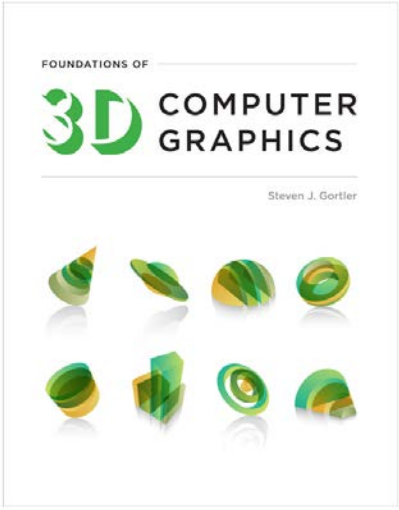
CS482: Interactive Computer Graphics

Min H. Kim
KAIST School of Computing

Reference Books

- [Basic]
  Steven J. Gortler (2012)
  Foundations of 3D Computer Graphics, MIT Press (available from the KAIST library)

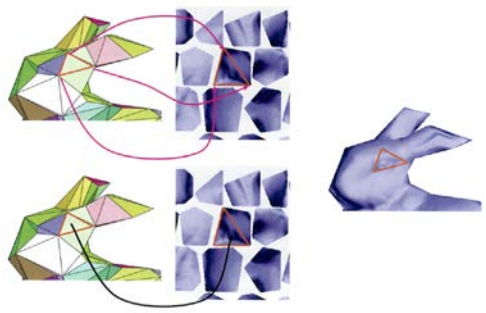


Chapter 15

TEXTURE MAPPING

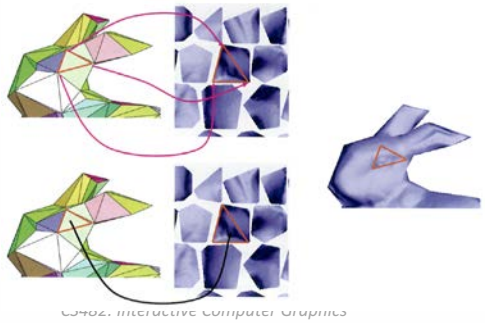
Texture mapping

- We have already seen and used texture mapping
- In basic texturing, we simply 'glue' part of an image onto a triangle by specifying texture coordinates at the three vertices.



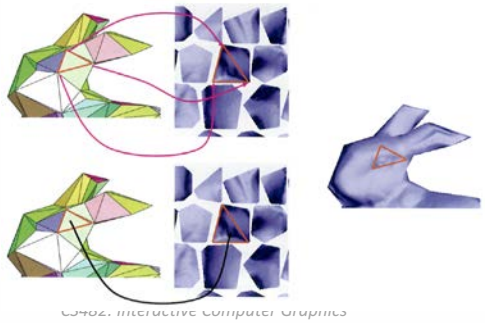
Texture mapping

- Varying variables are used to store texture coordinates.
- In this simplest incarnation, we just fetch r,g,b values from the texture and send them directly to the frame buffer.



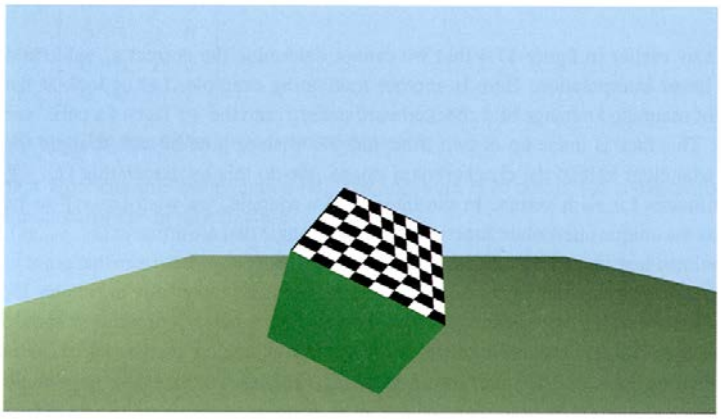
Texture mapping

- Alternatively, the texture data could be interpreted as, say, the diffuse material color of the surface point, which would then be followed by the diffuse material computation.

