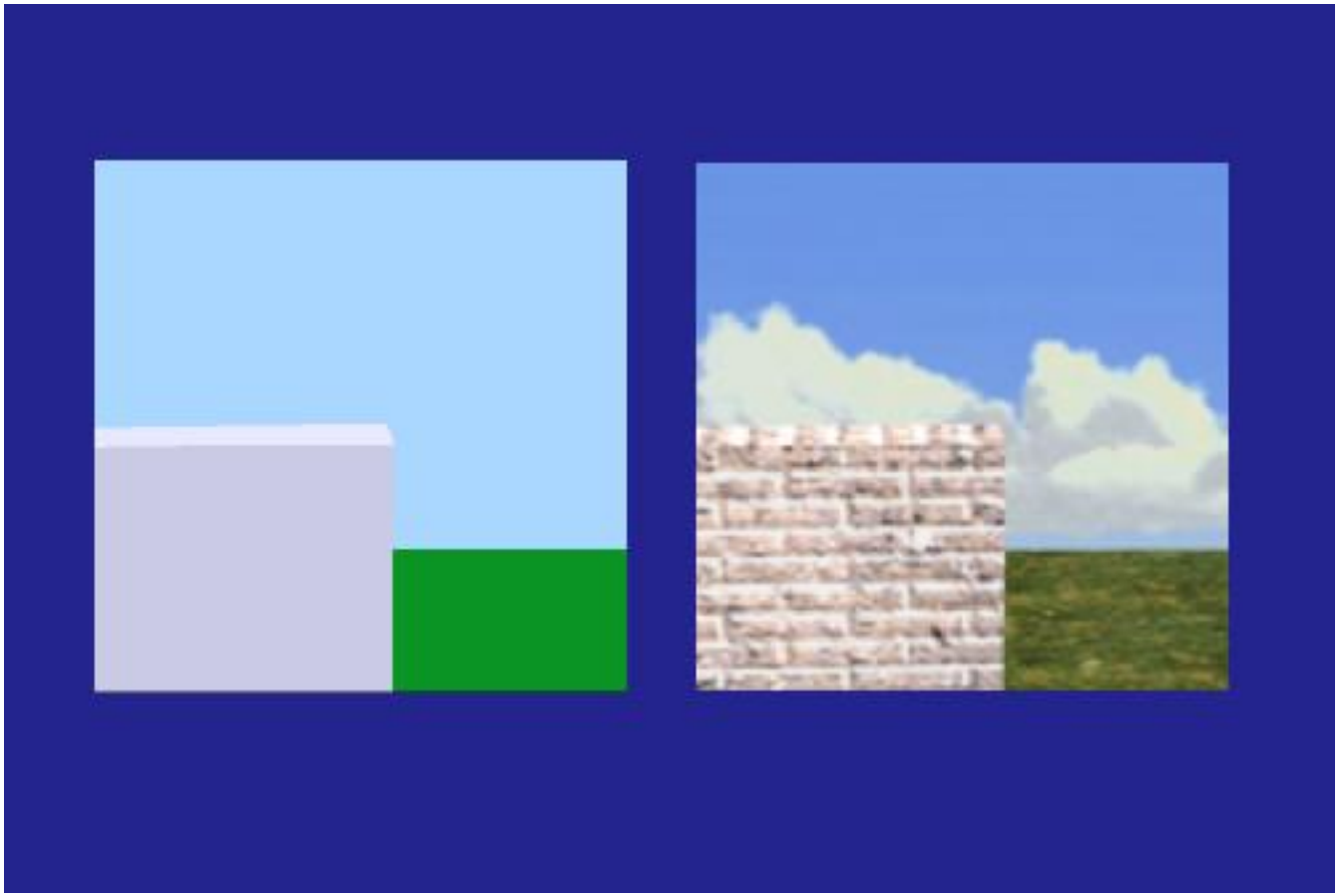
DigiPen
INSTITUTE OF TECHNOLOGY

CS300
Texture Mapping

What is Texture Mapping?

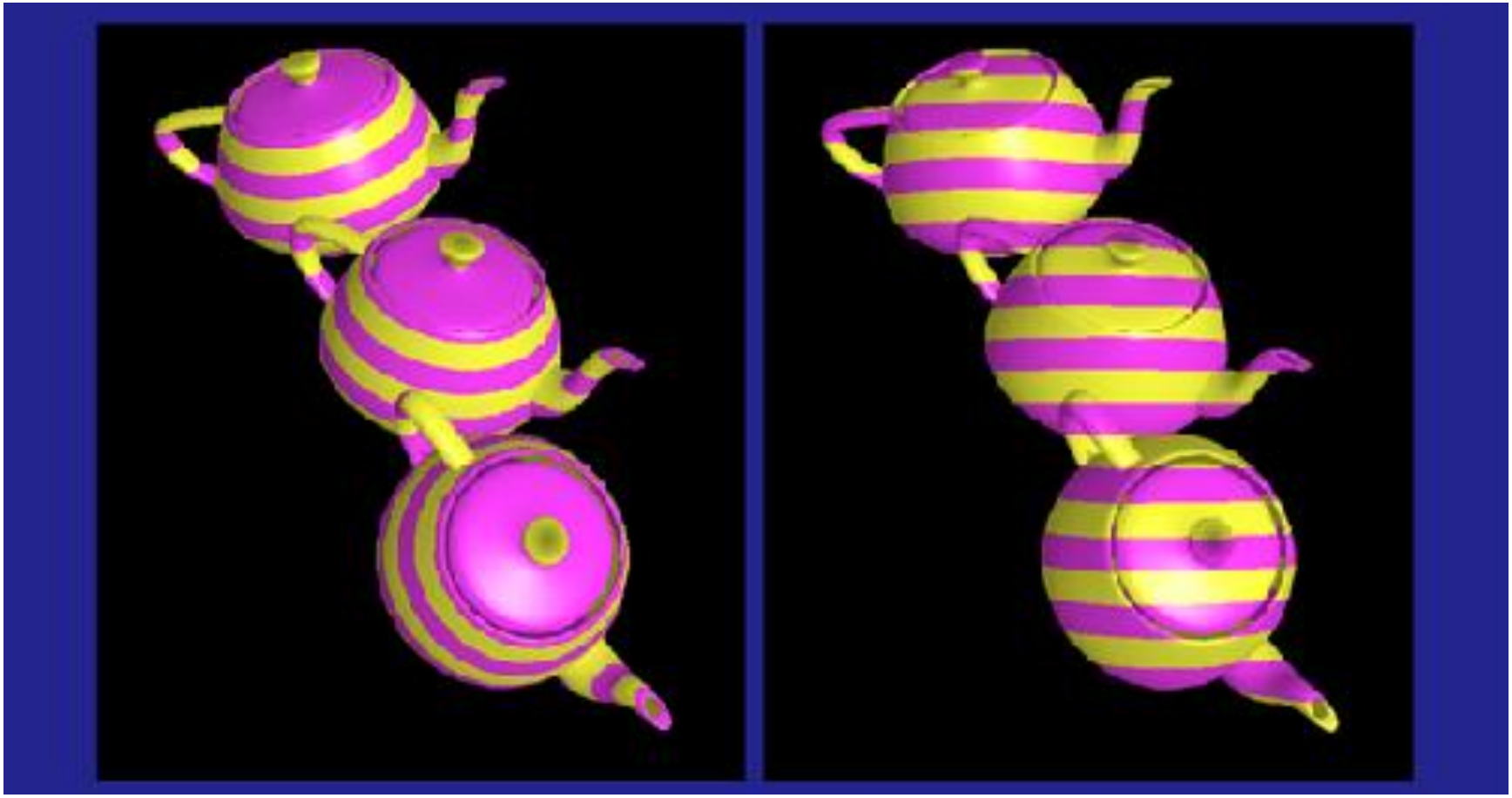


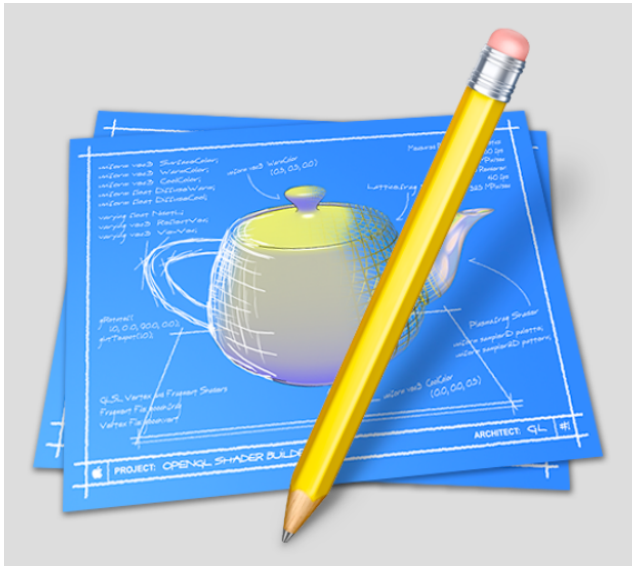
Texture Mapping

- Adding "details" to the scene
  - WITHOUT adding additional geometry
  - WITHOUT changing the existing geometry
  - Applying / draping / pasting a pattern over the geometry that makes the scene appear realistic

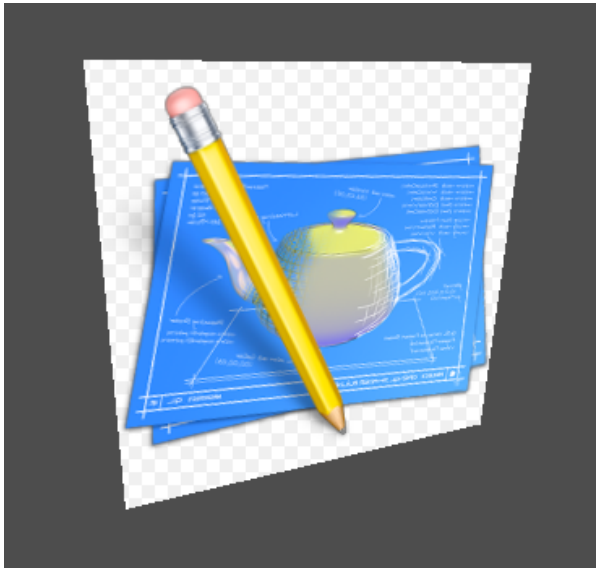
- Technical Question: In what space does the texture live?

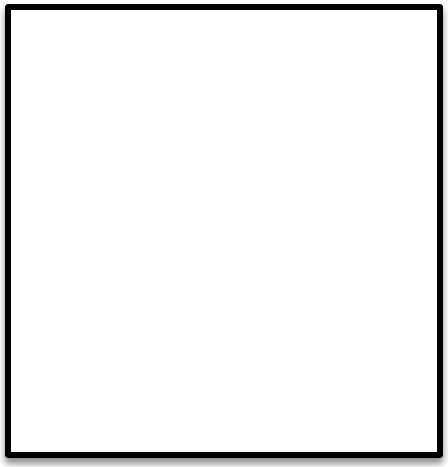
Texture maps to the object





2D Texture





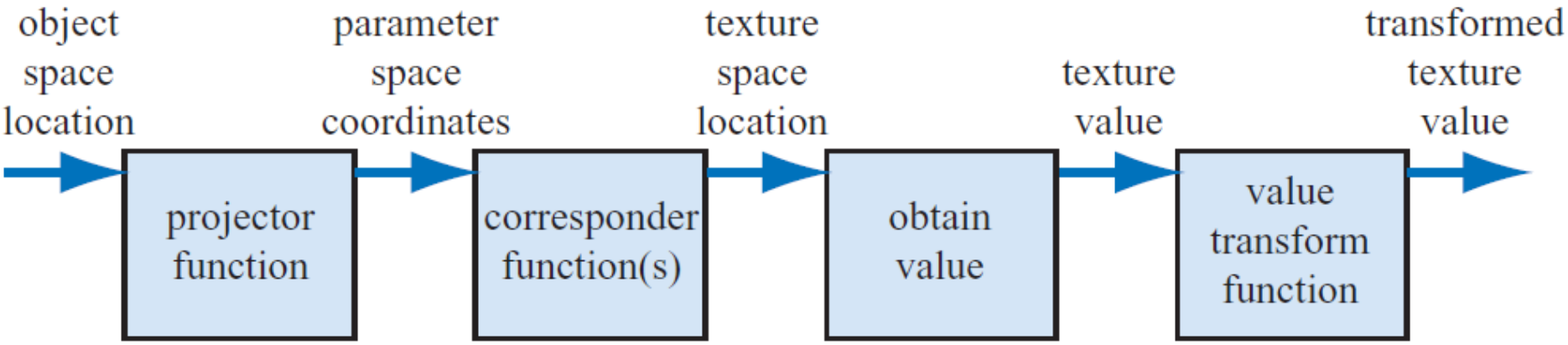
3D Shape

Textured Shape

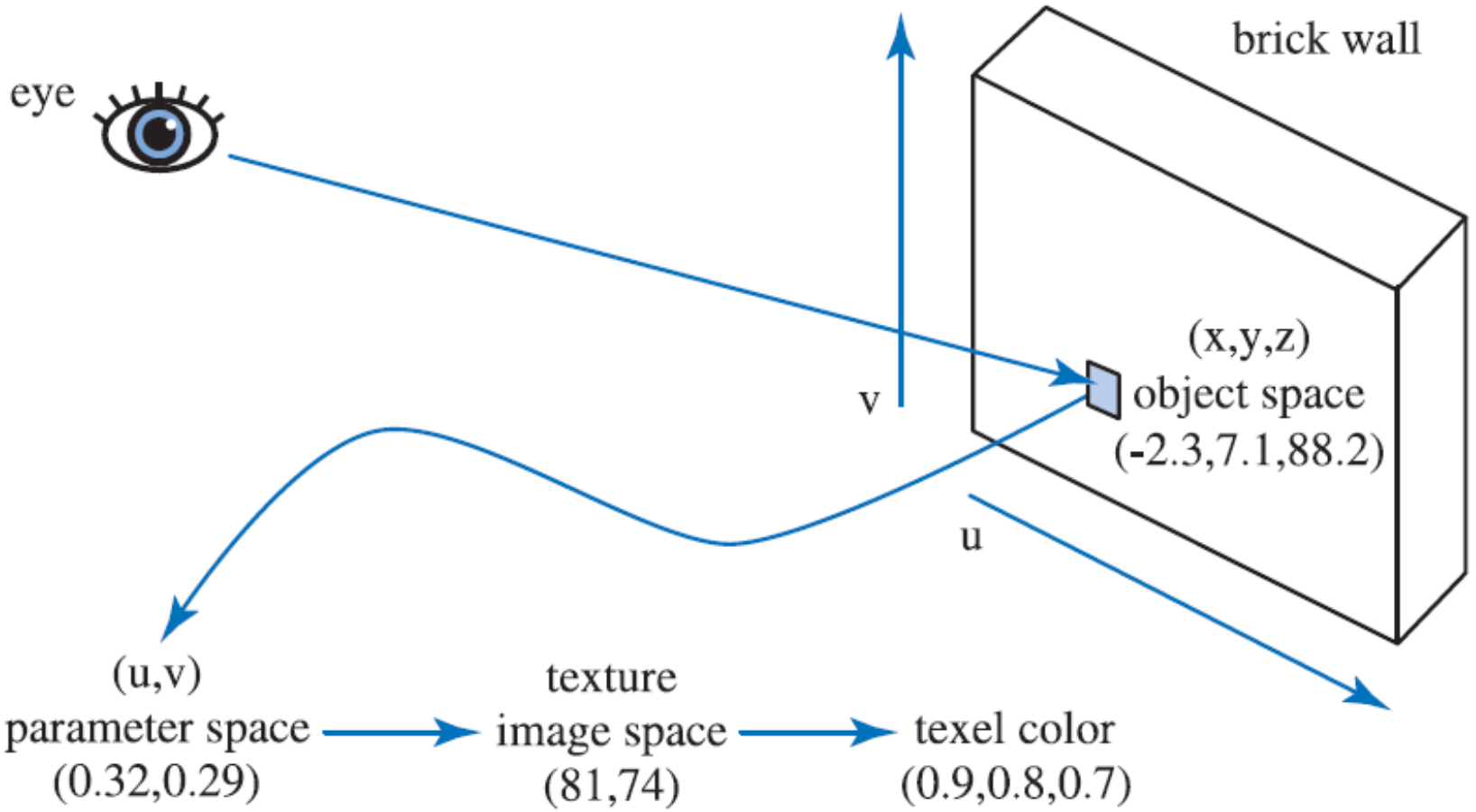
How do we apply textures to Objects?

