;
    }

    public Point (double x, double y, double z)
}

}

    {
        this.x = x ; this.y = y ; this.z = z ;
    }

    public Point times(double d)
    {
        return new Point(d*x, d*y, d*z) ;
    }

    public Point cross(Point p)
    {
        return new Point( y*p.z - z*p.y ,
            z*p.x - x*p.z ,
            x*p.y - y*p.x ) ;
    }

    public Point plus(double a, double b, double c)
    {
        return new Point(x+a, y+b, z+c) ;
    }

    public Point plus(Point p)
    {
        return new Point(x+p.x, y+p.y, z+p.z) ;
    }

    public Point minus(Point p)
    {
        return new Point(x-p.x, y-p.y, z-p.z) ;
    }

    public double dot(Point p)
    {
        return x*p.x + y*p.y + z*p.z ;
    }
}

class EventHandler implements MouseListener // GUI Garbage
{
    J3DPanel model ;
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
    public void mousePressed(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}
    public void mouseClicked(MouseEvent e) {}

    if (e.getModifiers() == e.BUTTON1_MASK)
        model.switchType() ;
    else
        model.switchMap() ;
}

}

J3DPanel.java - 4

```

```

/**
 * Levi D. Smith
 * command@cc.gatech.edu
 * February 25, 2001
 */
// Assignment # 2: Ver. 1.0

#include <glut.h>
#include <GL/gl.h>
#include <GL/glu.h>

#include <math.h>
#include <stdlib.h>
#include <vector>
#include <fstream>
#include <iostream>
using namespace std;

/** CONSTANTS **/
const double MAXIMUM_D_VALUE = 9e19;
const double EPSILON = 1e-6;

/** FUNCTION PROTOTYPES **/
void mouse( int, int, int, int );
void display( void );
void reshape( int, int );
void init( char * );
bool readTRI( char * );
void MidPoint( void );
void zBuffer( int index );
void draw_buffer( void );
void reset_buffer( void );

class POINT
{
public:
    POINT( double a = 0.0, double b = 0.0, double c = 0.0 )
    {
        x[ 0 ] = a;
        x[ 1 ] = b;
        x[ 2 ] = c;
    };

    ~POINT() {}

    inline double & operator [] ( int index )
    {
        return x[ index ];
    }

    void normalize( void )
    {
        double den = 1 / magnitude();

        for ( int m = 0; m < 3; m++ )
            x[ m ] *= den;
    }

    double magnitude( void )
    {
        return sqrt( x[ 0 ] * x[ 0 ] + x[ 1 ] * x[ 1 ] + x[ 2 ] * x[ 2 ] );
    }

    inline bool operator == ( const POINT &p )
    {
        return ( x[ 0 ] == p.x[ 0 ] && x[ 1 ] == p.x[ 1 ] && x[ 2 ] == p.x[ 2 ] );
    }
};

```

```

inline bool zero( void )
{
    return ( x[ 0 ] == x[ 1 ] && x[ 0 ] == x[ 2 ] && x[ 0 ] == 0 );
}

// I provided this function to help you debug
void print( void )
{
    cout << "\n===== \n" << x[ 0 ] << " " << x[ 1 ]
         << " " << x[ 2 ] << "\n===== " << endl;
}

private:
    double x[ 3 ];
};

/////
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
/////

/////
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
/////

static int xdim, ydim; // window

static double RATIO_M_X, RATIO_B_X, RATIO_M_Y, RATIO_B_Y;

static bool beginning = true;
static bool gouraud = false;
static bool phong = false;

static POINT disp_right;
static POINT disp_down;
static POINT viewpoint;
static POINT topleft;
static POINT midpoint;
static POINT v_min, v_max;
static unsigned long num_Ts;
static unsigned long num_Vs;

static double *Z_BUFFER;
static unsigned long *TRIANGLE_BUFFER;

static POINT *v_table;
static POINT *T_normals;
static unsigned int *T_table;

/////
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
/////

/////
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
/////

```

```

////
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
////

void mouse( int button, int state, int x, int y )
{
    switch( button )
    {
        case GLUT_LEFT_BUTTON:
            if ( state == GLUT_DOWN )
            {
                phong = true;
                gouraud = false;
            }
            break;

        case GLUT_MIDDLE_BUTTON:
            if ( state == GLUT_DOWN )
            {
                phong = false;
                gouraud = true;
            }
            break;

        case GLUT_RIGHT_BUTTON:
            if ( state == GLUT_DOWN )
            {
                phong = gouraud = false;
            }
            break ;

        default:
            break ;
    }
};

////
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
////

////
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
////

void Axis( void )
{
    glPushMatrix();
    glLoadIdentity();
    glColor3f( 0.0, 0.0, 1.0 );
    glBegin( GL_LINES );

    // X axis
    glVertex3f( -100.0, 0.0, 0.0 );
    glVertex3f( 100.0, 0.0, 0.0 );
    // Y axis
    glVertex3f( 0.0, -100.0, 0.0 );
    glVertex3f( 0.0, 100.0, 0.0 );
    // Z axis

```

```

    glVertex3f( 0.0, 0.0, -100.0 );
    glVertex3f( 0.0, 0.0, 100.0 );

    glEnd();
    glPopMatrix();
}

/////
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
/////

void display( void )
{
    if ( beginning )
    {
        beginning = false;
        return ;
    }

    glClear( GL_COLOR_BUFFER_BIT );
    glMatrixMode( GL_MODELVIEW );

    // Draw the three axis ( x-, y-, and z- )
    Axis();

    cout << "Starting to Z-Buffer..." << endl;

    glLoadIdentity();

    // We'll go through and do the Z-Buffer algorithm
    // to each polygon
    for ( int i = 0; i < num_Ts; i++ )
        zBuffer( i );

    // now that we've done the Z-buffer, we'll display
    draw_buffer();

    glFlush();
}

/////
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
/////

void reshape( int new_width, int new_height )
{
    xdim = new_width;
    ydim = new_height;

    if ( Z_BUFFER != NULL )
        delete [] Z_BUFFER;
    if ( TRIANGLE_BUFFER != NULL )
        delete [] TRIANGLE_BUFFER;

    Z_BUFFER = new double[ xdim * ydim ];
    TRIANGLE_BUFFER = new unsigned long[ xdim * ydim ];

    reset_buffer();

    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    glViewport( 0, 0, (GLsizei) new_width, (GLsizei) new_height );

```