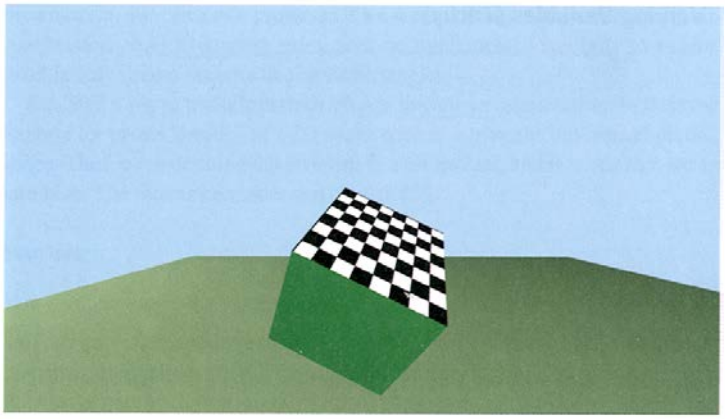


Wrong representation of texture

When texture coordinates are linearly interpolated in window coordinates, an incorrect image results.



Correct representation of texture



Motivation: texture coordinates

- Let us map a square checkerboard image onto a quad
- We break up the quad and the image each into two triangles.
- We will associate $[x_t, y_t]^t$ texture coordinates for each vertex.
- We desire that in the interior of a triangle, $[x_t, y_t]^t$ should be determined as the unique interpolant functions over the triangle that are affine in (x_o, y_o, z_o)

Motivation: texture coordinates

- Even steps on the geometry should be even steps in the texture
- If we use this interpolation and fetch the texture values we should get an expected foreshortening effect.

Affine functions

- We say a function v is an affine function in the variables x and y if it is of the form:

$$v(x,y) = ax + by + c$$

  for some constants a, b and c

- This can also be written as:

$$v = \begin{bmatrix} a & b & c \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Affine (adjective): allowing for or preserving parallel relationships (not necessarily orthogonal)
Affine space: n-dimensional coordinate system is determined by any basis of n vectors (e.g., Cartesian or Euclidean).

Affine functions

- Also called 'linear' but note the additive constant term.
- An affine function can be evaluated by plugging in x, y or by incremental evaluation along a line in x, y space.
- We already saw for example that z_n is an affine function of x_n, y_n and so was a 3D "edge function".

Affine/linear interpolation

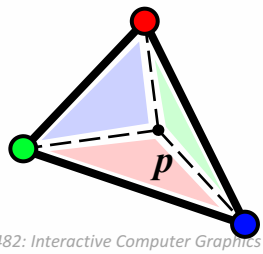
- If we are given v_i, the values of v for three (non-collinear) points in the (x, y) plane, say the vertices of a triangle,
- This determines an affine function v over the entire plane.
- In this case, we say that v is the *linear interpolant* of the values at the three vertices.
- The process of evaluating the linear interpolant of three vertices is called *linear interpolation*.
- Given the v_i one can use some simple matrix operations to solve for the $[a,b,c]$

3D affine

- We say a function v is affine in variable x, y and z if it is of the form:

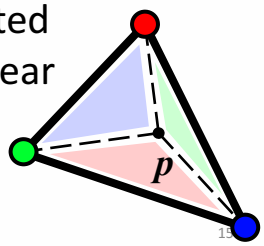
$$v(x,y,z) = ax + by + cz + d$$

- Such a function can be uniquely determined by its values at the four vertices of a tetrahedron sitting in 3D.



Triangles in 3D

- Given a triangle in 3D, suppose we specify the value of a function at its three vertices.
- There may be many functions that are affine in (x, y, z) that agree with these three values.
- But all of these functions will agree when restricted to the plane of the triangle.
- As such, we can refer to this restricted function over the triangle as the linear interplant of the vertex values.



Some more examples

- When we associate colors with vertices, it is natural to interpret our desired color field to be the unique interpolating function over the triangle that is affine in the object coordinates, (x_o, y_o, z_o) .
- During texture mapping, it is natural to interpret each of the texture coordinates, (x_t, y_t), as the unique interpolating functions over the triangle that affine in (x_o, y_o, z_o) ,

Some more examples

- As a rather self referential example, we can even think of each of the three object coordinates of a point on some triangle in 3D as affine functions in (x_o, y_o, z_o) . For example,

  $x_o(x_o, y_o, z_o) = x_o$ (i.e., $a = 1$ and $b = c = d = 0$)
- For this reason, the default semantics of OpenGL is to interpolate all varying variables as functions over triangles that are affine in (x_o, y_o, z_o) .