The 1024-million dollar question ….

Texture Mapping Pipeline

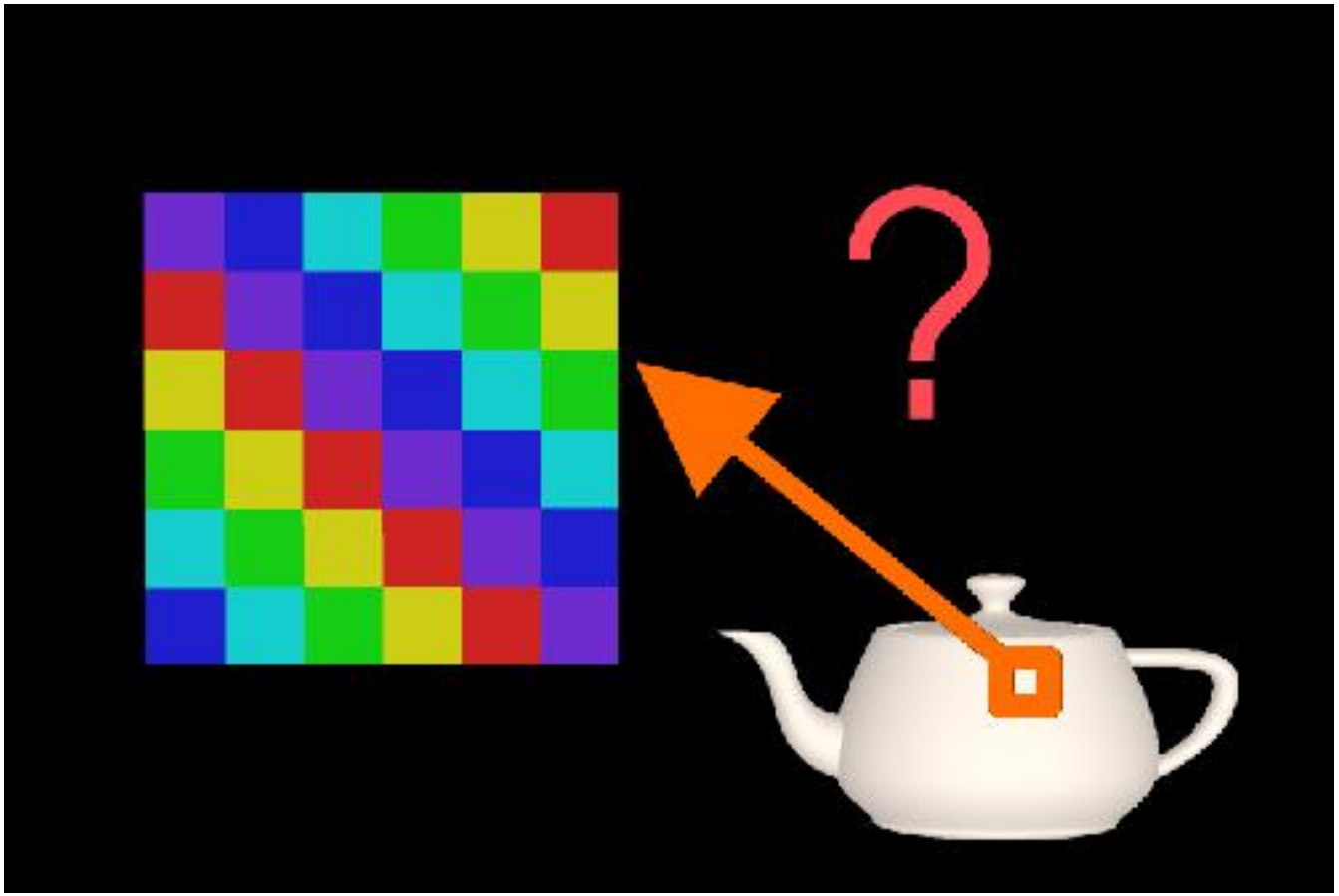


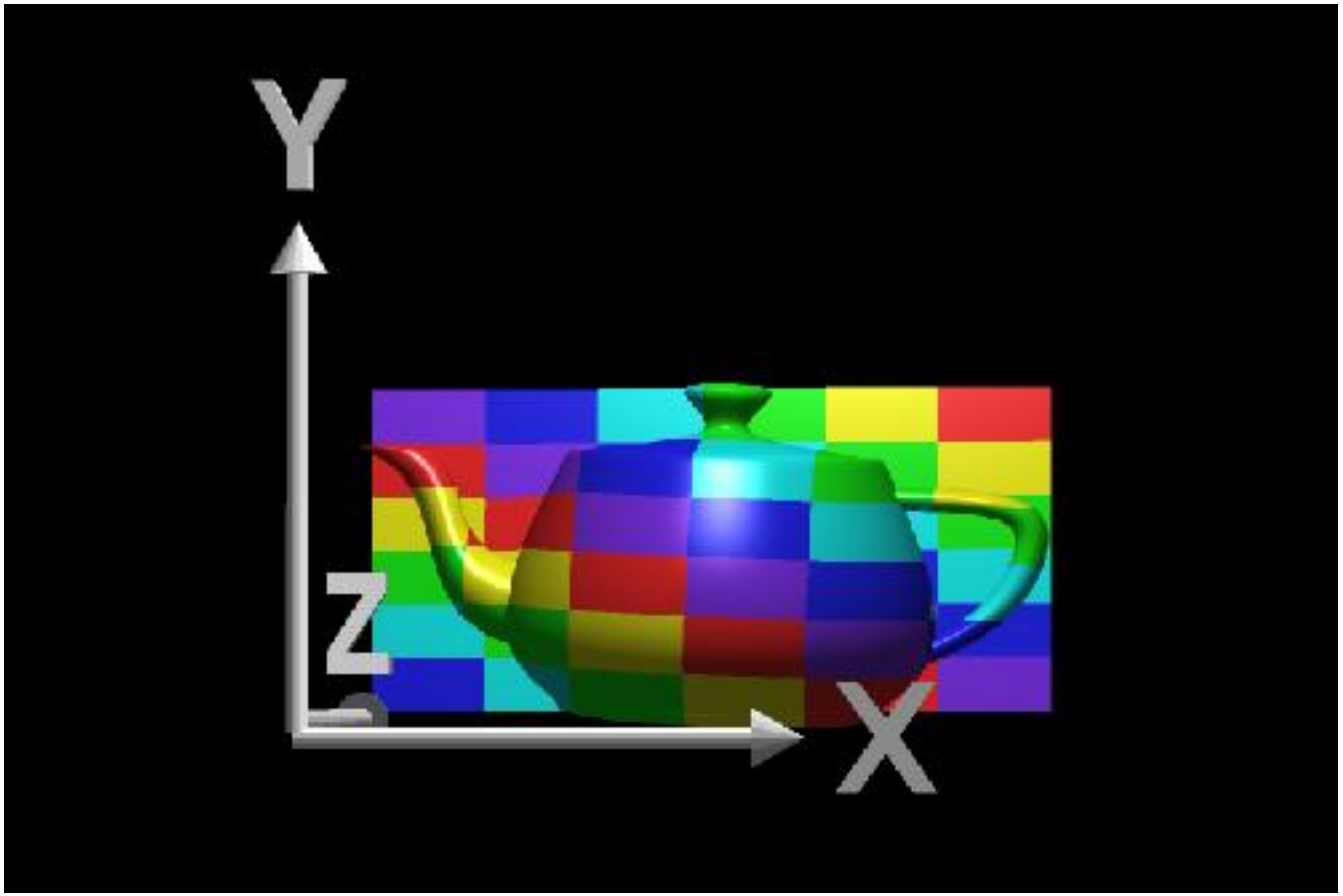
Example



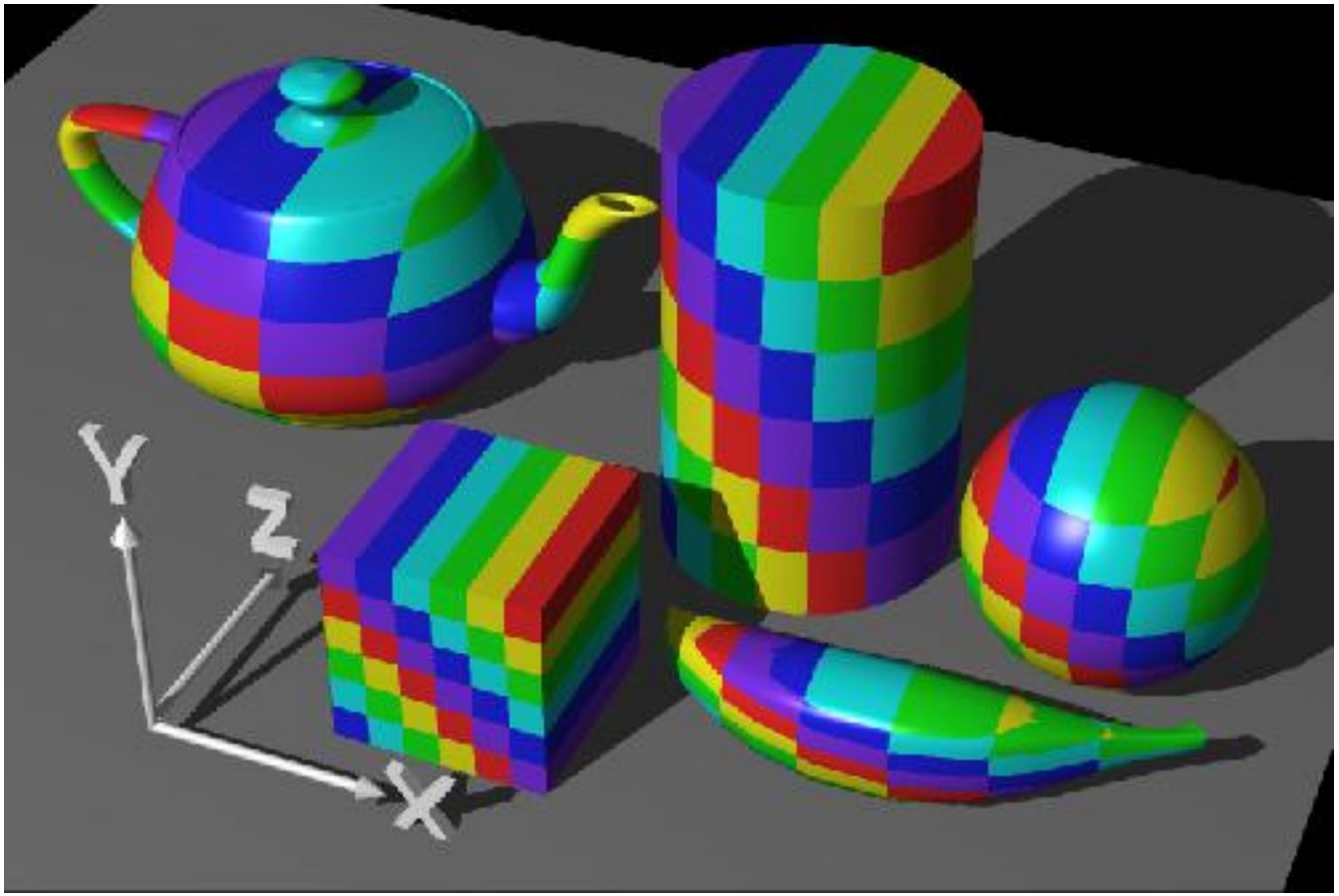
Texture Mapping Participants

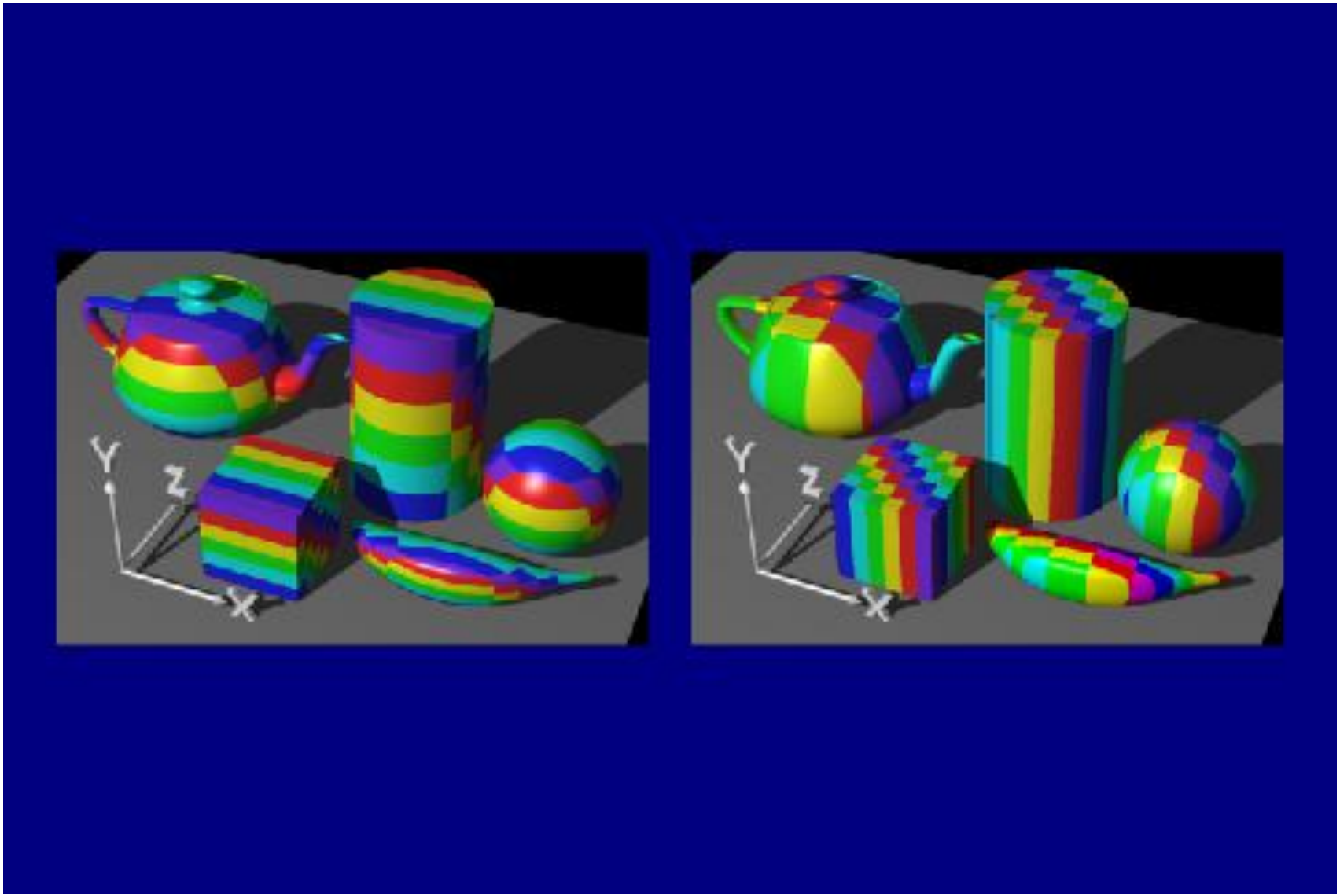
- Texture "Map"
  - Defines how a position on the object corresponds to a position in the texture
  - Individual texture elements are called "texels"
- Texture Map Shape
  - What does the map look like?
- Texture Map Entity
  - A frame of reference that uses the texture map





