COMP 261 Lecture 14

3D Graphics 2 of 2

Victoria University of Wellington — Te Whare Wānanga o te Ūpoko o te Ika a Māui — Capital City University

Doing better than 3x3?

- Translation, scaling, rotation are different.
- Awkward to combine them.
- Use *homogeneous coordinates* :
  - Convert 3D vectors to 4D vectors
  - use 1 for the value in the 4th dimension
  - express transformations by a 4 x 4 matrix.
- Lets us combine a sequence of transformations into a single transformation
- See more here:
  http://*www.essentialmath.com/**Affine**Xforms.pps*

Affine transformations: 4D

- Translation:

$$\begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \Rightarrow \begin{pmatrix} x+\Delta x \\ y+\Delta y \\ z+\Delta z \\ 1 \end{pmatrix}$$

- Scale:

$$\begin{pmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \Rightarrow \begin{pmatrix} x \cdot sx \\ y \cdot sy \\ z \cdot sz \\ 1 \end{pmatrix}$$

- Rotation:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \Rightarrow \begin{pmatrix} 1x + 0y + 0z \\ 0x + \cos\theta y - \sin\theta z \\ 0x + \sin\theta y + \cos\theta z \\ 1 \end{pmatrix}$$

Transformations

- Apply transformation
  input: point (x, y, z, 1)
  transform T (4x4 array)

  initialise newpoint to (0,0,0,1)
  for row ← 0 to 3
  for col ← 0 to 3
  newpoint[row] += T[row, col] * point[col]

  Consistent pattern for all transformations.

To transform polygon,
apply transform to each vertex.

Combining transformations:

- Translation followed by Rotation:

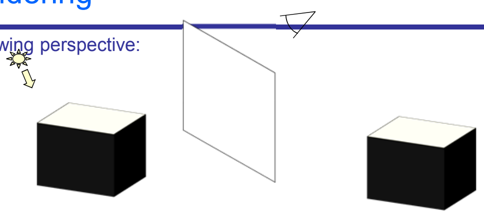
$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \sin\theta & -\cos\theta & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

- Matrix multiplication is associative:

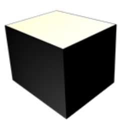
$$\begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & \sin\theta & -\cos\theta & \sin\theta\Delta y-\cos\theta\Delta z \\ 0 & \cos\theta & \sin\theta & \cos\theta\Delta y+\sin\theta\Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Rendering

- Viewing perspective:



- Assume viewer looks along z-axis
- Rotate and translate object to make this the desired view
- Remove polygons facing away from viewer



Remove hidden polygons

- Compute normal to surface:
  (vector perpendicular to surface, pointing out of the face)
- facing away from viewer if z-component of normal is +ve

  normal = edge1 × edge2 (cross product of the vectors)

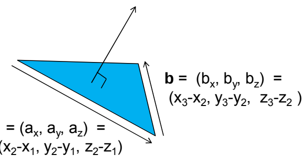
  $\mathbf{n} = (n_x,\ n_y,\ n_z)$

  $= (a_y b_z - a_z b_y,\ \ a_z b_x - a_x b_z,\ \ a_x b_y - a_y b_x)$
- Remove if

  $a_x b_y > a_y b_x$

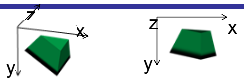
  or

  $(x_2-x_1)(y_3-y_2) > (y_2-y_1)(x_3-x_2)$



Simple Z-buffer Rendering Pipeline

input: set of polygons
viewing direction
direction of light source(s)
size of window.

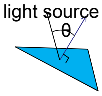


output: an image

Actions
- ✓ rotate polygons and light source so viewing along z axis
- ✓ translate & scale to fit window / clip polygons out of view
- remove any polygons facing away from viewer (normal$_z$ > 0)
- for each polygon
  - compute shading
  - work out which image pixels it will affect
  - for each pixel write shading and depth to z-buffer (retains only the shading of the closest surface)
- convert z-buffer to image

Illumination

- Light reflected from a surface depends on
  - light sources
  - reflectivity
  - matte vs shiny → diffuse or specular reflection
  - color, texture, pattern, … (variation in reflectivity)
- Simple option:
  - assume matte, uniform reflectance for red, green, blue:
    - (R_r, R_g, R_b)
  - assume some ambient light
    light ← reflectance × ambient light level
  - diffuse reflection depends on light source direction:
    light ← reflectance × light source × cos(angle of incidence)
    $\cos(\theta)$ = normal • lightdirection
    (if both unit vectors: length 1)



dot product

Computing Illumination

Matte or Diffuse material, “Lambert law”:

Amount of light reflected is proportional to the cosine of the angle between the incoming light direction, and the surface normal

We’re given the light direction

Need to calculate the surface normal

Unit Normal to surface

- Normal =

  n $= (n_x,\ n_y,\ n_z)$

  $= (a_y b_z - a_z b_y,\ \ a_z b_x - a_x b_z,\ \ a_x b_y - a_y b_x)$

  where

  $a_x = (x_2\text{-}x_1)$ $a_y = (y_2\text{-}y_1)$ $a_z = (z_2\text{-}z_1)$

  $b_x = (x_3\text{-}x_2)$ $a_y = (y_3\text{-}y_2)$ $b_z = (z_3\text{-}z_2)$

- Unit normal (perpendicular vector of length 1) =

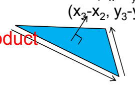
  $$\frac{n}{|n|} = \left(\frac{n_x}{|n|}, \frac{n_x}{|n|}, \frac{n_x}{|n|}\right)$$

  where

  $$|n| = \sqrt{{n_x}^2 + {n_y}^2 + {n_z}^2}$$

Cross product

b = (b_x, b_y, b_z) =
$(x_3\text{-}x_2,\ y_3\text{-}y_2,\ z_3\text{-}z_2)$



a = (a_x, a_y, a_z) =
$(x_2\text{-}x_1,\ y_2\text{-}y_1,\ z_2\text{-}z_1)$

$$\mathbf{n}/|\mathbf{n}| = (n_x/|n|,\ n_y/|n|,\ n_z/|n|)$$

$$|\mathbf{n}| = \sqrt{n_x^2 + n_y^2 + n_z^2}$$

Computing Illumination

input: **n** = (n_x, n_y, n_z) *surface normal unit vector*

d = (d_x, d_y, d_z) *light direction unit vector*

(a_r, a_g, a_b) *ambient light level*

(R_r, R_g, R_b) *reflectance in each colour*

(I_r, I_g, I_b) *intensity/colour of incident light*

output: (O_r, O_g, O_b)

actions

costh ← **n • d** [$= (n_x d_x + n_y d_y + n_z d_z)$]

for c in red, green, blue:

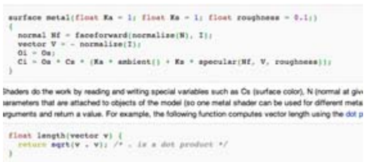
$O_c \leftarrow (a_c + I_c \times \text{costh}) \times R_c$

Further reading…for geeks!

- Shading languages:
  - Renderman shading language
  - GLSL
  - Adobe Pixel Bender
  - Playstation Shader Language
  - Etc…

Further reading…for geeks!

- Shading languages:
  - Syntax similar to C or Java
  - Built in variables like N (= normal), I (= eye)
  - Special operations like . (= dot product)



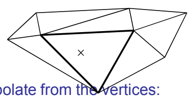
```
surface metal(float Ka = 1; float Ks = 1; float roughness = 0.1;)
{
  normal Nf = faceforward(normalize(N), I);
  vector V = - normalize(I);
  Oi = Os;
  Ci = Os * Cs * (Ka * ambient() + Ks * specular(Nf, V, roughness));
}
```

Shaders do the work by reading and writing special variables such as Cs (surface color), N (normal at giv
parameters that are attached to objects of the model (so one metal shader can be used for different meta
arguments and return a value. For example, the following function computes vector length using the dot p

```
float length(vector v) {
  return sqrt(v . v); /* . is a dot product */
}
```

Shading

- Light reflected from a polygon:
  - could be uniform (if assume each polygon is a flat, uniform surface)
    - ⇒ compute once for whole polygon
  - could vary across surface (if polygons approximate a curved surface)



  - Can interpolate from the vertices:
    - use "vertex normals" (average of surfaces at vertex)
    - either interpolate shading from vertices
    - or interpolate normals from vertices and compute shading
- What about shadows and reflected light from other sources
  - ray tracing!! expensive, we will ignore it