Some more examples

- As we will see this is equivalent to a function being affine over eye coordinates, (x_e, y_e, z_e) , but it is not equivalent to a function being affine over normalized device coordinates, (x_n, y_n, z_n) , or window coordinates.

How to evaluate the varying variables

- Recall $\begin{bmatrix} x_n w_n \\ y_n w_n \\ z_n w_n \\ w_n \end{bmatrix} = PM \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix}$
- Inverting our matrices, this implies that at each point on the triangle, we have the relation:

$$\begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix} = M^{-1}P^{-1} \begin{bmatrix} x_n w_n \\ y_n w_n \\ z_n w_n \\ w_n \end{bmatrix}$$

How to evaluate the varying variables

- Now suppose that *v* is an affine function of (x_o, y_o, z_o).
- We also make use of the obvious fact that the constant function 1 is also affine in (x_o, y_o, z_o)
- Thus for some (a,b,c,d), we have

$$\begin{bmatrix} v \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c & d \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix}$$

How to evaluate the varying variables

- And therefore:

$$\begin{bmatrix} v \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c & d \\ 0 & 0 & 0 & 1 \end{bmatrix} M^{-1}P^{-1} \begin{bmatrix} x_n w_n \\ y_n w_n \\ z_n w_n \\ w_n \end{bmatrix}$$

$$= \begin{bmatrix} e & f & g & h \\ i & j & k & l \end{bmatrix} \begin{bmatrix} x_n w_n \\ y_n w_n \\ z_n w_n \\ w_n \end{bmatrix}$$

- For some appropriate values *e…l*

How to evaluate the varying variables

- Now divide both sides by w_n and we get

$$\begin{bmatrix} \frac{v}{w_n} \\ \frac{1}{w_n} \end{bmatrix} = \begin{bmatrix} e & f & g & h \\ i & j & k & l \end{bmatrix} \begin{bmatrix} x_n \\ y_n \\ z_n \\ 1 \end{bmatrix}$$

- Conclusion: $\frac{v}{w_n}$ and $\frac{1}{w_n}$ are affine functions of normalized device coordinates.

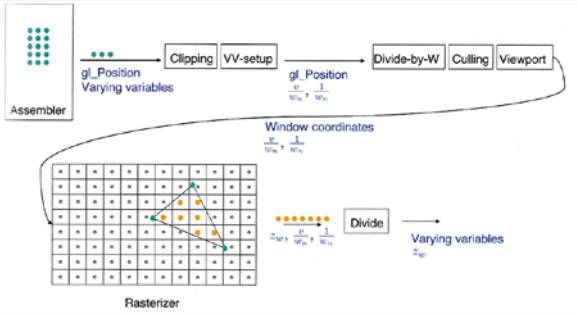
How to evaluate the varying variables

- "going sideways" we deduce that $\frac{v}{w_n}$ and $\frac{1}{w_n}$ are affine functions of window coordinates (x_w, y_w, z_w).
- "going down" we can conclude: $\frac{v}{w_n}$ and $\frac{1}{w_n}$ are both affine function of (x_w, y_w).
- Meaning we can calculate their values at each pixel, just given their values at the vertices and linear interpolation.
- The above derivation can now be thrown away.

VV in OpenGL

- Now we can see how OpenGL can perform the correct "rational linear interpolation" to calculate *v* at each pixel.
  - The vertex shader is run on each vertex, calculating clip coordinates and varying variables for each vertex.
  - Clipping is run on each triangle; this may create new vertices. Linear interpolation in clip coordinates space is run to determine the clip coordinates and varying variable values for each such new vertex.