Planar Texture Mapping





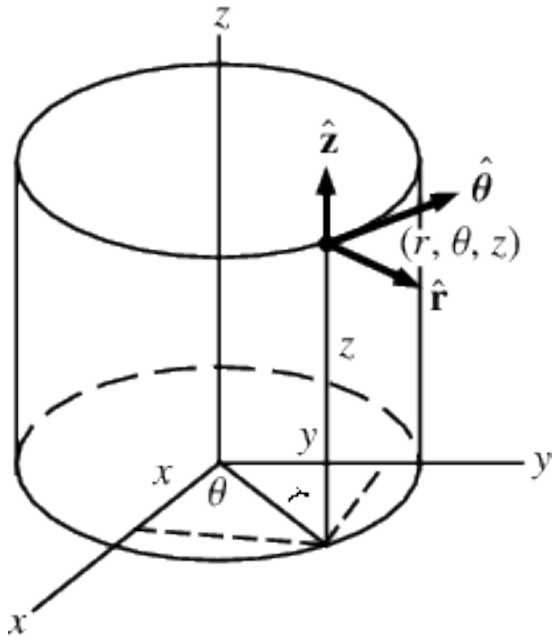
What planes are used in the above mapping?

Problems with Planar Mapping

- Highly dependent on the direction of mapping
  - Which plane is used to represent the texture?
- Not a realistic depiction of surfaces in real world
- Texture = 2D, Object = 3D
  - How do we “drape” a 2D shape on a 3D shape?

Cylindrical Texture Mapping

Conversion to Cylindrical Coordinates



$$r = \sqrt{x^2 + y^2}$$

$$\theta = tan^{-1}\left(\frac{y}{x}\right)$$

$$z = z$$

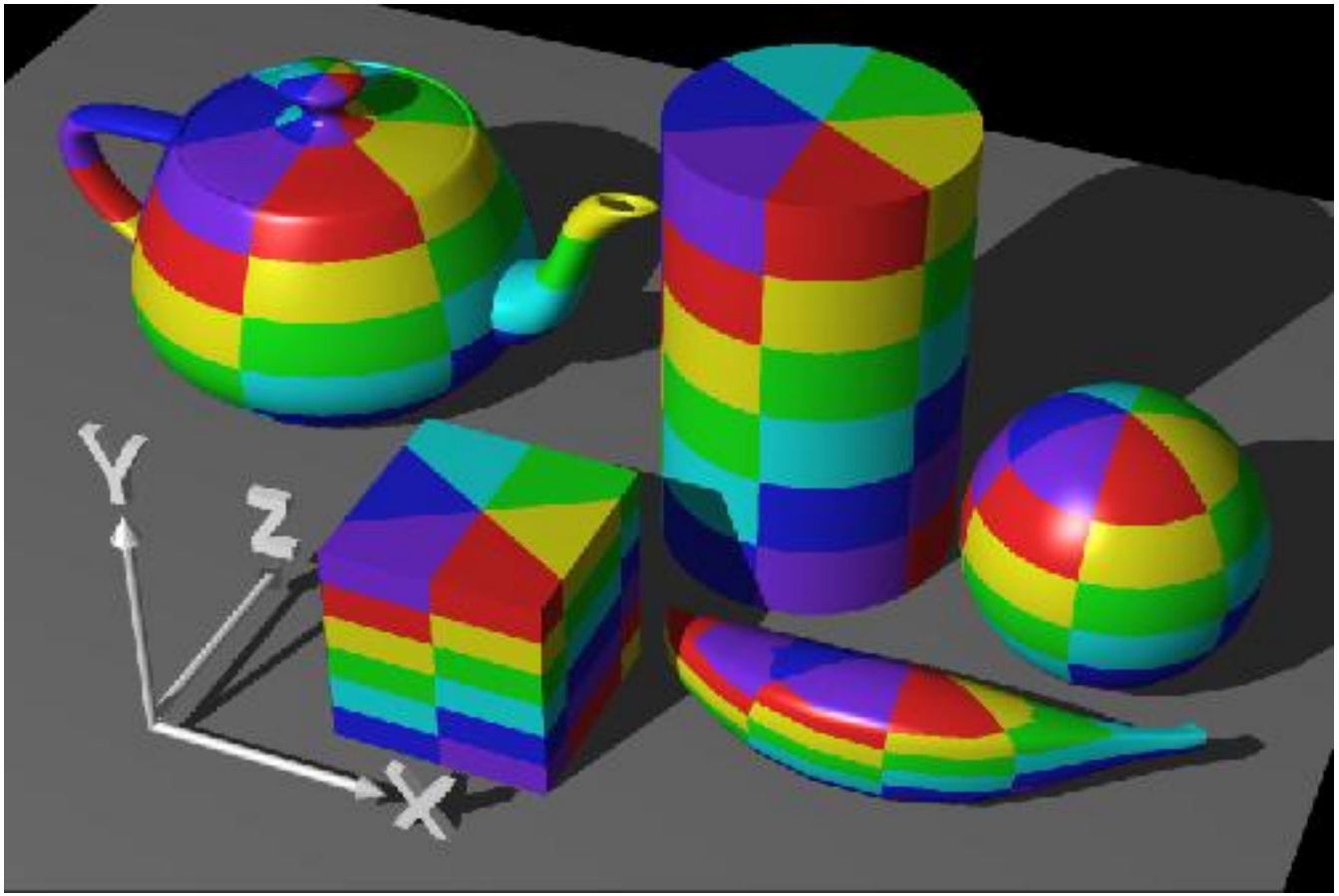
Cartesian to Cylindrical

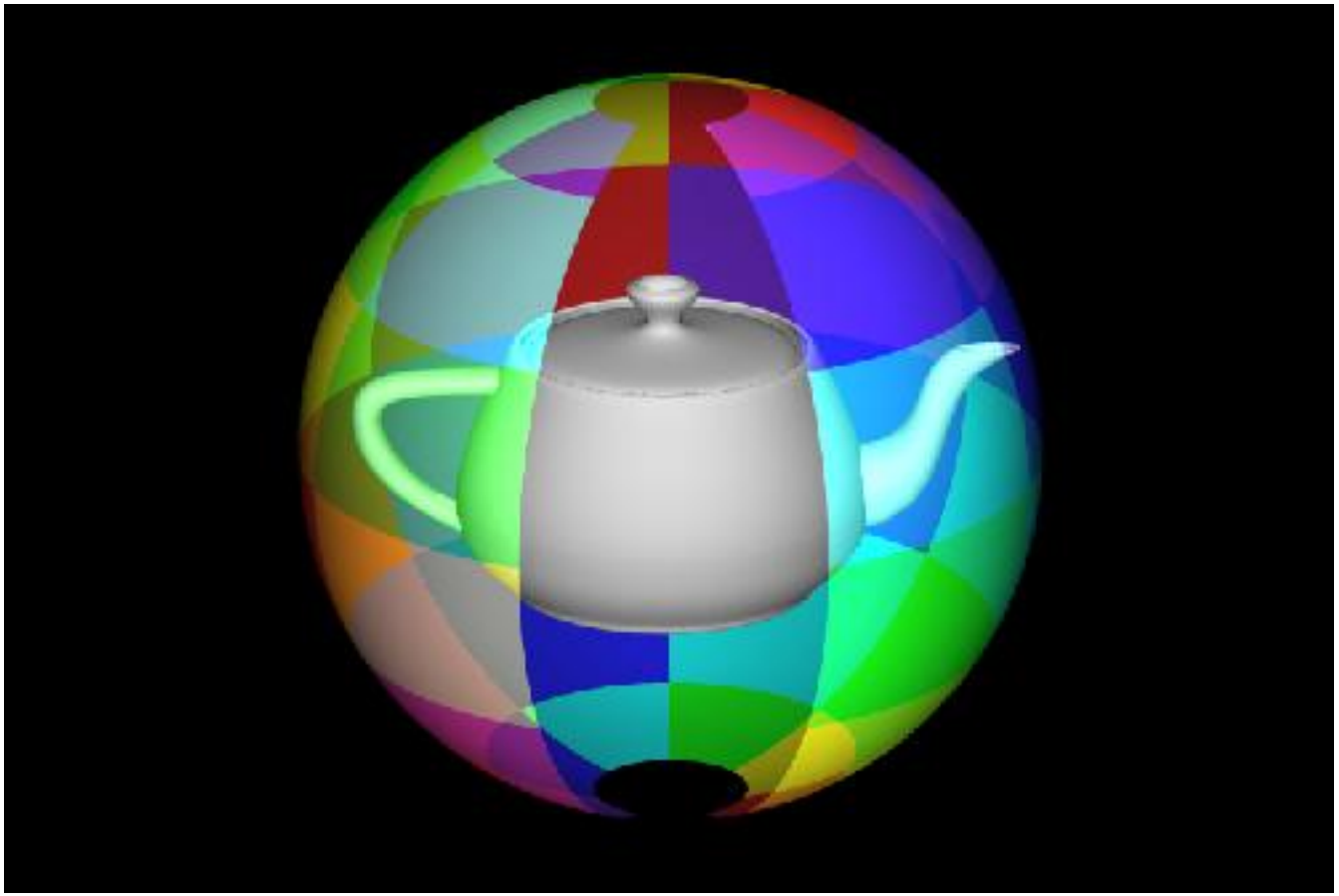
$$x = r\ cos\theta$$

$$y = r\ sin\theta$$

$$z = z$$

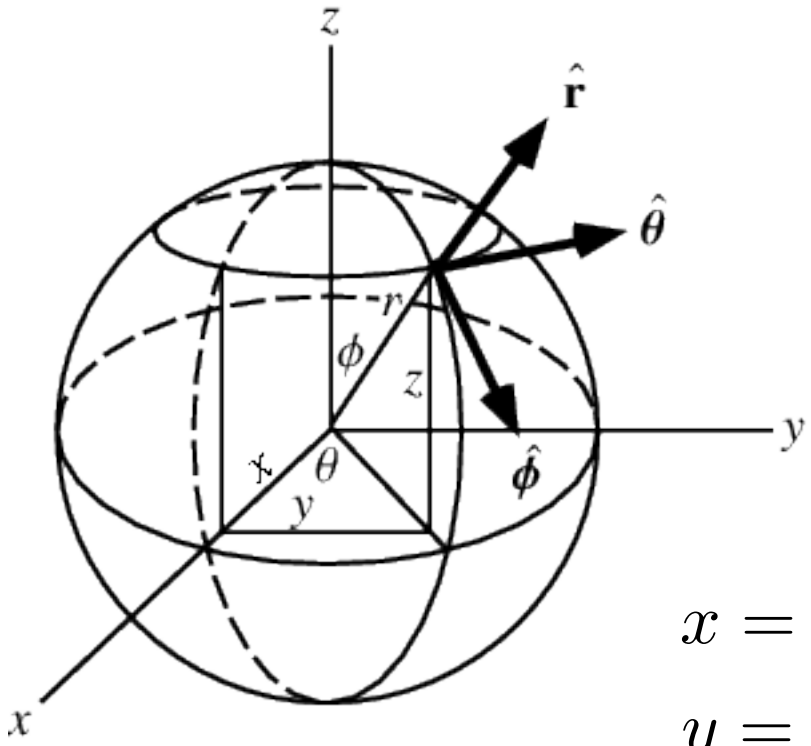
Cylindrical to Cartesian





Spherical Mapping – eliminating pie slices on the top

Cartesian to Spherical (and back)