```

////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
void zBuffer( int index )
{
    // These are three vertices of the triangle you're working with
    POINT v0( v_table[ T_table[ index + 0 ] ][ 0 ],
              v_table[ T_table[ index + 0 ] ][ 1 ],
              v_table[ T_table[ index + 0 ] ][ 2 ] );
    POINT v1( v_table[ T_table[ index + 1 ] ][ 0 ],
              v_table[ T_table[ index + 1 ] ][ 1 ],
              v_table[ T_table[ index + 1 ] ][ 2 ] );
    POINT v2( v_table[ T_table[ index + 2 ] ][ 0 ],
              v_table[ T_table[ index + 2 ] ][ 1 ],
              v_table[ T_table[ index + 2 ] ][ 2 ] );

    /* CODE */
    // Insert Your Code HERE //

}

////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////

void draw_buffer( void )
{
    cout << "Drawing buffer...";
    cout.flush();

    int max = ydim * xdim;

    RATIO_M_X = ( v_max[ 0 ] - v_min[ 0 ] ) / xdim;
    RATIO_M_Y = ( v_max[ 1 ] - v_min[ 1 ] ) / ydim;

    RATIO_B_X = v_min[ 0 ];
    RATIO_B_Y = v_min[ 1 ];

    glBegin( GL_POINTS );
    for ( int i = 0; i < max; i++ )
        if ( fabs( Read_Z( i ) ) < MAXIMUM_D_VALUE )
            triangle_color( i );
    glEnd();

    cout << "finished. " << endl;
}

////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////

void init( char *name )
{
    readTRI( name );

    glClearColor( 0.0, 0.0, 0.0, 0.0 );
    glShadeModel( GL_FLAT );

    glMatrixMode( GL_PROJECTION );

```

```

glLoadIdentity();

gluLookAt( viewpoint[ 0 ], viewpoint[ 1 ], viewpoint[ 2 ],
           midpoint [ 0 ], midpoint [ 1 ], midpoint [ 2 ],
           disp_down[ 0 ], disp_down[ 1 ], disp_down[ 2 ] );

gluOrtho2D( v_min[ 0 ], v_max[ 0 ], v_max[ 1 ], v_min[ 1 ] );

glMatrixMode( GL_MODELVIEW );
}

/////
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
/////

bool readTRI( char *name )
{
    ifstream fin( name );

    v_table = NULL;
    T_table = NULL;
    T_normals = NULL;

    if ( !fin.is_open() )
    {
        cerr << "Error opening output file" << endl;

        disp_right = POINT( 1.0, 0.0, 0.0 );
        disp_down  = POINT( 0.0, -1.0, 0.0 );
        topleft    = POINT( 100, 100, 0 );
        v_min      = POINT( -1e9, -1e9, -1e9 );
        v_max      = POINT( 1e9, 1e9, 1e9 );

        viewpoint  = POINT( 0, 0, -5 );

        midpoint   = POINT( 0.0, 0.0, 0.0 );

        return false;
    }

    // read in the viewpoint (camera placement)
    fin >> viewpoint[ 0 ] >> viewpoint[ 1 ] >> viewpoint[ 2 ];

    // read in the screen plane
    fin >> topleft[ 0 ] >> topleft[ 1 ] >> topleft[ 2 ];

    // read distance between pixels
    fin >> disp_right[ 0 ] >> disp_right[ 1 ] >> disp_right[ 2 ];
    fin >> disp_down[ 0 ] >> disp_down[ 1 ] >> disp_down[ 2 ];

    // # of triangles
    fin >> num_Ts;

    // set up the vertex table and the triangle table
    v_table = new POINT [ 3 * num_Ts ];
    T_normals = new POINT[ 3 * num_Ts ];
    T_table = new unsigned int[ 3 * num_Ts ];

    double temp[ 3 ][ 3 ];
    short flag[ 3 ];

    num_Vs = 0;

    for ( int i = 0; i < num_Ts; i++ )

```

```

{
    flag[ 0 ] = flag[ 1 ] = flag[ 2 ] = -1;
    // Read in 3 sets of vertices that make up one triangle
    for ( int n = 0; n < 3; n++ )
        for ( int m = 0; m < 3; m++ )
            fin >> temp[ n ][ m ];

    // Have we added this vertex before?
    for ( int m = 0; m < 3; m++ )
        for ( int j = num_Vs - 1; j >= 0; j-- )
            if ( temp[ m ][ 0 ] == v_table[ j ][ 0 ] &&
                temp[ m ][ 1 ] == v_table[ j ][ 1 ] &&
                temp[ m ][ 2 ] == v_table[ j ][ 2 ] )
                {
                    flag[ m ] = j;
                    break ;
                }

    for ( int m = 0; m < 3; m++ )
        if ( flag[ m ] == -1 )
            // we haven't added this vertex yet
            {
                v_table[ num_Vs ] = POINT( temp[ m ][ 0 ], temp[ m ][ 1 ],
                                           temp[ m ][ 2 ] );

                flag[ m ] = num_Vs++;
            }

    // Insert which indices into the v_table refer to this triangle
    int j = i * 3;
    for ( int m = 0; m < 3; m++ )
        {
            // taking this out for this project
            // incident[ flag[ m ] ].push_back( i );
            T_table[ j + m ] = flag[ m ];
        }
}

MidPoint();

cout << "Viewpoint: ( " << viewpoint[ 0 ] << ", "
      << viewpoint[ 1 ] << ", " << viewpoint[ 2 ] << " )\n"
      << "Midpoint: ( " << midpoint[ 0 ] << ", "
      << midpoint[ 1 ] << ", " << midpoint[ 2 ] << " )\n"
      << "Minimum: ( " << v_min[ 0 ] << ", "
      << v_min[ 1 ] << ", " << v_min[ 2 ] << " )\n"
      << "Maximum: ( " << v_max[ 0 ] << ", "
      << v_max[ 1 ] << ", " << v_max[ 2 ] << " )\n"
      << "Number of Vertices: " << num_Vs
      << "\nNumber of Triangles: " << num_Ts << endl;

return true;
}

/////
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
/////

void MidPoint(void)
{
    for ( int m = 0; m < 3; m++ )
        {
            v_min[ m ] = MAXIMUM_D_VALUE;

```

```

        v_max[ m ] = -MAXIMUM_D_VALUE;
    }

    for ( int j = 0; j < num_Vs; j++ )
        for ( int m = 0; m < 3; m++ )
            {
                v_min[ m ] = ( v_table[ j ][ m ] < v_min[ m ] ) ? v_table[ j ][ m ] : v_min[ m ];
                v_max[ m ] = ( v_table[ j ][ m ] > v_max[ m ] ) ? v_table[ j ][ m ] : v_max[ m ];
            }

    for ( int m = 0; m < 3; m++ )
        {
            midpoint[ m ] = ( v_min[ m ] + v_max[ m ] ) * 0.5;
            double diff = ( v_max[ m ] - v_min[ m ] ) * 0.2;

            v_min[ m ] -= diff;
            v_max[ m ] += diff;
        }
}

/////
/////
/////
/////
/////

/**
 * called first when the program is executed
 */
int main( int argc, char *argv[] ) {
    if ( argc < 2 )
        {
            cerr << "Usage error: " << argv[ 0 ] << " <name of input file>" << endl;
            exit( 1 );
        }

    xdim = ( argc > 2 ) ? atoi( argv[ 2 ] ) : 500;
    ydim = ( argc > 3 ) ? atoi( argv[ 3 ] ) : 500;

    init( argv[ 1 ] );