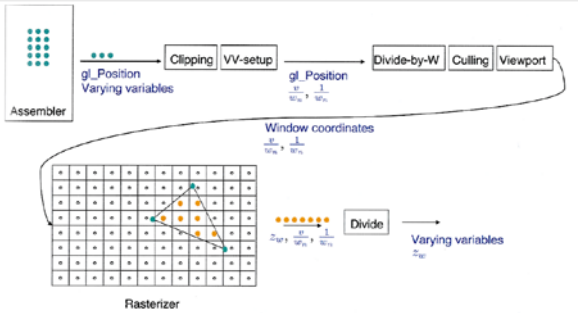
VV in OpenGL



– For each vertex, and for each varying variable *v*, OpenGL creates an internal variable $\frac{v}{w_n}$.

Additionally, for each vertex OpenGL creates one internal variable $\frac{1}{w_n}$.

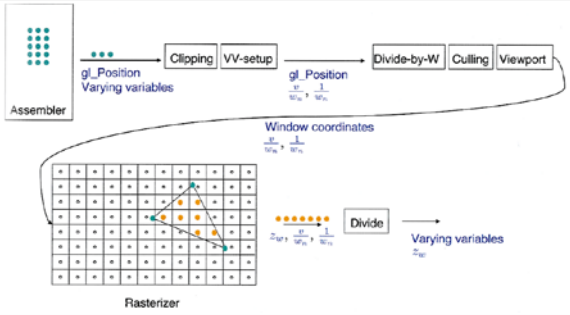
VV in OpenGL



– For each vertex, division is done to obtain the normalized device coordinates.

$$x_n = \frac{x_c}{w_c}, y_n = \frac{y_c}{w_c}, z_n = \frac{z_c}{w_c},$$

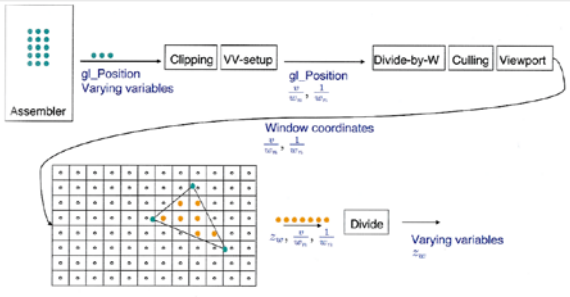
VV in OpenGL



– For each vertex, the normalized device coordinates are transformed to window coordinates.

– The $[x_w, y_w]^t$ coordinates are used to position the triangle on the screen.

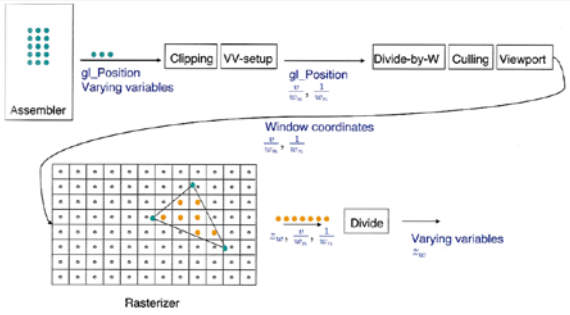
VV in OpenGL



– For every interior pixel of the triangle, linear interpolation is used to obtain the interpolated values of z_w, $\frac{v}{w_n}(\text{for all } v)$ and $\frac{1}{w_n}$.

– At each pixel, the interpolated z_w value is used for z-buffering.

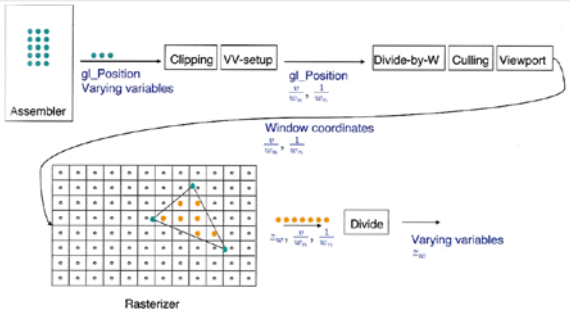
VV in OpenGL



- At each pixel, and for all varying variables, division is done on the interpolated internal variables to obtain the correct answer:

$$v = \left(\frac{v}{w_n} \right) / \frac{1}{w_n}$$

VV in OpenGL



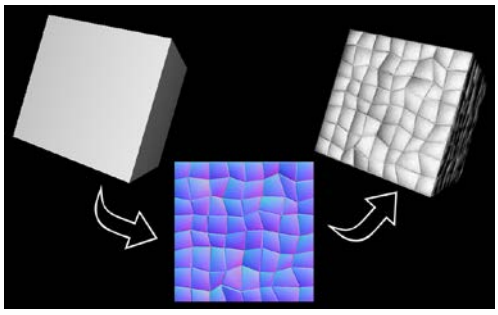
- The varying variables *v* is passed into the fragment shader.