$$r = \sqrt{x^2 + y^2 + z^2}$$

$$\theta = tan^{-1}(\frac{y}{x})$$

$$\phi = cos^{-1}(\frac{z}{r})$$

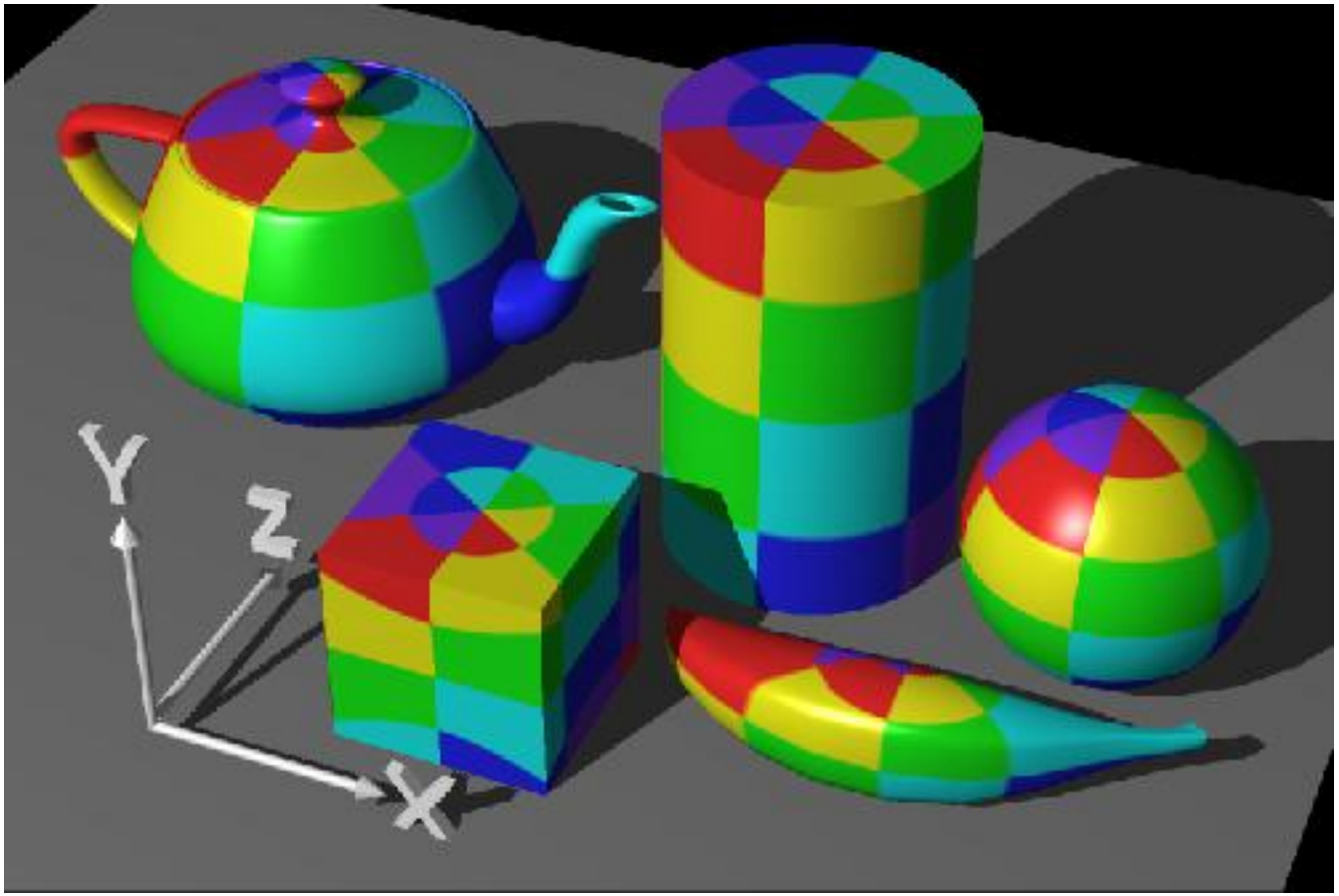
Cartesian to Spherical

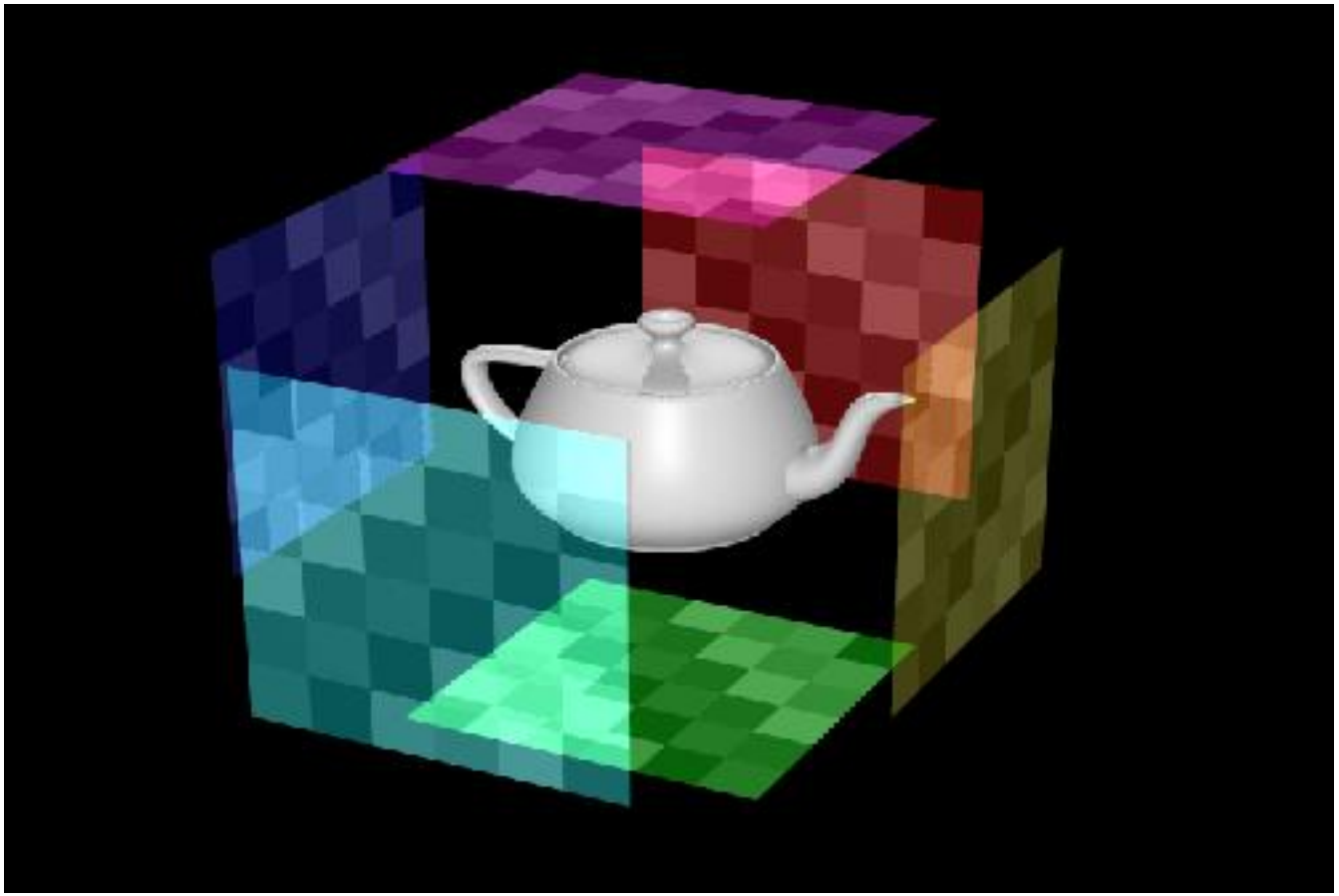
$$x = r\ cos\theta\ sin\phi$$

$$y = r\ sin\theta\ sin\phi$$

$$z = r\ cos\phi$$

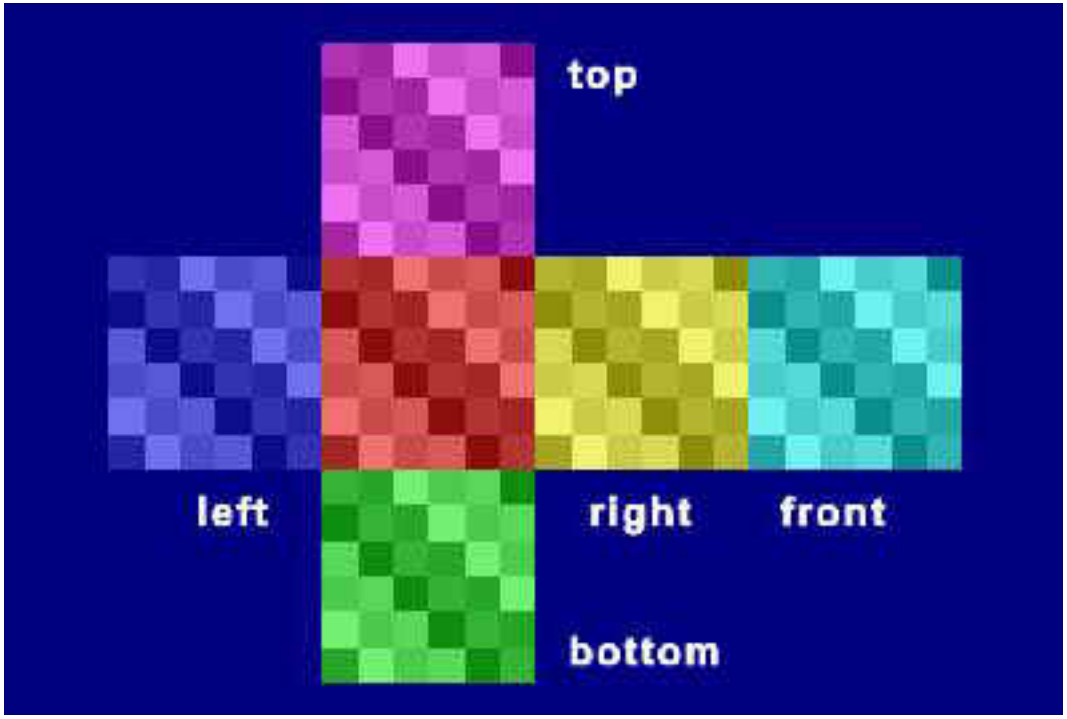
Spherical to Cartesian

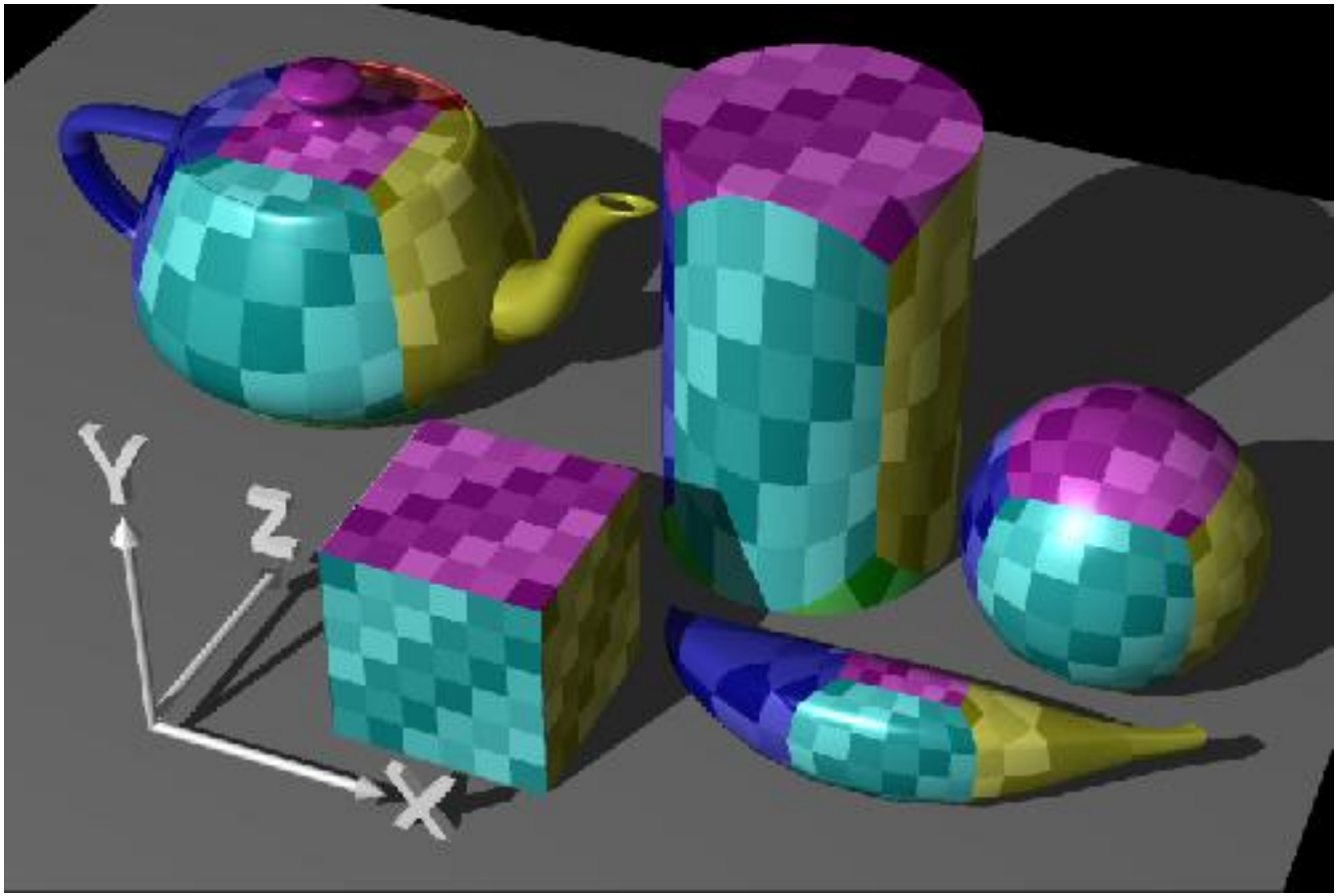




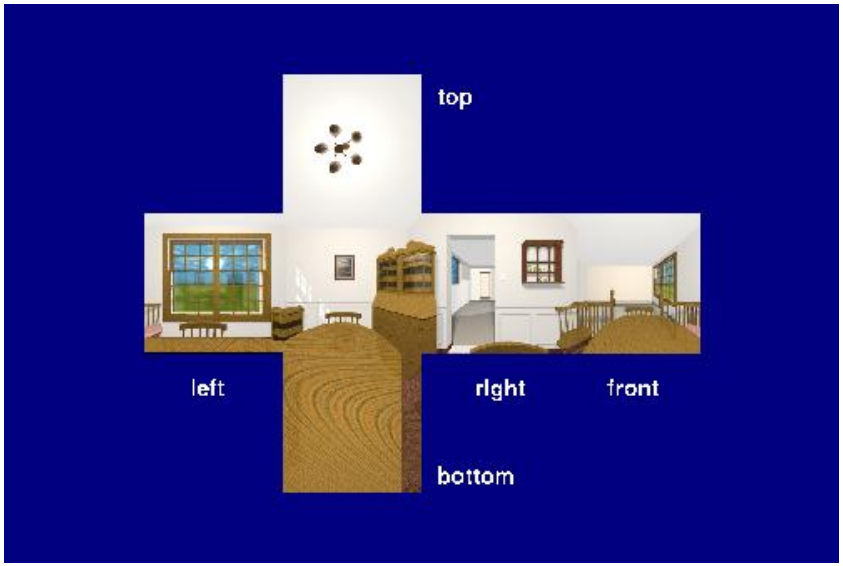
Using 6 planar texture maps

Cube Map

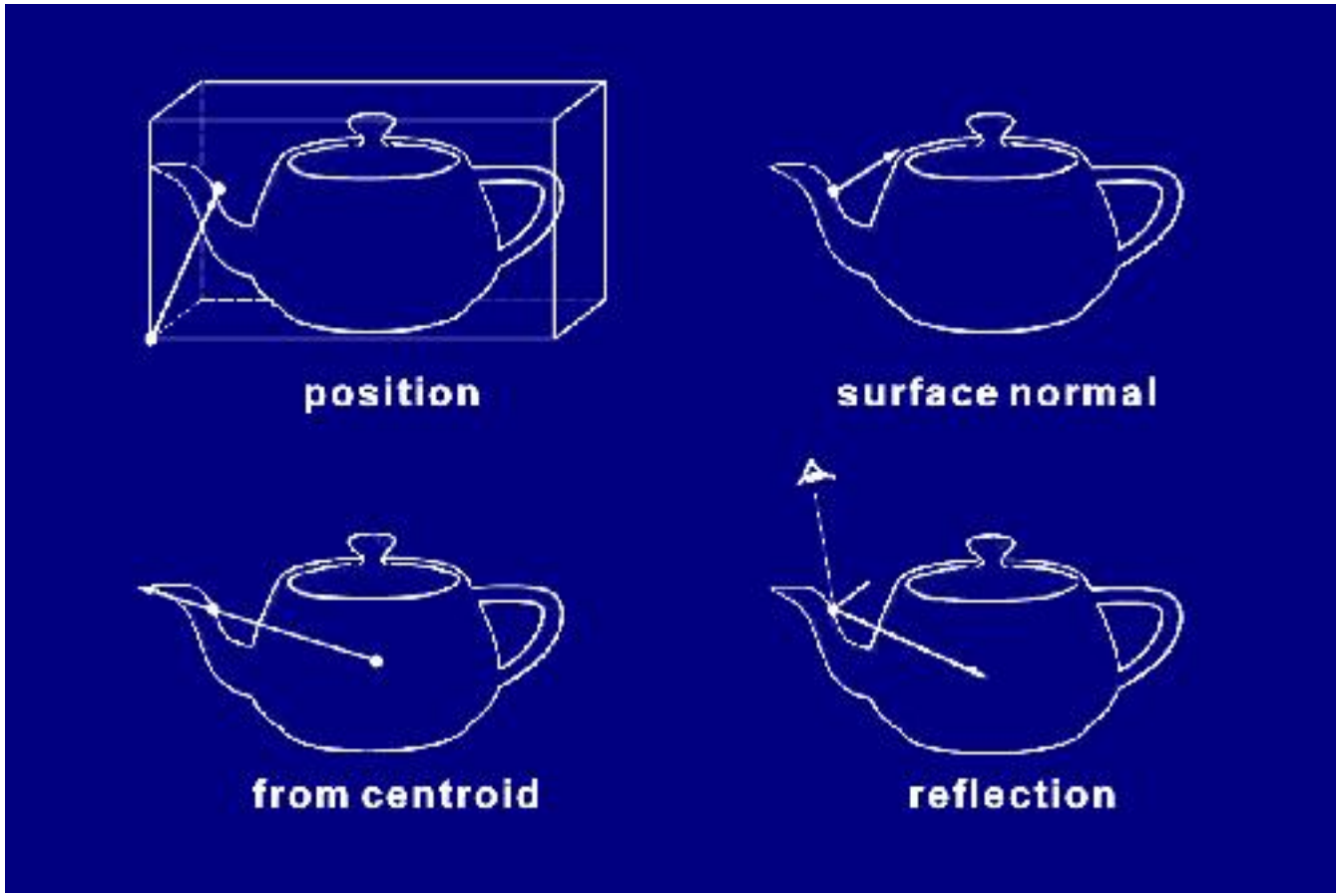




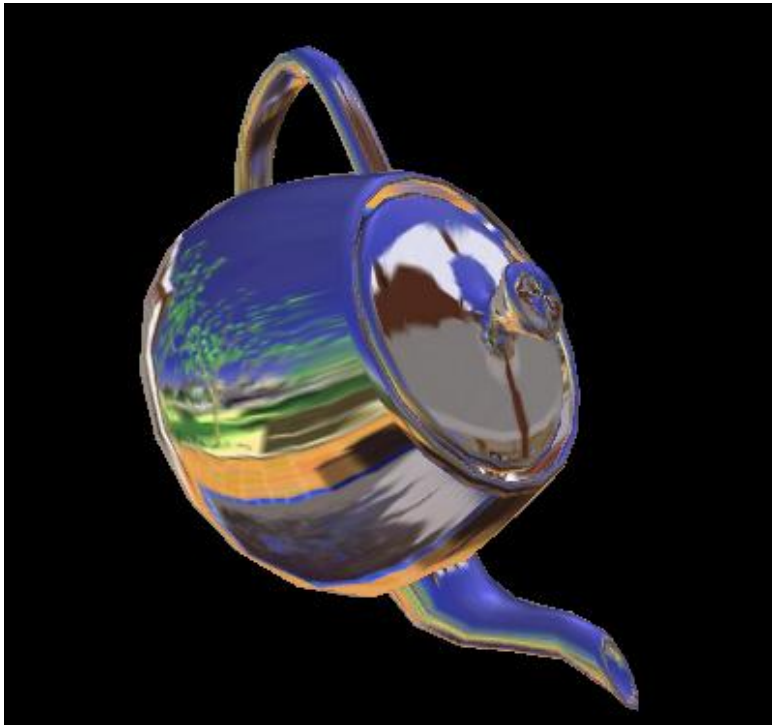
Example Map
(Steve van der Burg)