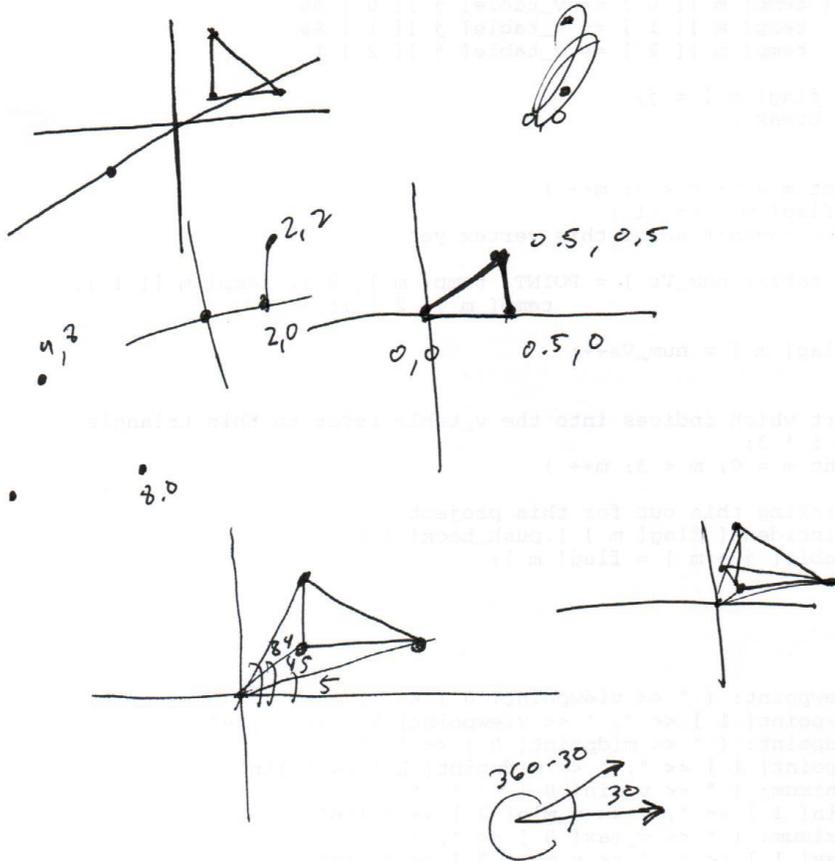
    glutInit( &argc, argv );
    glutInitDisplayMode( GLUT_SINGLE | GLUT_RGB );
    glutInitWindowSize( xdim, ydim );
    glutCreateWindow( "Z-Buffering, Z-Buffering..." );
    glClearColor( 0.0, 0.0, 0.0, 0.0 );
    glShadeModel( GL_FLAT );

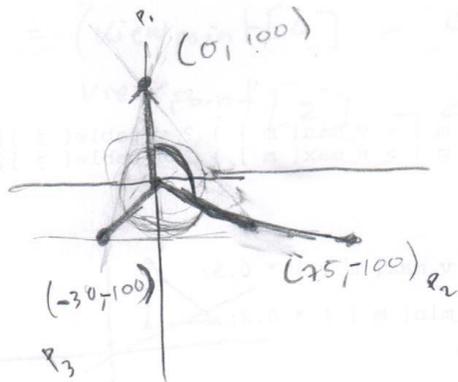
    glutDisplayFunc( display );
    glutReshapeFunc( reshape );
    glutMouseFunc( mouse );

    glutMainLoop();
}

```

$$\text{Ray} = (\text{viewpoint}[0] - 0, \text{viewpoint}[1] - 0, \text{viewpoint}[2] - 0)$$



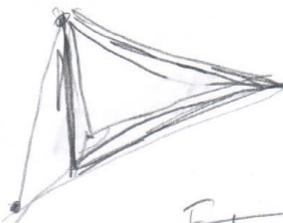
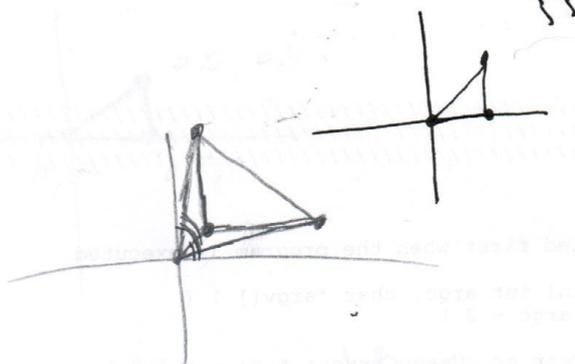


$$P_1 = 90^\circ$$

$$P_2 = 270^\circ - 360^\circ$$

$$P_3 = 180^\circ - 270^\circ$$

$$\begin{matrix} 0 & 0 & 0 \\ 0.5 & 0 & 0 \\ 0.5 & 0.5 & 0 \end{matrix}$$



Find the largest angle containing all three vectors

If one of the small angles is greater than  $90^\circ$ , then  $t \rightarrow 180^\circ$

normalize current point

```
//  
//  
// * You need to fill in the information about yourself */  
//  
//  
//  
// KEEP THE FOLLOWING LINE:  
// Assignment # 3: Ver. 1.1  
  
#include <glut.h>  
#include <GL/gl.h>  
#include <GL/glu.h>  
  
#include <math.h>  
#include <stdlib.h>  
#include <vector>  
#include <fstream>  
#include <iostream>  
using namespace std;  
  
// this lets us know some stats as they occur  
// DO NOT ERASE ANY OF THESE!!! I WILL USE THIS TO  
// HELP ME GRADE YOUR PROJECTS!!  
#define _TA_STATS_  
  
// Use this one if you want to add your own debugs  
#define _YOUR_STATS_  
  
/////  
////////////////////////////////////  
////////////////////////////////////  
////////////////////////////////////  
////////////////////////////////////  
////////////////////////////////////  
  
// Largest Value  
const double MAXIMUM_D_VALUE = 9e200;  
// Smallest Value  
const double EPSILON = 1e-10;  
  