Chapter 15

APPLICATIONS

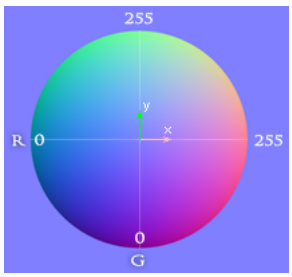
Normal mapping

- The data from a texture can also be interpreted in more interesting ways.
- In normal mapping, the r,g,b values from a texture are interpreted as the three coordinates of the normal at the point.
- This normal data can then be used as part of some material simulation,



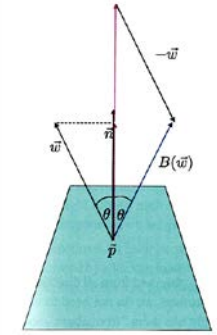
Normal mapping

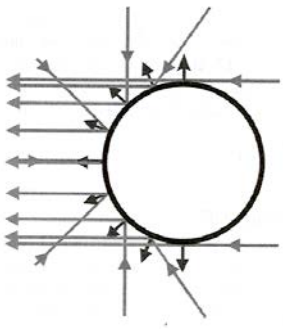
- Normal data has three coordinate values, each in the range [-1…1], while RGB textures store three values, each in the range [0…1] (0…255)
- So need some conversions:
  - R = normal_x/2.0 + 0.5;
  - normal_x = 2*R-1;
- Sometime need to deal with coordinate system conversions.
- X: red (right), Y: green (up), Z: blue (front)

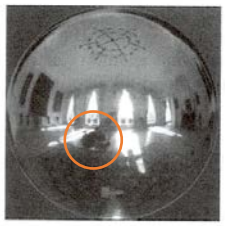


Capturing Environment Maps

- Photographing a light probe produces an environment map representing incident radiance from all directions.



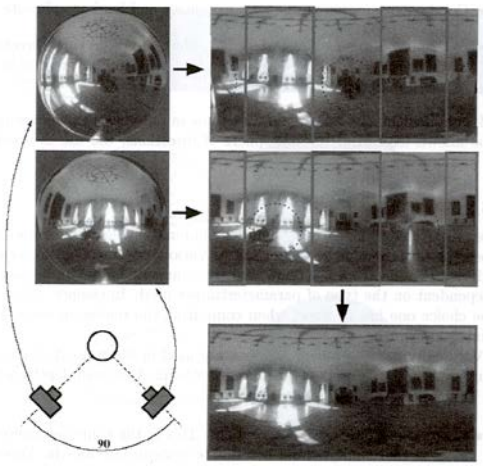




$$B(\vec{w}) = 2(\vec{w} \cdot \vec{n})\vec{n} - \vec{w}$$

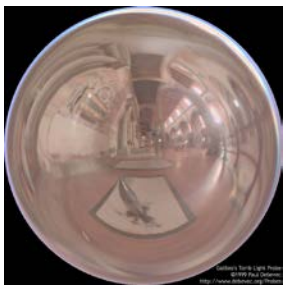
Capturing Environment Maps

- How to remove the camera from the environment map?