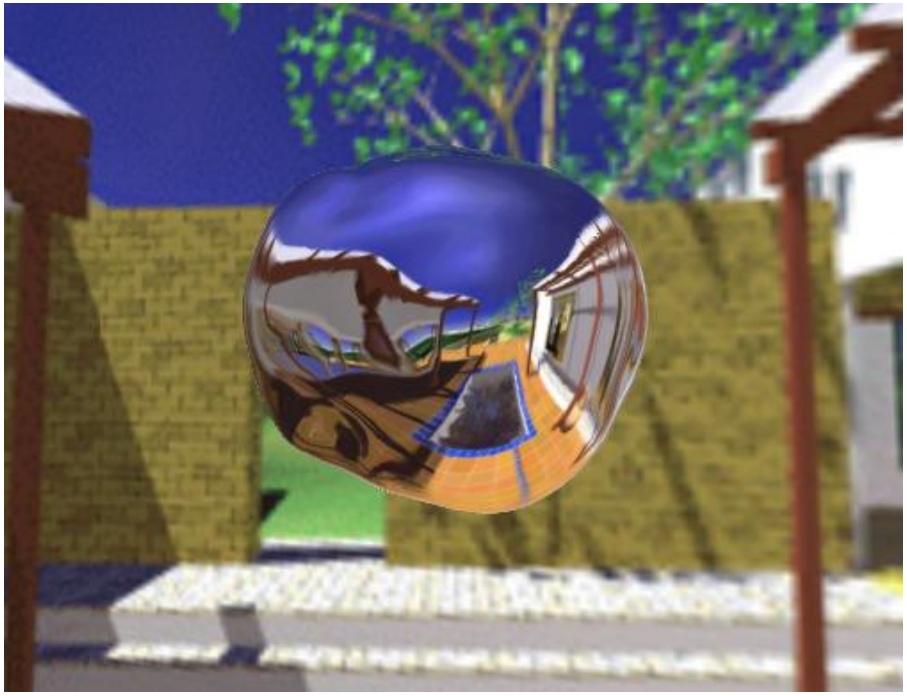


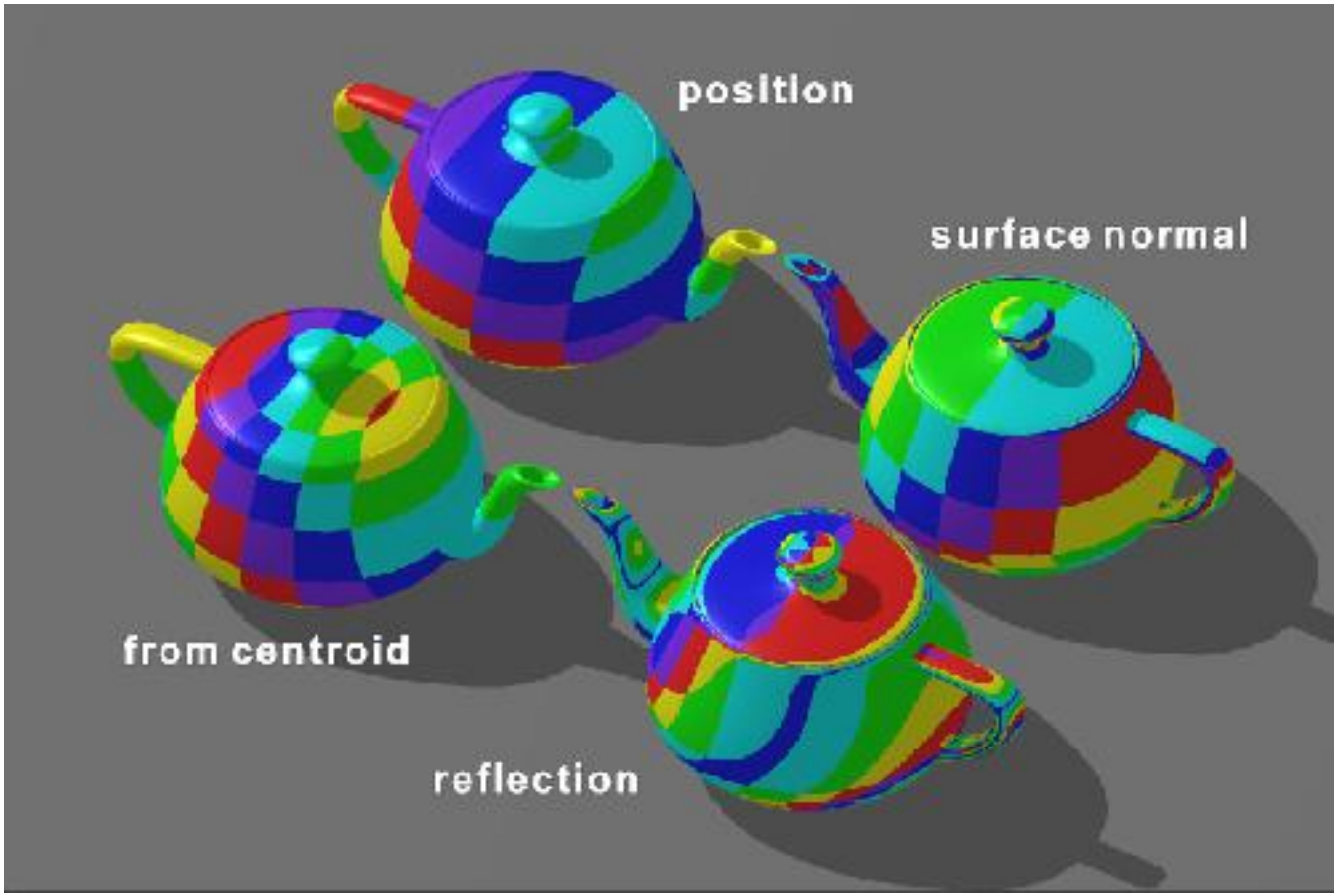
Texture Map Entity



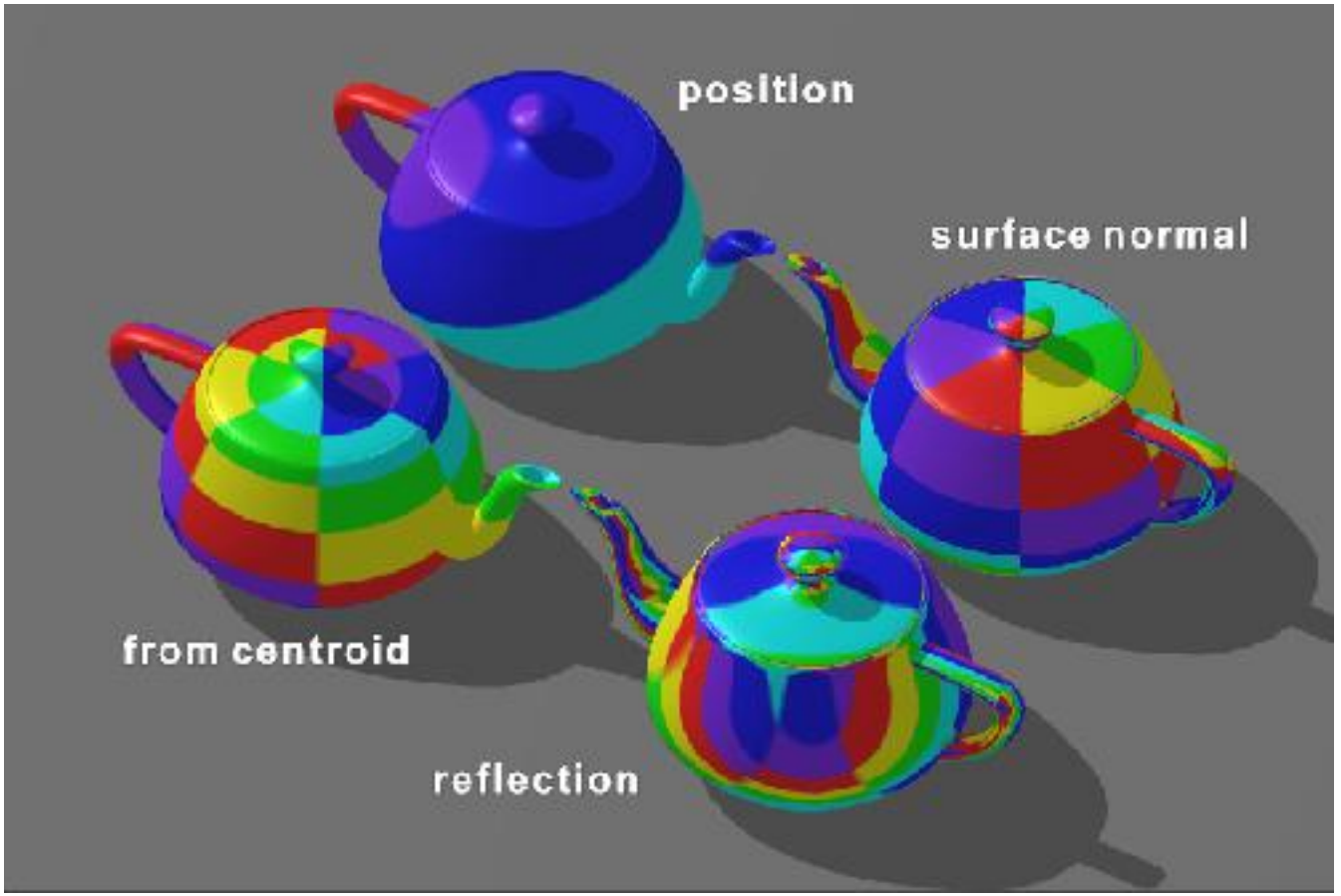
Using Reflection Entity



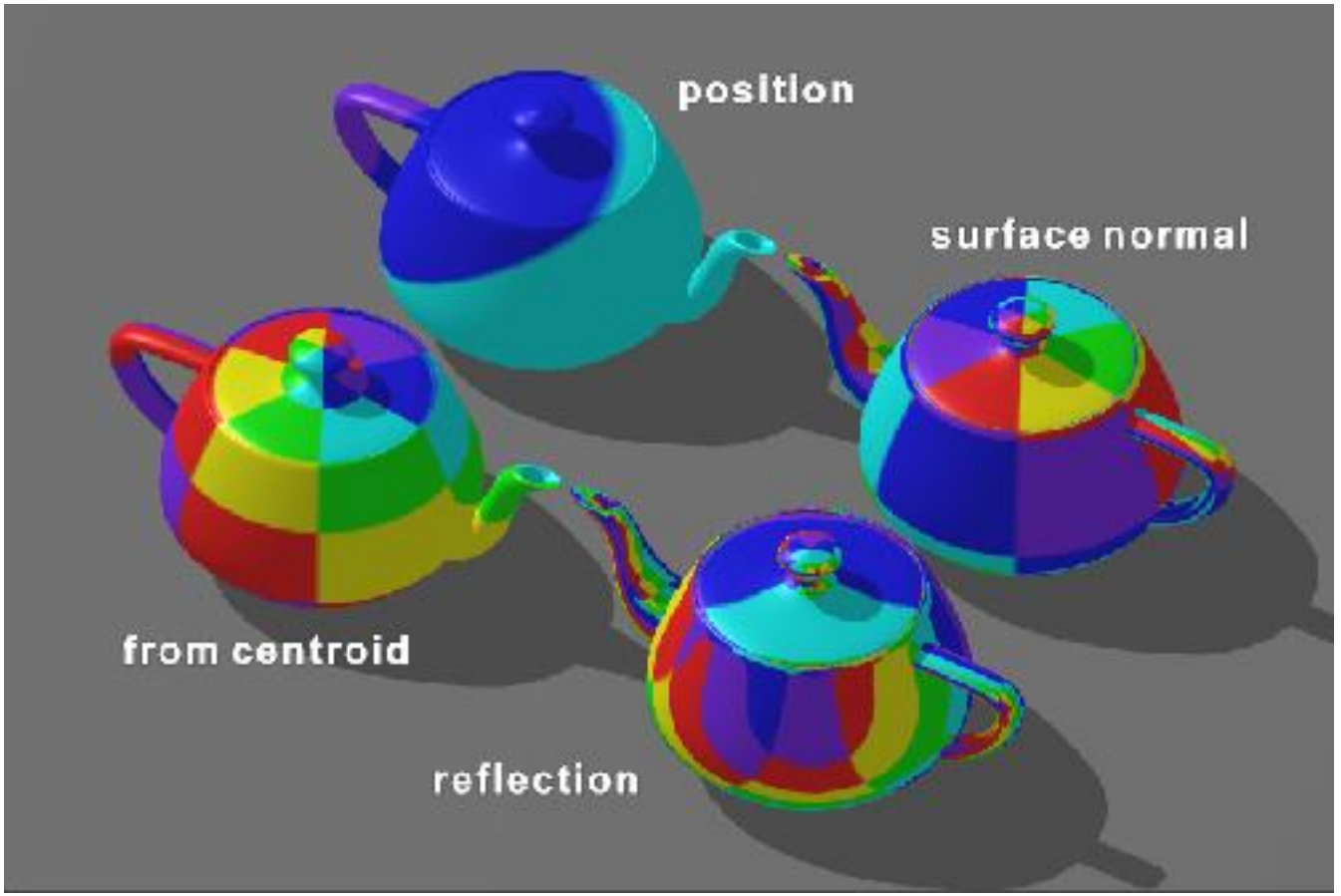




Planar Map with different projection entities



Cylindrical Map with different projection entities



Spherical Map with different projection entities

Textures in OpenGL

- **Texture basics**
  - glGenTextures(n, &id)
  - glTexImage[123]D(target, level, iFormat, w[, h[, d]], border, format, type, data)
  - glBindTexture(GL_TEXTURE_[123]D, id)
  - glEnable(GL_TEXTURE_[123]D)
  - glDisable(GL_TEXTURE_[123]D)

Textures in OpenGL *(cont'd)*

- **Specifying a texture image**
  - glTexImage[123]D(target, level, iFormat, w[, h[, d]], b, format, type, data)
    - target: specify texture target type (GL_TEXTURE[1/2/3]D)
    - level: number of level of detail (mipmap)
    - iFormat: ~40 types specifying number of channels and bits/channel:
    - w, h, d: width, height, depth of the texture map
    - b: border 0 or 1
    - format: ~11 types color channels in data
    - type: ~20 bit arrangement of data
    - data: pointer to the original texture data

Textures in OpenGL *(cont'd)*

- **Run-time parameters**
  - glTexParameter[fi](GL_TEXTURE_[123]D, pname, parms)
  - glTexEnv[fi](target, pname, param)
  - glTexEnv[fi]v(target, pname, param)
- **Auto generation of tex coordinates**
  - glTexGen[ifd][v](coord, pname, parms)

Texture Access Functions in GLSL

```
vec4 texture1D (sampler1D sampler, float coord[, float bias])
vec4 texture2D (sampler2D sampler, vec2 coord[, float bias])
vec4 texture3D (sampler3D sampler, vec3 coord[, float bias])
```

- Access texel color at the given texture coordinate.
- OpenGL texture properties (filtering method, mipmap, etc) are taken into account when using these functions. However, level of detail is not taken into account when accessing texture in the vertex shader.
- Returns texel color in <r, g, b, a> format.
- The 'bias' is optional and can only be used fragment shader. It will be ignored if texture has no mipmap.

Texturing