/////  
////////////////////////////////////  
////////////////////////////////////  
////////////////////////////////////  
////////////////////////////////////  
////////////////////////////////////  
  
/////  
////////////////////////////////////  
////////////////////////////////////  
////////////////////////////////////  
////////////////////////////////////  
////////////////////////////////////  
  
/////  
////////////////////////////////////  
////////////////////////////////////  
////////////////////////////////////  
////////////////////////////////////  
////////////////////////////////////  
  
class POINT  
{  
public:  
    POINT( double a = 0.0, double b = 0.0, double c = 0.0 )  
    {  
        x[ 0 ] = a;  
        x[ 1 ] = b;  
        x[ 2 ] = c;  
    };  
  
    ~POINT() {}  
  
    inline double & operator [] ( int index )  
    {  
        return x[ index ];  
    }  
  
    void normalize( void )  
    {  
        double den = 1 / magnitude();  
        for ( int m = 0; m < 3; m++ )  
            x[ m ] *= den;  
    }  
  
    double magnitude( void ) const  
    {  
        return sqrt( x[ 0 ] * x[ 0 ] + x[ 1 ] * x[ 1 ] + x[ 2 ] *  
x[ 2 ] );  
    }  
};
```

```
};  
  
inline bool operator == ( const POINT &p ) const  
{  
    return ( x[ 0 ] == p.x[ 0 ] && x[ 1 ] == p.x[ 1 ] && x[ 2 ]  
== p.x[ 2 ] );  
}  
  
inline bool operator != ( const POINT &p ) const  
{  
    return ( x[ 0 ] != p.x[ 0 ] || x[ 1 ] != p.x[ 1 ] || x[ 2 ]  
!= p.x[ 2 ] );  
}  
  
inline bool zero( void ) const  
{  
    return ( x[ 0 ] == x[ 1 ] && x[ 0 ] == x[ 2 ] && x[ 0 ]  
== 0 );  
}  
  
////////////////////////////////////  
  
// I provided this function to help you debug  
void print( void ) const  
{  
    cout << "\n===== \n" << x[ 0 ] << " " << x[ 1 ]  
<< " " << x[ 2 ] << "\n===== " << endl;  
}  
  
////////////////////////////////////  
  
void reverse( void )  
{  
    x[ 0 ] = -x[ 0 ];  
    x[ 1 ] = -x[ 1 ];  
    x[ 2 ] = -x[ 2 ];  
}  
  
POINT & operator += ( const POINT &p )  
{  
    x[ 0 ] += p.x[ 0 ];  
    x[ 1 ] += p.x[ 1 ];  
    x[ 2 ] += p.x[ 2 ];  
    return *this;  
}  
  
POINT & operator *= ( const double t )  
{  
    x[ 0 ] *= t;  
    x[ 1 ] *= t;  
    x[ 2 ] *= t;  
    return *this;  
}  
  
POINT operator * ( const double t ) const  
{  
    return POINT( x[ 0 ] * t, x[ 1 ] * t, x[ 2 ] * t );  
}  
  
double dot_product( const POINT &p ) const  
{  
    return ( x[ 0 ] * p.x[ 0 ] + x[ 1 ] * p.x[ 1 ] + x[ 2 ] *  
p.x[ 2 ] );  
}  
  
POINT cross_product( const POINT &p ) const  
{  
    return POINT( x[ 1 ] * p.x[ 2 ] - p.x[ 1 ] * x[ 2 ],  
x[ 2 ] * p.x[ 0 ] - p.x[ 2 ] * x[ 0 ],  
x[ 0 ] * p.x[ 1 ] - p.x[ 0 ] * x[ 1 ] );  
}  
  
POINT operator / ( double p ) const  
{  
    return ((*this) * ( 1 / p ));  
}  
  
POINT operator - ( const POINT &p ) const  
{  
    return POINT( x[ 0 ] - p.x[ 0 ],  
x[ 1 ] - p.x[ 1 ],  
x[ 2 ] - p.x[ 2 ] );  
}  
  
POINT operator + ( const POINT &p ) const  
{  
    return POINT( x[ 0 ] + p.x[ 0 ],  
x[ 1 ] + p.x[ 1 ],  
x[ 2 ] + p.x[ 2 ] );  
};
```

```

}

private:
double x[ 3 ];
};

/////
/////
/////
/////

struct PIXEL
{
PIXEL( double dist = MAXIMUM_D_VALUE )
{
distance = dist;
}

PIXEL( double dist, POINT &g, POINT &p, POINT &f )
{
distance = dist;

this->gouraud = g;
this->phong = p;
flat = f;
}

inline POINT operator [] ( short index )
{
return ( index == 0 ) ? this->gouraud : ( index == 1 ) ?
this->phong : flat;
}

double distance;
POINT gouraud;
POINT phong;
POINT flat;
};

/////
/////
/////
/////

struct AVERAGE_INCIDENT
{
AVERAGE_INCIDENT()
{
for ( int i = 0; i < 6; i++ )
element[ i ] = -1;

total = -1;

next = NULL;
}

~AVERAGE_INCIDENT()
{
if ( next != NULL )
delete next;
};

int & operator [] ( int index )
{
return ( index > 5 ) ? (*next)[ index - 6 ] : element[
index ];
}

void addElement( int e )
{
if ( total++ == 5 )
next = new AVERAGE_INCIDENT;

if ( total > 5 )
{
next->addElement( e );
return ;
}

element[ total ] = e;
};

inline int size( void )
{
return total + 1;
}

private:
int element[ 6 ];

int total;

```

```

AVERAGE_INCIDENT *next;
};

/////
/////
/////
/////

/////
/////
/////
/////

static int xdim, ydim; // window

static bool beginning = true; // A little hack for speed
static bool gouraud = false; // Display Gouraud Shading?
static bool phong = false; // Display Phong Shading?

static POINT light_source; // this is the light source
you will be using

static POINT disp_right; // pixel distance sideways
(World Coord.)
static POINT disp_down; // pixel distance downwards
(World Coord.)
static POINT viewpoint; // camera
static POINT topLeft; // topleft of the view screen
static POINT midpoint; // middle of the image
static POINT v_min, v_max; // Min/Max values of x-, y-,
and z-
static unsigned long num_Ts; // # of Triangles
static unsigned long num_Vs; // # of Vertices
static POINT Fn; // Viewing Plane Normal
static double Fn_D; // D of the Viewing Plane
Equation (Ax+By+Cz+D=0)

// displacement pixel placement
double step_x, step_y;

// Triangle and Vertex tables
static unsigned int *T_table; // triangle table
static POINT *v_table; // vertex table

// Z-Buffer
static PIXEL *z_buffer;

/////
/////
/////
/////

// Gouraud Shading Variables

// Vertex Intensities
static POINT *vertex_intensities;