Examples

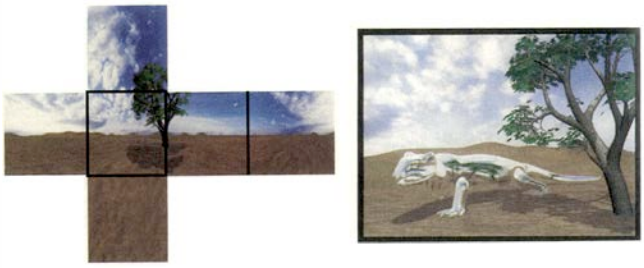
- Galileo's Tomb



http://www.pauldebevec.com/Probes/

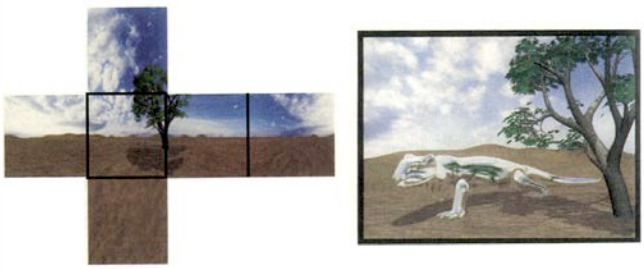
Environment cube maps

- Textures can also be used to model the environment in the distance around the object being rendered.
- In this case, we typically use 6 square textures representing the faces of a large cube surrounding the scene.



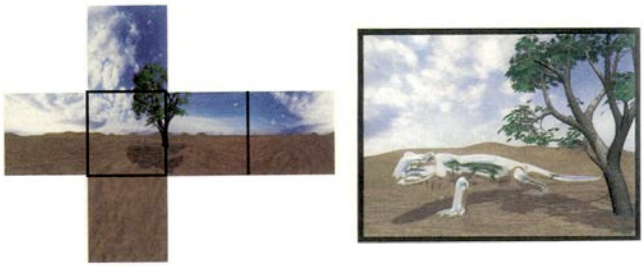
Environment cube maps

- Each texture pixel represents the color as seen along one direction in the environment.
- This is called a *cube map*. GLSL provides a cube-texture data type, samplerCube specifically for this purpose.



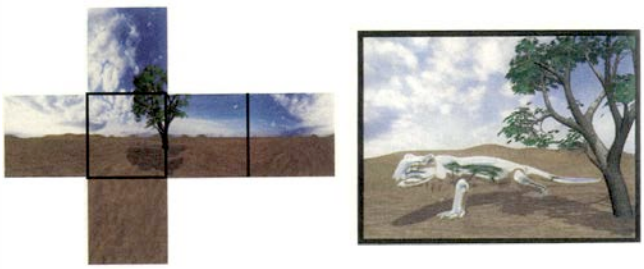
Environment cube maps

- During the shading of a point, we can treat the material at that point as a perfect mirror and fetch the environment data from the appropriate incoming direction.



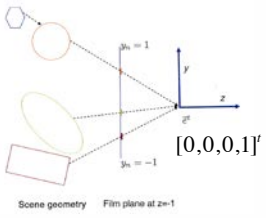
Environment map shader

- We calculate $B(\vec{v})$ in the previous lecture.
- This bounced vector will point points towards the environment direction, which would be observed in a mirrored surface.
- By looking up the cube map, using this direction, we give the surface the appearance of a mirror.



Environment map shader

- Fragment shader



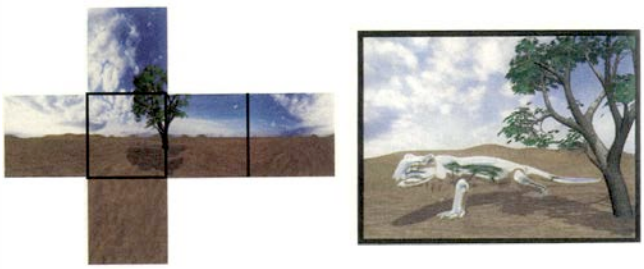

```
#version 130
uniform sampler2D uTexUnit0;
in vec3 nNormal;
in vec4 vPosition;
out vec4 fragColor;

vec3 reflect(vec3 w, vec3 n){
  return n*(dot(w,n)*2.0) - w; // bounce vector
}

void main() {
  vec3 normal = normalize(vNormal);
  vec3 reflected = reflect(normalize(vec3(-vPosition)), normal);
  vec4 texColor0 = textureCube(uTexUnit0, reflected);
  fragColor = vec4(texColor0.r, texColor0.g, texColor0.b, 1.0);;
}
```


Environment map shader

- -vPosition represents the view vector $\vec{v}$
- textureCube is a special GLSL function that takes a direction vector and returns the color stored at this direction in the cube texture map.
- Here we assume eye-coordinates, but frame changes may be needed.



Environment map shader

- This can be used for refraction.