I. Affine texture mapping and filtering

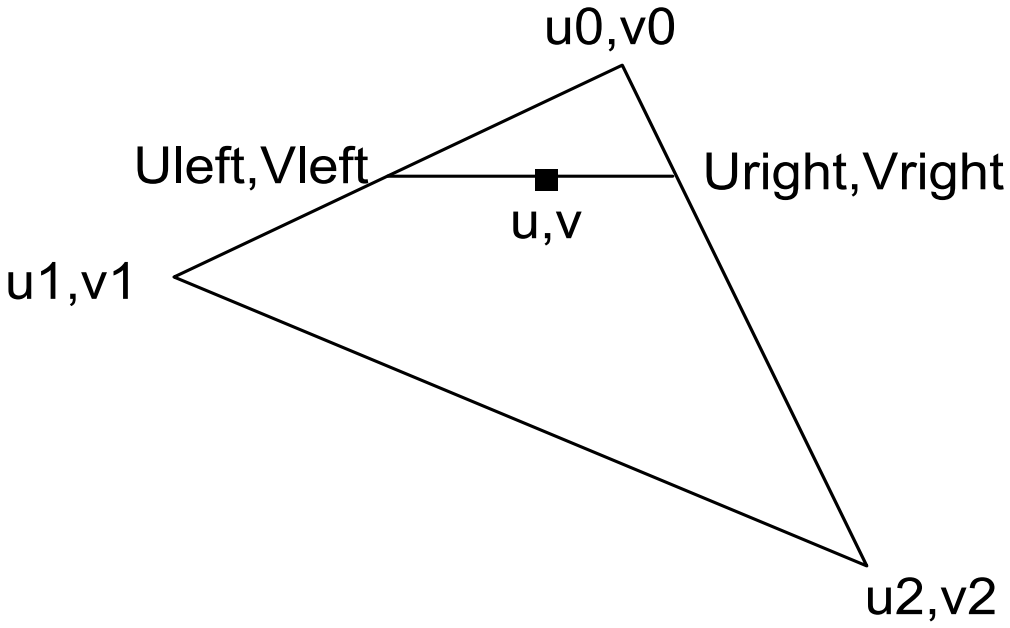
II. Perspective correct texture mapping

I. Affine Texture Mapping and Filtering

- A texture map is a 2D bitmapped image which is wrapped around a 3D object
- The texture map is made from texture elements called Texel
- The concept of texture mapping is to map each pixel in the rendered 3D object to a corresponding texel in the 2D texture

I. Affine Texture Mapping and Filtering *(cont'd)*

- When a texture is wrapped around a 3D object, each vertex would be assigned the respective texture coordinate



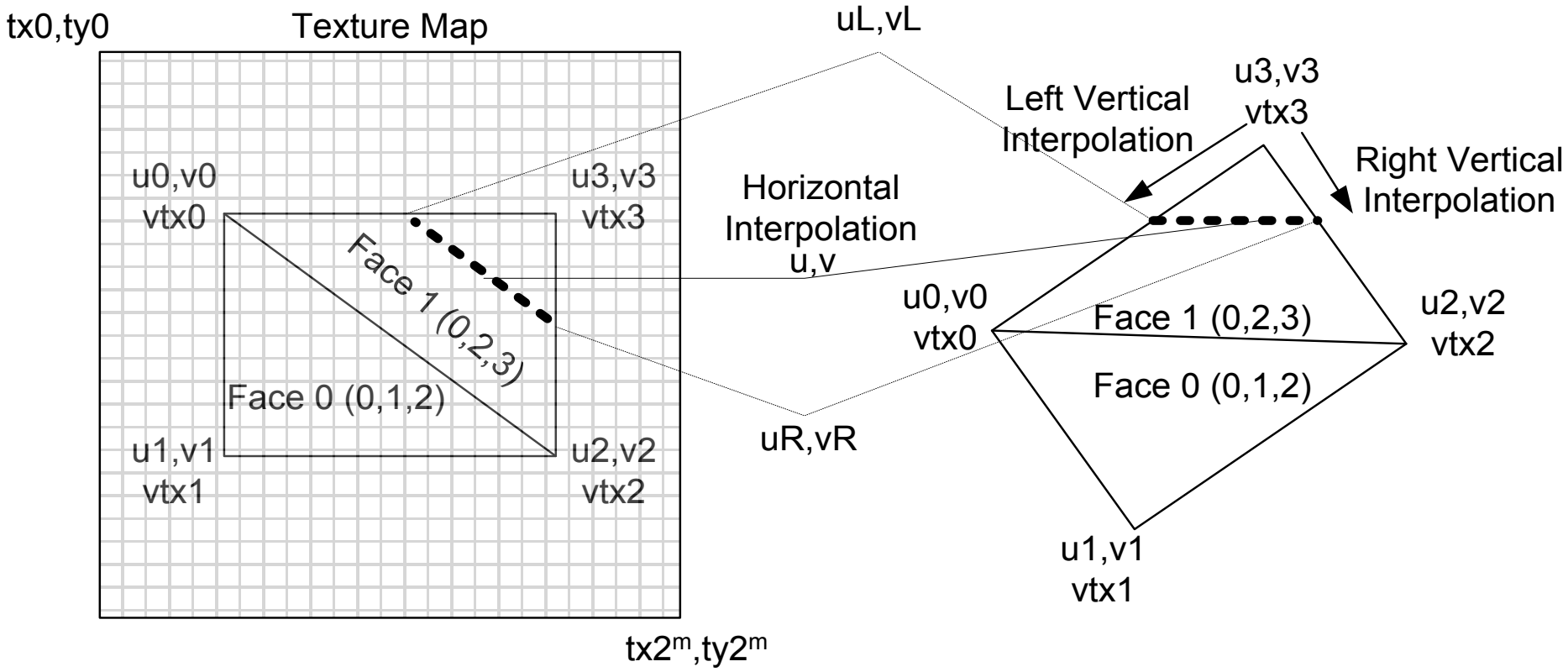
Affine Texture Mapping

- The affine texture mapping linearly interpolates the u, v coordinates along the left and right edges during edge scanning

- Once the left and right texture coordinates are known along a horizontal span, the texture mapping to a pixel is done by linearly interpolating horizontally the left and right texture coordinates

Affine Texture Mapping *(cont'd)*

- In other words we have to interpolate the texture coordinates vertically then horizontally



Affine Texture Mapping *(cont'd)*

- Similar to the interpolation of the vertex left and right position, normal, light intensity etc… the left and right texture coordinate (u,v) are incrementally computed

- The vertical increment is:

$$\frac{Texture\,\Delta u}{Face\,\Delta y} and \frac{Texture\,\Delta v}{Face\,\Delta y}$$

Affine Texture Mapping *(cont'd)*

- The horizontal increment is

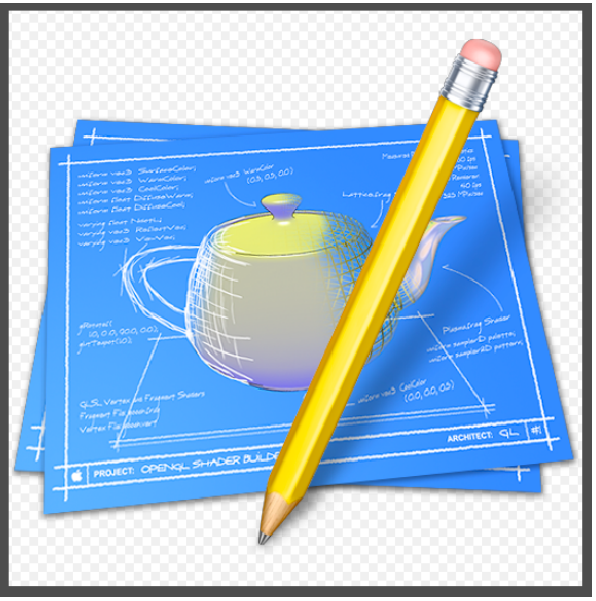
$$\frac{Texture:uRight - uLeft}{Face:xRight - xLeft} \; or \; \frac{Texture\,\Delta u}{Face\,\Delta x}$$

$$\frac{Texture:vRight - vLeft}{Face:xRight - xLeft} \; or \; \frac{Texture\,\Delta v}{Face\,\Delta x}$$

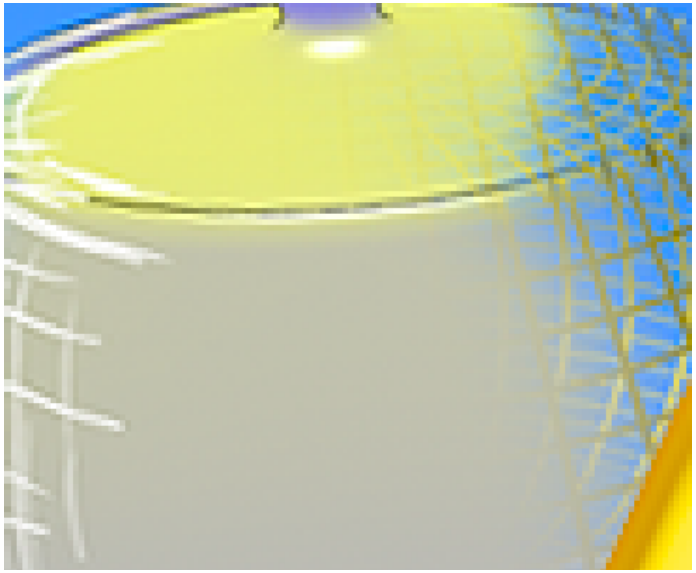
Minification and Magnification

- During the mapping process, when the size of the texture map and the face is different, several texels may cover the same pixel or several pixels may use the same texel
- If there are more texels than pixels, then the texture is minified.
- If there are less texels then pixels, then the texture is magnified.