/////
/////
/////
/////

// Phong Shading Variables

// None Needed

/////
/////
/////
/////

// All Shading Methods Variables

// normal tables
static POINT *T_normals;
static POINT *V_normals;

// Triangles incident upon each vertex
static AVERAGE_INCIDENT *incident;

/////
/////
/////
/////

/////
/////
/////
/////

/////
/////
/////
/////

// forwarding functions
POINT screen_coordinates( POINT &p );
void Write_Z( int index, double z, POINT &gouraud_color,
POINT &phong_color, POINT &flat_color );
double Read_Z( int index );

```





```
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
void create_vertex_intensities( void )
{
    // the direction of the light to the surface
    POINT direction;

    for ( int i = 0; i < num_Vs; i++ )
    {
        double x = intensity_at_normal( V_normals[ i ],
                                        v_table[ i ] );
        vertex_intensities[ i ] = POINT( x, x, x );
    }
}

////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
void Write_Z( int buffer_index, double z_value, POINT &g,
             POINT &p, POINT &f )
{
    z_buffer[ buffer_index ] = PIXEL( z_value, g, p, f );
}

////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
double Read_Z( int index )
{
    return z_buffer[ index ].distance;
}

////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
double interpolate( POINT &first, POINT &second, double x,
                 double y )
{
    double change = first[ 0 ] - second[ 0 ];
    if ( change == 0 )
    {
        change = first[ 1 ] - second[ 1 ];
        if ( change == 0 )
            return first[ 2 ];

        change = ( y - first[ 1 ] ) / change;
        return change + first[ 1 ];
    }

    change = ( x - first[ 0 ] ) / change;
    return change + first[ 0 ];
}

////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
// returns a status variable
//
// status -> true = okay
//           false = line is vertical
bool line_equation( POINT &first, POINT &second, double &m,
                  double &b )
{
    // determine  $y = m * x + b$ 
    m = first[ 0 ] - second[ 0 ];

    // check for division by zero
    if ( fabs( m ) < EPSILON )
        return false;

    // determine the rest of m
    m = ( first[ 1 ] - second[ 1 ] ) / m;

    // determine b
    b = second[ 1 ] - m * second[ 0 ];
}

////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
```

```
return true;
}

////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
bool scan_line_intersect_line( POINT &v0, POINT &v1, int y,
                             POINT &intersection )
{
    // take care of the case where it doesn't intersect
    if ( ( v0[ 1 ] > y && v1[ 1 ] > y ) ||
         ( v0[ 1 ] < y && v1[ 1 ] < y ) )
        return false;

    // it DOES intersect, so let's figure it out
    double m, b;
    //  $y = mx + b$ 
    if ( !line_equation( v0, v1, m, b ) )
        // the line is vertical
        {
            intersection = POINT( v0[ 0 ], y,
                                 interpolate( v0, v1, v0[ 0 ],
                                             y ) );
            return true;
        }

    // this means the line is horizontal - we can cheat and
    // forget it.
    if ( fabs( m ) < EPSILON )
        return false;

    // temporarily use m for the x value
    m = ( y - b ) / m;
    intersection = POINT( m, y, interpolate( v0, v1, m, y ) );
    return true;
}

////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
bool scan_line_intersect( POINT *vertices, int y, int
                       *pairings,
                       POINT &A, POINT &B )
{
    int first, second;
    if ( scan_line_intersect_line( vertices[ 0 ], vertices[ 1 ],
                                   y, A ) )
    {
        first = ( vertices[ 0 ][ 1 ] < vertices[ 1 ][ 1 ] ) ? 1
        : 0;
        second = ( first + 1 ) % 2;
        pairings[ 0 ] = first;
        pairings[ 1 ] = second;

        if ( scan_line_intersect_line( vertices[ 0 ], vertices[ 2 ],
                                       y, B ) )
        {
            first = ( vertices[ 0 ][ 1 ] < vertices[ 2 ][ 1 ] )
            ? 2 : 0;
            second = ( first + 2 ) % 4;
            pairings[ 2 ] = first;
            pairings[ 3 ] = second;
            return true;
        }

        if ( !scan_line_intersect_line( vertices[ 1 ],
                                        vertices[ 2 ], y, B ) )
            return false;

        first = ( vertices[ 1 ][ 1 ] < vertices[ 2 ][ 1 ] ) ? 2
        : 1;
        second = ( first % 2 ) + 1;
        pairings[ 2 ] = first;
        pairings[ 3 ] = second;
        return true;
    }
}
```

```

if ( !scan_line_intersect_line( vertices[ 0 ], vertices[ 2
], y, A ) )
return false;
if ( !scan_line_intersect_line( vertices[ 1 ], vertices[ 2
], y, B ) )
return false;
first = ( vertices[ 0 ][ 1 ] < vertices[ 2 ][ 1 ] ) ? 2 :
0;
second = ( first + 2 ) & 4;
pairings[ 0 ] = first;
pairings[ 1 ] = second;
first = ( vertices[ 1 ][ 1 ] < vertices[ 2 ][ 1 ] ) ? 2 :
1;
second = ( first & 2 ) + 1;
pairings[ 2 ] = first;
pairings[ 3 ] = second;
return true;
}

////////////////////////////////////
POINT Intersect( POINT ffirst, POINT ssecond, int edge )
{
// this is the intersection value for either x- or y-
int clip_point = ( edge < 2 ) ? 0 :
( ( edge == 2 ) ? xdim - 1 : ydim - 1 );
// y=mx+b
double m, b;
// determine the line equation
if ( !line_equation( first, second, m, b ) )
{
// the line is vertical and that means there's only
// two clip boundaries it could possibly care
// about (top and bottom)
return POINT( first[ 0 ], clip_point,
(first[ 2 ] - second[ 2 ]) *
fabs(first[ 1 ] - clip_point) /
fabs(first[ 1 ] - second[ 1 ]));
}
// Similar to the last time, if this is zero -> then
// the line is horizontal and it matters only to
// two clip boundaries ( left and right )
if ( fabs(m) < EPSILON )
return POINT( clip_point, first[ 1 ],
(first[ 2 ] - second[ 2 ]) *
fabs(first[ 0 ] - clip_point) /
fabs(first[ 0 ] - second[ 0 ]));
// this is either x or y
int axis = ( edge & 2 == 0 ) ? 0 : 1;
// determine the intersection for y=mx+b for the clip edge
double x = ( axis == 0 ) ? clip_point : ( clip_point - b )
/ m;
double y = ( axis == 0 ) ? m * clip_point + b : clip_point;
return POINT( x, y, interpolate( first, second, x, y ) );
}

////////////////////////////////////
// Edge => 0 : left side of viewing screen
//          1 : top side of viewing screen
//          2 : right side of viewing screen
//          3 : bottom side of viewing screen
bool Inside( POINT sp, int edge )
{
if ( edge < 2 )
return ( p[ edge ] >= 0 );
else
return (( edge == 2 ) ? ( p[ 0 ] < xdim ) : ( p[ 1 ] <
ydim ));
}

////////////////////////////////////

```

```

////////////////////////////////////
// this is taken almost directly from p.128 in Foley,
// van Dam, Fainor, and Hughes
int SutherlandHodgmanPolygonClip( POINT *vertexArray )
{
vector< POINT > clipped;
POINT s, p, i;
// initial size of vertexArray
int size = 3;
// This is the clipping boundary (there are 4 sides to
// the viewing plane
for ( int k = 0; k < 4; k++ )
{
// initialize
s = vertexArray[ 2 ];
// There are always three vertices passed into our
version
// of this algorithm
for ( int j = 0; j < size; j++ )
{
p = vertexArray[ j ];
if ( Inside( p, k ) )
{
if ( Inside( s, k ) )
clipped.push_back( p );
else
{
i = Intersect( s, p, k );
clipped.push_back( i );
clipped.push_back( p );
}
}
else
{
if ( Inside( s, k ) )
{
i = Intersect( s, p, k );
clipped.push_back( i );
}
}
s = p;
}
// Since we know what we just got, let's reset it and
go to the next
// phase (next boundary)
size = clipped.size();
for ( int i = 0; i < size; i++ )
vertexArray[ i ] = clipped[ i ];
// clear out the vector
clipped.clear();
}
return size;
}

////////////////////////////////////
POINT world_coordinates( POINT sp )
{
double x = ( step_x == 0 ) ? step_x : ( p[ 0 ] / step_x ) +
topleft[ 0 ];
double y = ( step_y == 0 ) ? step_y : ( p[ 1 ] / step_y ) +
topleft[ 1 ];
return POINT( x, y, p[ 2 ] );
}

////////////////////////////////////
POINT screen_coordinates( POINT sp )
{
double x = ( step_x == MAXIMUM_D_VALUE ) ? step_x : ( p[ 0
] - topleft[ 0 ] ) * step_x;
double y = ( step_y == MAXIMUM_D_VALUE ) ? step_y : ( p[ 1
] - topleft[ 1 ] ) * step_y;
}

```

```

return POINT( x, y, p[ 2 ] );
}

////
////
////
void screen_intersection( POINT sv0, POINT sv1, POINT sv2,
                        POINT ss0, POINT ss1, POINT
                        ss2 )
{
// these are used for calculating the point of intersection
double t, denominator;

// directions of the rays (from each vertex)
POINT Rd_0( v0 - viewpoint );
POINT Rd_1( v1 - viewpoint );
POINT Rd_2( v2 - viewpoint );

// normalize the direction vectors
// (this makes sure all values are <= MAX_D_VAL)
Rd_0.normalize();
Rd_1.normalize();
Rd_2.normalize();

////
//// Determine s0

// Figure out whether or not (v0 - viewpoint) is parallel
// to the viewing plane.
denominator = Pn.dot_product( Rd_0 );

if ( fabs( denominator ) < EPSILON )
{
// this is at infinity, but since we need these points,
// we "simulate" infinity
s0 = viewpoint + Rd_0 * MAXIMUM_D_VALUE;

// temporarily re-using this variable for speed reasons
Rd_0 = ( v0 - viewpoint );
}
else
{
// Figure out t
t = - ( Pn.dot_product( viewpoint ) + Pn_D ) /
denominator;

// final determination of (x,y) of s0
s0 = viewpoint + Rd_0 * t;

// distance between eye and v0
s0[ 2 ] = v0[ 2 ];

////
//// Determine s1

denominator = Pn.dot_product( Rd_1 );

if ( fabs( denominator ) < EPSILON )
{
s1 = viewpoint + Rd_1 * MAXIMUM_D_VALUE;
Rd_1 = ( v1 - viewpoint );
}
else
{
t = - ( Pn.dot_product( viewpoint ) + Pn_D ) /
denominator;
s1 = viewpoint + Rd_1 * t;
}

s1[ 2 ] = v1[ 2 ];

////
//// Determine s2

denominator = Pn.dot_product( Rd_2 );

if ( fabs( denominator ) < EPSILON )
{
s2 = viewpoint + Rd_2 * MAXIMUM_D_VALUE;
Rd_2 = ( v2 - viewpoint );
}
else
{
t = - ( Pn.dot_product( viewpoint ) + Pn_D ) /
denominator;
s2 = viewpoint + Rd_2 * t;
}

s2[ 2 ] = v2[ 2 ];
}
}

```

```

////
////
////
void draw_triangles( void )
{
#ifdef _TA_STATS_
// This is to let you know it's working
cout << "Beginning program." << endl;
#endif // _TA_STATS_

for ( int i = 0; i < num_Ts; i++ )
{
#ifdef _TA_STATS_
// This is to let you know it's working
cout << "Processing Triangle # " << i << endl;
#endif // _TA_STATS_

// We're going to cull this triangle
if ( viewpoint.dot_product( T_normals[ i ] ) < 0 )
{
#ifdef _TA_STATS_
cout << "Triangle " << i << " facing the wrong
way. Moving on." << endl;
#endif // _TA_STATS_
continue;
}

int t_index = i * 3;
int v_index[] = { T_table[ t_index + 0 ],
                 T_table[ t_index + 1 ],
                 T_table[ t_index + 2 ] };

// The three vertices of the triangle
POINT v0( v_table[ v_index[ 0 ] ] );
POINT v1( v_table[ v_index[ 1 ] ] );
POINT v2( v_table[ v_index[ 2 ] ] );

// These are the three vertices that the triangle
projects onto
// the screen with. So ... s0 is made up of (x, y,
z), where
// (x,y) is pixel(x,y) and z is the distance between
the eye
// and that vertex. Also, they are rearranged so
that the
// vertex (s0) at the highest pixel (and if there's
a tie, the
// one on the right), the second vertex (s1) is to
the left of
// the third vertex (s2).
POINT s0, s1, s2;

// Find out the points of intersection with the screen
screen_intersection( v0, v1, v2, s0, s1, s2 );

// Z-buffer this region
draw_region( s0, s1, s2, v_index, i );
}

#ifdef _TA_STATS_
// This is to let you know it's finished working
cout << "Finished displaying." << endl;
#endif // _TA_STATS_