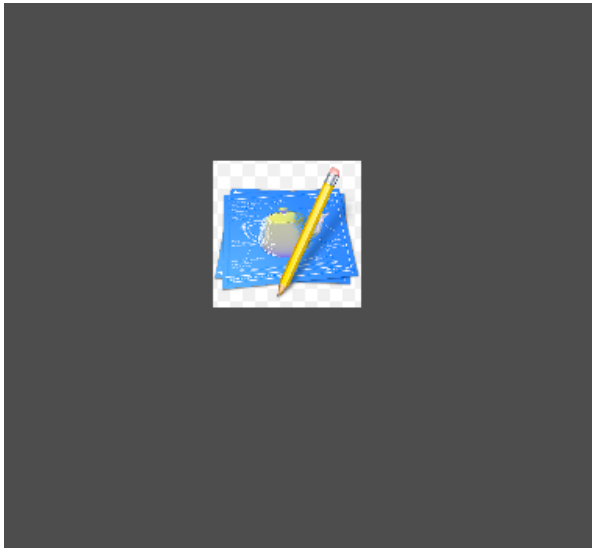
Magnification and Minification



512x512
Polygon area



1024x1024
Polygon area



256x256
Polygon area

Minification and Magnification *(cont'd)*

- In the minification case the texture u increment is $\Delta u/\Delta x = 512/256 = 2$
- In the magnification case the texture u increment is $\Delta u/\Delta x = 512/1024 = 0.5$
- The texture index floating point value is converted to an integer and the pixel is point sampled.
- Consequently, the fractional portion of the pixel was not taken into consideration.
- We call this sampling method **point sampling**.

Bilinear filtering

- The Bilinear filtering samples more than one texel by taking into consideration the floating point value and by blending horizontally and vertically four texels

- The filtering is Bilinear because the filtering is applied on two dimensions, the horizontal and the vertical

Bilinear Filtering *(cont'd)*

- This technique is used to smooth the textures during the magnification or minification of the texture
- For example, if the floating point value is 6.3f. 70% of texel index 6 is blended with 30% of texel index 7
- Pixel color = texel[6]*0.7f + texel[7]*0.3f

Bilinear Filtering *(cont'd)*

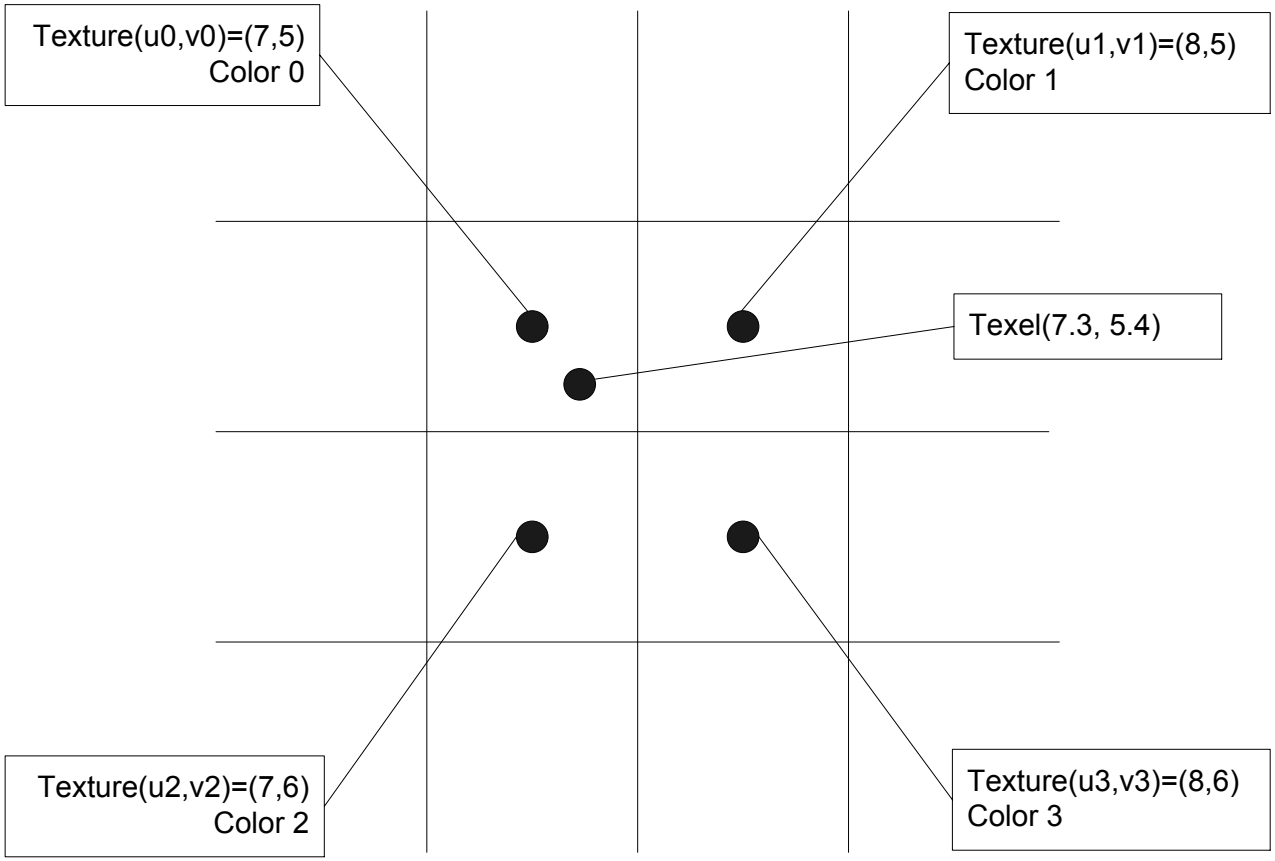
- Since floating point (u,v) texel value will most likely not point to an exact texel, the (u,v) location would be located between four texels

- Consequently, we will blend the upper row with the lower row

Bilinear Filtering *(cont'd)*

- The upper row color would be the blending of the up-left texel with the up-right texel
- Similarly, the lower row would be the blending of the down-left texel with the down-right texel

Bilinear Filtering *(cont'd)*



Bilinear Filtering *(cont'd)*

- Horizontal Interpolation

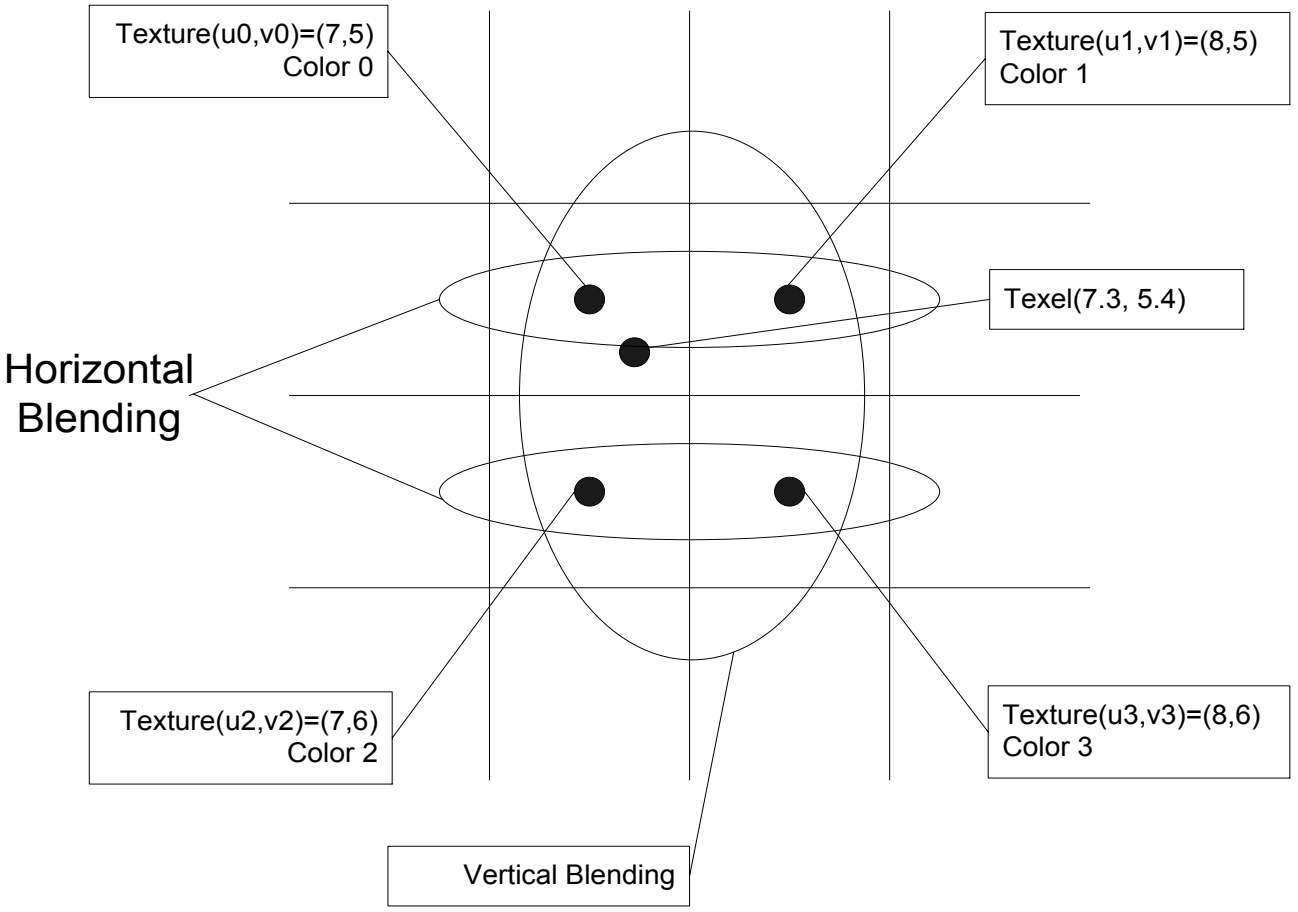
$$Ch0 = C0 + \frac{(C1 - C0)}{(u1 - u0)}(u - u0)$$

$$Ch1 = C2 + \frac{(C3 - C2)}{(u3 - u2)}(u - u2)$$

- Vertical Interpolation

$$C = Ch1 + \frac{(Ch1 - Ch0)}{(v2 - v0)}(v - v0)$$

Bilinear Filtering *(cont'd)*



Out of Boundary Texels

- Since we are incrementing the u, v coordinates, we might have a texel coordinate that lies outside the texture map, i.e. the u or v is less than 0 or greater than or equal to 1

- Some options:
  - Repeat the last texel value. (u, v) = clamp((u, v), 0, 1)
  - Wrap the texture coordinate. (u, v) = fract(u, v)
  - Use constant (same) color for all outside texel.

What to do with texels color?

- If there is no lighting:
  - The texel color is used as the surface color
  - The texel color is multiplied with the surface color
- If there is lighting:
  - The texel color could be used as one (or more) of the material coefficient(s). It is commonly used as the diffuse and specular component.

Bilinear Interpolation Limitation

- Bilinear filtering works well when, the pixels to texels ratio is no more than 1:2 (minification case).
- In other words, if the pixels to texels ratio is more than 1:2, then the blending function would be missing some texels.
- The solution to this problem is to use Mip mapping which allow to get a pixels to texels ratio between 1:1 and 1:2.

Mipmapping

- When mapping a texel from a texture to a polygon using point sampling or bilinear filtering the result might look incorrect when the pixels to texels ratio is more than 1:2.
- The mipmapping strategy solves the problem by creating a list of textures where each texture is half as large as the previous one.
- Mip stands for “multum in parvo” in latin which in English means “many in small place”.

- The list of textures ends when we reach a texture of size 1x1

- Benefit of mipmapping:
  - reduce rendering artifacts
  - improve rendering performance

Generating a Mipmap

- The mipmaps could be generated in many ways.
- The common way is to average the higher resolution texture map to generate the next lower resolution one.
- Once the list of mipmaps is generated we need to select the right mipmap to use during rendering process.

Generating a Mipmap *(cont'd)*

- The original shape and size of the original texture map should be a rectangle with power of 2 width and height.

- The total number of mipmap levels including the original texture map is:

$$\log_2 \max(\text{width}, \text{height}) + 1$$

Generating a Mipmap *(cont'd)*

- For example, if the original texture size is 512x512:
  - Level 0, the texture size is 512x512.
  - Level 1, it is 256x256.
  - .
  - .
  - Level 9, it is 1x1.
  - The total number of mipmap textures are:

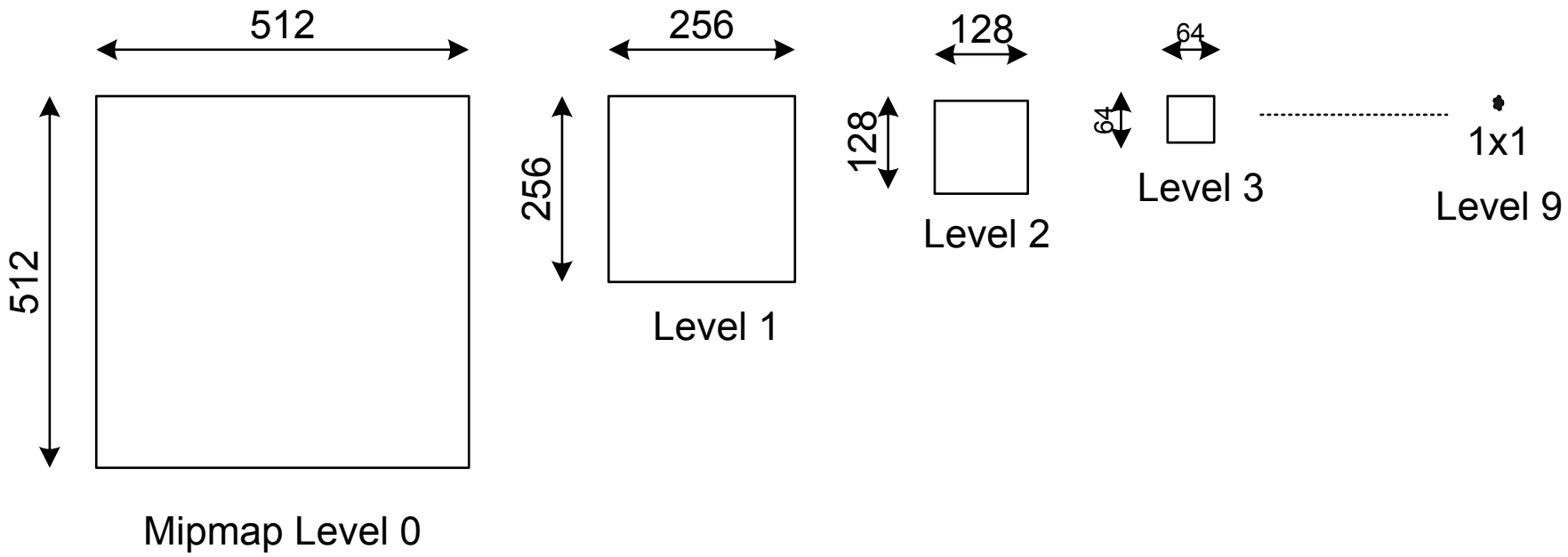
    $\text{Log}_2\ 512 + 1 = 9 + 1 = 10$