}

////
////
////
void draw_buffer( void )
{
#ifdef _TA_STATS_
int count = 0;
#endif // _TA_STATS_

int index = 0;

short shading = ( gouraud ) ? 0 : ( phong ) ? 1 : 2;

glBegin( GL_POINTS );
for ( int y = 0; y < ydim; y++ )
for ( int x = 0; x < xdim; x++, index++ )
if ( Read_Z( index ) < MAXIMUM_D_VALUE )
{
glColor3f( z_buffer[ index ][ shading ][ 0 ],
           z_buffer[ index ][ shading ][ 1 ],
           z_buffer[ index ][ shading ][ 2 ] );
}
}
}

```

*v\_index = tri*



```

        glVertex3f( (double) x / (double) xdim, (double)
y / (double) ydim, 0 );
#ifdef _TA_STATS_
        count++;
#endif // _TA_STATS_
    glEnd();

#ifdef _TA_STATS_
    // Light Source ( blue cross-hairs )
    POINT temp( screen_coordinates( light_source ) );

    glColor3f( 0.0, 0.0, 1.0 );
    glBegin( GL_LINES );
    glVertex3f( (temp[ 0 ] / (double) xdim) - 0.05, temp[ 1 ] /
(double) ydim, 0 );
    glVertex3f( (temp[ 0 ] / (double) xdim) + 0.05, temp[ 1 ] /
(double) ydim, 0 );
    glVertex3f( temp[ 0 ] / (double) xdim, (temp[ 1 ] /
(double) ydim) - 0.05, 0 );
    glVertex3f( temp[ 0 ] / (double) xdim, (temp[ 1 ] /
(double) ydim) + 0.05, 0 );

    // Positive x and y directions ( x -> light blue, y ->
light green )
    glColor3f( 0.0, 0.5, 1.0 );
    temp = POINT( 0, 0, 0 );
    temp = screen_coordinates( temp );
    glVertex3f( (temp[ 0 ] / (double) xdim) + 0.00, (temp[ 1 ] /
(double) ydim), 0 );
    temp = POINT( 1, 0, 0 );
    temp = screen_coordinates( temp );
    glVertex3f( (temp[ 0 ] / (double) xdim), (temp[ 1 ] /
(double) ydim), 0 );

    glColor3f( 0.0, 1.0, 0.5 );
    temp = POINT( 0, 0, 0 );
    temp = screen_coordinates( temp );
    glVertex3f( (temp[ 0 ] / (double) xdim), (temp[ 1 ] /
(double) ydim) + 0.00, 0 );
    temp = POINT( 0, 1, 0 );
    temp = screen_coordinates( temp );
    glVertex3f( (temp[ 0 ] / (double) xdim), (temp[ 1 ] /
(double) ydim), 0 );
    glEnd();

    // Vertices
    glPointSize( (GLfloat) 3.0 );
    glBegin( GL_POINTS );
    for ( int i = 0; i < num_Vs - 1; i++ )
    {
        temp = screen_coordinates( v_table[ i ] );
        double x = (double) (i+1) / (double) num_Vs;
        glColor3f( 1.0, x, 0.0 );
        glVertex3f( (temp[ 0 ] / (double) xdim, temp[ 1 ] /
(double) ydim, 0 );
    }
    glEnd();
    glPointSize( (GLfloat) 1.0 );
#endif // _TA_STATS_

#ifdef _TA_STATS_
    cout << "Total pixels: " << count << endl;
#endif // _TA_STATS_
}

////
////
////
////

void Axis( void )
{
    glPushMatrix();
    glLoadIdentity();
    glBegin( GL_LINES );

    // X AXIS
    glColor3f( 1.0, 0.0, 0.0 );
    glVertex3f( -1.0, 0.0, 0.0 );
    glVertex3f( 1.0, 0.0, 0.0 );

    // Y AXIS
    glColor3f( 0.0, 1.0, 0.0 );
    glVertex3f( 0.0, -1.0, 0.0 );
    glVertex3f( 0.0, 1.0, 0.0 );

    // Z AXIS
    glColor3f( 0.0, 0.0, 1.0 );
    glVertex3f( 0.0, 0.0, -1.0 );
    glVertex3f( 0.0, 0.0, 1.0 );

    glEnd();

    glPopMatrix();
}

////
////
////
////

void display( void )
{
    // This is a hack to display the screen once the first time
// through. Sorry I don't know more to answer why it
does
// that.
if ( beginning )
{
    beginning = false;
    return ;
}

    glClear( GL_COLOR_BUFFER_BIT );
    glMatrixMode( GL_MODELVIEW );

    // Draw the three axis ( x-, y-, and z- )
    Axis();

    glLoadIdentity();

    // print out our buffer
    draw_buffer();

    glFlush();
}

////
////
////
////

void reshape( int new_width, int new_height )
{
    xdim = new_width;
    ydim = new_height;

    // initialize all aspects of the
    if ( z_buffer != NULL )
        delete [] z_buffer;
    z_buffer = new PIXEL[ xdim * ydim ];

    // do this only once
    if ( beginning )
    {
        // we only need to build this table once
        create_incidence();

        // we only need to build these tables once
        create_normals();

        // now we need to do stuff for gouraud
        create_vertex_intensities();

        // create Viewing Plane Normal
        Pn = disp_down.cross_product( disp_right );
        Pn.normalize();

        // determine D (for the ray-plane intersection)
        Pn_D = - Pn.dot_product( topleft );
    }

    // computer the z_buffer
    draw_triangles();

    glViewport( 0, 0, (GLsizei) new_width, (GLsizei) new_height );

    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();

    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();

    double x = topleft[ 0 ] + ( xdim / 2 ) * disp_right[ 0 ]
+ ( xdim / 2 ) * disp_down[ 0 ];
    double y = topleft[ 1 ] + ( ydim / 2 ) * disp_right[ 1 ]
+ ( ydim / 2 ) * disp_down[ 1 ];

    gluLookAt( viewpoint[ 0 ], viewpoint[ 1 ], viewpoint[ 2 ],
x, y, 0,
-disp_down[ 0 ], disp_down[ 1 ], disp_down[ 2 ]
);

    glutPostRedisplay();
}

```

```

/////
// Read in 3 sets of vertices that make up one triangle
// Read in 3 sets of vertices that make up one triangle
// Read in 3 sets of vertices that make up one triangle
/////

void MidPoint(void)
{
    for ( int m = 0; m < 3; m++ )
    {
        v_min[ m ] = MAXIMUM_D_VALUE;
        v_max[ m ] = -MAXIMUM_D_VALUE;

        for ( int j = 0; j < num_Vs; j++ )
        for ( int m = 0; m < 3; m++ )
        {
            v_min[ m ] = ( v_table[ j ][ m ] < v_min[ m ] ) ?
v_table[ j ][ m ] : v_min[ m ];
            v_max[ m ] = ( v_table[ j ][ m ] > v_max[ m ] ) ?
v_table[ j ][ m ] : v_max[ m ];
        }

        for ( int m = 0; m < 3; m++ )
        {
            midpoint[ m ] = ( v_min[ m ] + v_max[ m ] ) * 0.5;

            double diff = ( v_max[ m ] - v_min[ m ] ) * 0.2;

            v_min[ m ] -= diff;
            v_max[ m ] += diff;
        }
    }
}

/////
// Read in 3 sets of vertices that make up one triangle
// Read in 3 sets of vertices that make up one triangle
// Read in 3 sets of vertices that make up one triangle
/////

bool readTRI( char *name )
{
    ifstream fin( name );

    v_table = NULL;
    T_table = NULL;
    T_normals = NULL;

    if ( !fin.is_open() )
    {
        cerr << "Error opening output file" << endl;
        exit(1);
    }

    // read in the viewpoint (camera placement)
    fin >> viewpoint[ 0 ] >> viewpoint[ 1 ] >> viewpoint[ 2 ];

    // read in the screen plane
    fin >> topleft[ 0 ] >> topleft[ 1 ] >> topleft[ 2 ];

    // read distance between pixels
    fin >> disp_right[ 0 ] >> disp_right[ 1 ] >> disp_right[ 2 ];
    fin >> disp_down[ 0 ] >> disp_down[ 1 ] >> disp_down[ 2 ];

    // # of triangles
    fin >> num_Ts;

    // light source
    light_source = POINT( viewpoint[ 0 ] + ( disp_right[ 0 ] *
40 ),
                        viewpoint[ 1 ] + ( disp_down[ 1 ] * 40 ),
                        viewpoint[ 2 ] );

    // set up displacement steps for later calculation of where
    // point falls within the screen.
    step_x = disp_down[ 0 ] + disp_right[ 0 ];
    step_y = disp_down[ 1 ] + disp_right[ 1 ];

    step_x = ( step_x == 0 ) ? MAXIMUM_D_VALUE : 1 / step_x;
    step_y = ( step_y == 0 ) ? MAXIMUM_D_VALUE : 1 / step_y;

    // set up the vertex table and the triangle table
    v_table = new POINT[ 3 * num_Ts ];
    T_normals = new POINT[ num_Ts ];
    T_table = new unsigned int[ 3 * num_Ts ];

    double temp[ 3 ][ 3 ];
    short flag[ 3 ];

    num_Vs = 0;

    for ( int i = 0; i < num_Ts; i++ )

```

```

        flag[ 0 ] = flag[ 1 ] = flag[ 2 ] = -1;

        // Read in 3 sets of vertices that make up one triangle
        for ( int n = 0; n < 3; n++ )
            for ( int m = 0; m < 3; m++ )
                fin >> temp[ n ][ m ];

        // Have we added this vertex before?
        for ( int m = 0; m < 3; m++ )
            for ( int j = num_Vs - 1; j >= 0; j-- )
                if ( temp[ m ][ 0 ] == v_table[ j ][ 0 ] &&
                    temp[ m ][ 1 ] == v_table[ j ][ 1 ] &&
                    temp[ m ][ 2 ] == v_table[ j ][ 2 ] )
                {
                    // flag here spr01_1.0
                    flag[ m ] = j;
                    break ;
                }

        for ( int m = 0; m < 3; m++ )
            if ( flag[ m ] == -1 )
                // we haven't added this vertex yet
                v_table[ num_Vs ] = POINT( temp[ m ][ 0 ],
temp[ m ][ 1 ],
temp[ m ][ 2 ] );

            flag[ m ] = num_Vs++;

        // Insert which indices into the v_table refer to this
        // triangle
        int j = i * 3;
        for ( int m = 0; m < 3; m++ )
        {
            // taking this out for this project
            // incident[ flag[ m ] ].push_back( i );
            T_table[ j + m ] = flag[ m ];
        }
    }

    // reduce our storage to just the # of vertices we actually
    // have
    POINT *temp_v = v_table;
    v_table = new POINT[ num_Vs ];
    for ( int i = 0; i < num_Vs; i++ )
        v_table[ i ] = temp_v[ i ];
    delete [] temp_v;

    // storage for vertex normals
    V_normals = new POINT[ num_Vs ];

    // storage for vertex intensities
    vertex_intensities = new POINT[ num_Vs ];

    // create storage for incidence
    incident = new AVERAGE_INCIDENT[ num_Vs ];

    MidPoint();

#ifdef TA_STATS
    cout << "viewpoint: ( " << viewpoint[ 0 ] << ", "
        << viewpoint[ 1 ] << ", " << viewpoint[ 2 ] << " )\n"
        << "Midpoint: ( " << midpoint[ 0 ] << ", "
        << midpoint[ 1 ] << ", " << midpoint[ 2 ] << " )\n"
        << "Minimum: ( " << v_min[ 0 ] << ", "
        << v_min[ 1 ] << ", " << v_min[ 2 ] << " )\n"
        << "Maximum: ( " << v_max[ 0 ] << ", "
        << v_max[ 1 ] << ", " << v_max[ 2 ] << " )\n"
        << "Number of Vertices: " << num_Vs
        << "\nNumber of Triangles: " << num_Ts << endl;
#endif

    return true;
}

/////
// Read in 3 sets of vertices that make up one triangle
// Read in 3 sets of vertices that make up one triangle
// Read in 3 sets of vertices that make up one triangle
/////

void init( char *name )
{
    z_buffer = NULL;

    readTRI( name );

    glClearColor( 0.0, 0.0, 0.0, 0.0 );
}

/////
// Read in 3 sets of vertices that make up one triangle
// Read in 3 sets of vertices that make up one triangle
// Read in 3 sets of vertices that make up one triangle
/////

```



## History:

- 1887 - Emile Rayed projects animation using his praxinoscope. Opens the first animation theatre.
- 1906 - J. Stuart Blackton creates *Humorous Phases of a Funny Face*, the first 2D drawn film.
  
- 1926 - Keyframing developed, inking and painting assembly lines are set up. (Disney)
- 1932 - Disney produces *Flowers and Trees*, the first color cartoon.
- 1937 - Disney produces first full-length feature, *Snow White and the Seven Dwarves*.
  
- 1967 - Micheal Noll at Bell Labs creates first computer animated stereo movie.
- 1982 - *TRON* is produced by Walt Disney
- 1982 - SGI's Iris system is introduced.
- 1985 - Wavefront, Alias, etc. find there way into the video and film production environment.
- 1989 - Introduction of Virtual Reality.



0  
43100 34026  
3