Generating a Mipmap *(cont'd)*

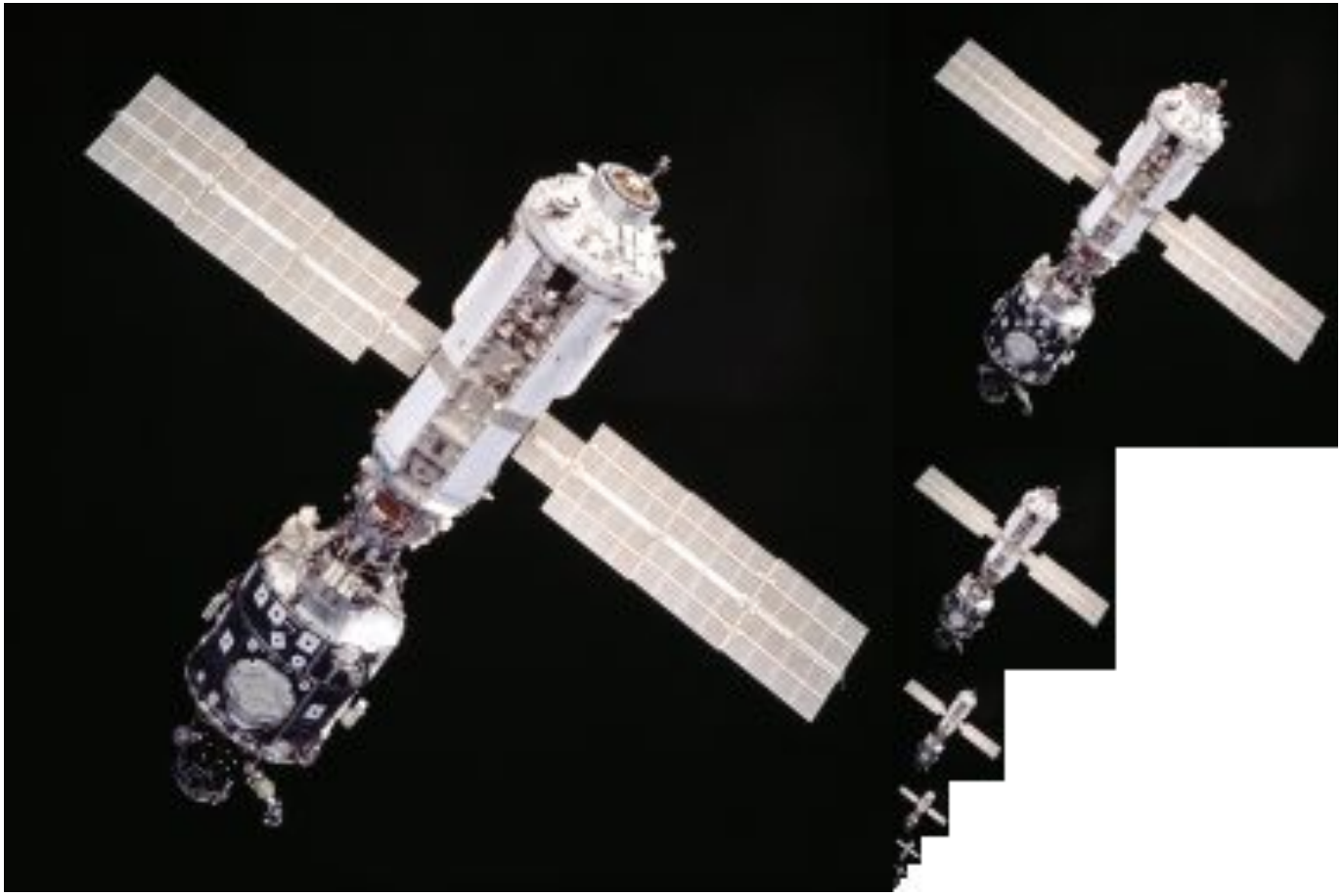
- If the original size is 128x32:
  - Level 0, the size is 128x32.
  - Level 1, 64x16.
  - Level 2, 32x8.
  - Level 3, 16x2.
  - Level 4, 8x1.
  - Level 5, 4x1.
  - Level 6, 2x1.
  - Level 7, 1x1.
  - Total number of mipmap textures are:

    $\text{Log}_2 \max(128, 32) + 1 = 7 + 1 = 8$

Generating a Mipmap *(cont'd)*



Mipmap Example



Generating a Mipmap *(cont'd)*

- Because the texture size is reduced by ½ each time, the storage requirement for the next level is ¼ of the previous one.

- The total increase in storage space required to store all mipmap textures is 1/3. Why?

Generating a Mipmap *(cont'd)*

- For example, if the original texture map size is 512x512:
  - The size of the next level is 256x256. The storage ratio is:

    (256x256) / (512x512) = 65536 / 262144 = ¼.
  - The total required storage is:

    $512^2 + 256^2 + 128^2 + 64^2 + 32^2 + 16^2 +$

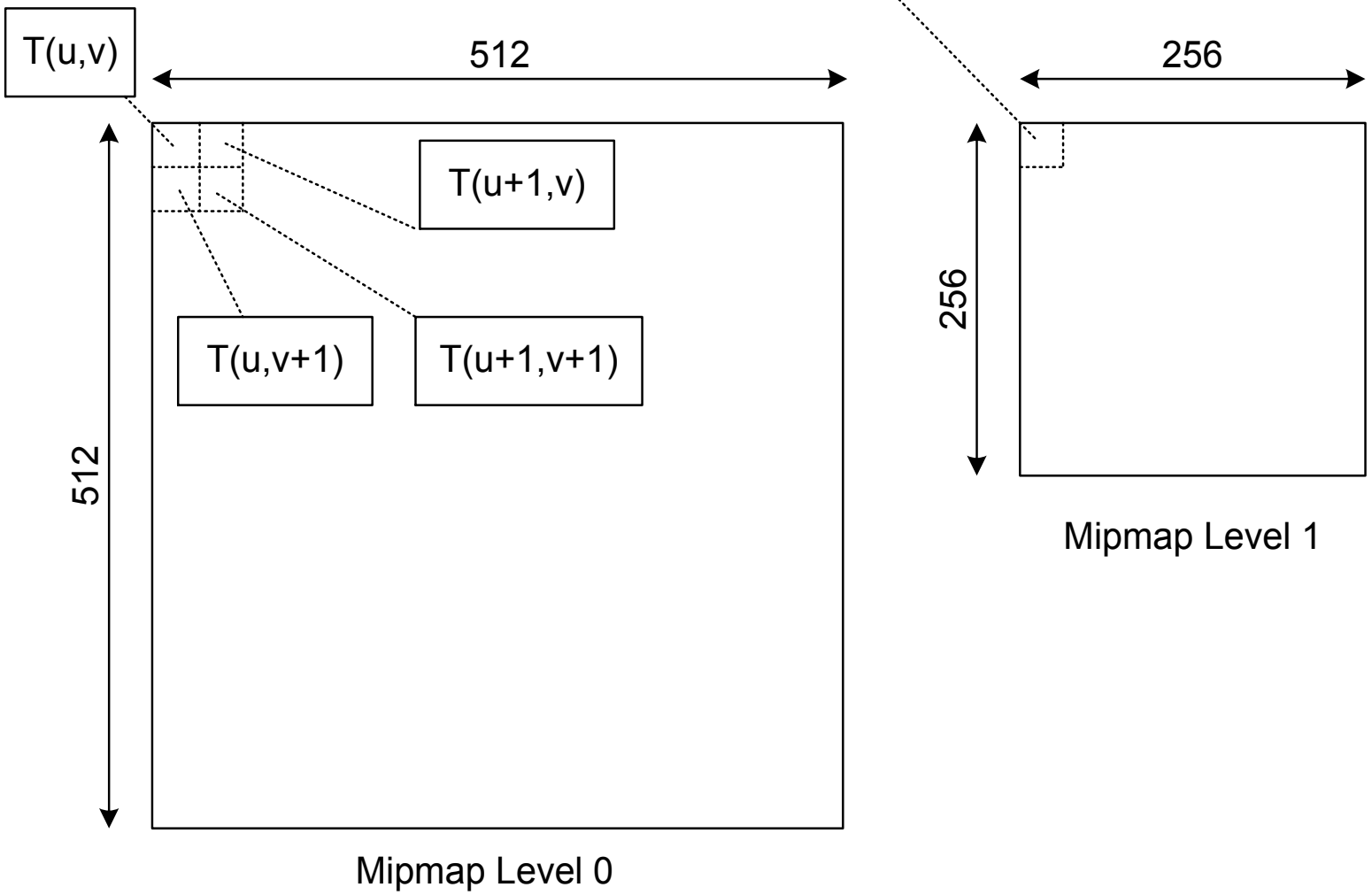
    $8^2 + 4^2 + 2^2 + 1^2 = 349525$
  - The ratio storage ratio is: 349525 / 262144 = 1.33

Generating a Mipmap *(cont'd)*

- To generate a mipmap of a particular level, average the four pixels using the averaging filter of the previous mipmaps level.
- In this case, a texel(u,v) at mipmap level n would be:

$$Texel_n(u,v) = \frac{Texel_{n-1}(2u,2v) + Texel_{n-1}(2u+1,2v) + Texel_{n-1}(2u+1,2v+1) + Texel_{n-1}(2u,2v+1)}{4}$$

$$Texel_n(u,v) = \frac{Texel_{n-1}(u,v) + Texel_{n-1}(u+1,v) + Texel_{n-1}(u+1,v+1) + Texel_{n-1}(u,v+1)}{4}$$



Mipmap Level Selection

- Texel to Pixel ratio mipmap selection

Texel to Pixel Ratio Mipmap Selection

The mipmap ratio is:

$$\frac{\text{area of the original texture map in texel}}{\text{area of the projected polygon in pixel}}$$

Texel to Pixel Ratio Mipmap Selection *(cont'd)*

Once the mipmap ratio is known, the correct mipmap level is calculated as -

$$\log_4(\text{mipmap ratio})$$

because for every factor of 4 we need to move to the next mipmap level

Texel to Pixel Ratio Mipmap Selection *(cont'd)*

- For example, suppose the original texture map size is 512x512 and the projected polygon area is 32x70.
  The mipmap ratio would be 262144/2240 = 117.03
  Log_4 117.03 = ln 117.03 / ln 4 = 4.76 / 1.39 = 3.43
  So a 32x70 pixel face will use 57% of mipmap3(64x64) and 43% of mipmap4(32x32)

Texel to Pixel Ratio Mipmap Selection *(cont'd)*

- For example, suppose the original texture map size is 512x512 and the projected polygon area is 2x2. The mipmap ratio would be 262144/4 = 65536.00

  Log_4 65536 = ln 65536 / ln 4 = 11.09 / 1.39 = 8

  So a 2x2 pixel face will use 100% of mipmap8(2x2) and 0% of mipmap9(1x1)

Trilinear Filtering

- There are 2 common methods to calculate the final texture color when doing mipmaping.
  - Do point sampling on each mipmap and then linearly interpolate the color values based on the mipmap ratio.
  - Do bilinear sampling on each mipmap and then linearly interpolate the color values based on the mipmap ratio. This method is called trilinear filtering.

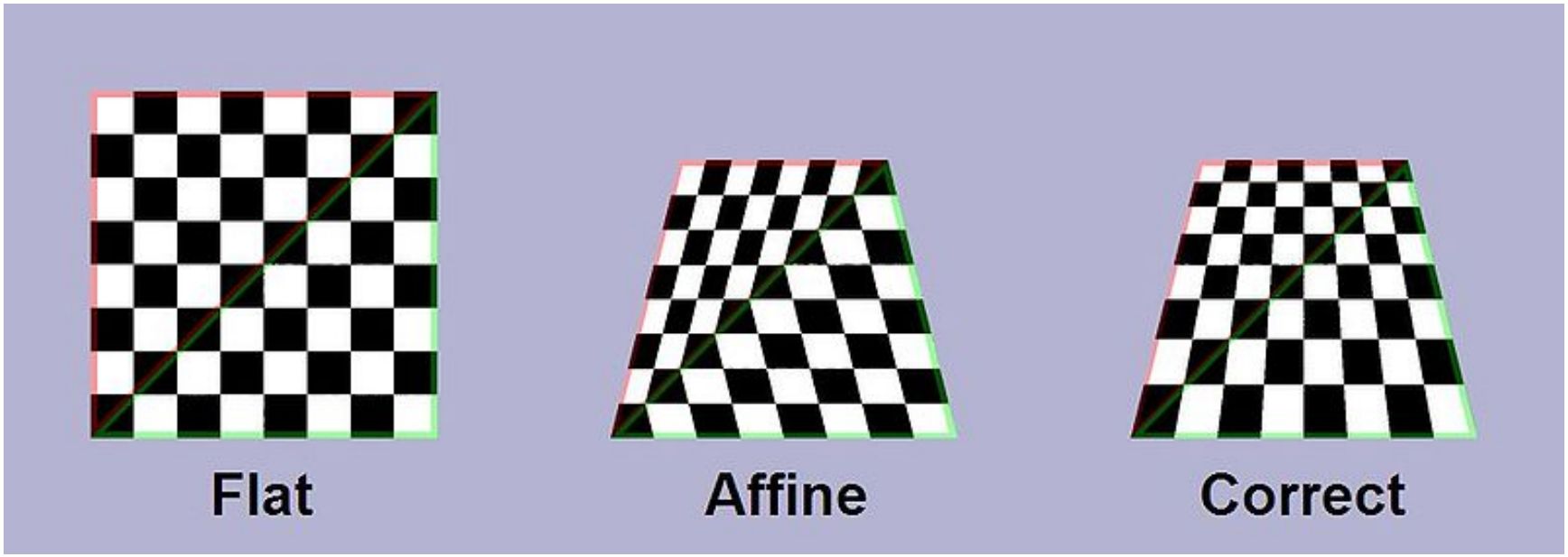
Trilinear Filtering *(cont'd)*

A 32x70 pixel face will use 57% of the bilinear interpolation of Mipmap 3 (64x64) and 43% of the bilinear interpolation of Mipmap 4 (32x32)

Perspective Correct Texture Mapping

Gouraud Shading

Comparison of Texture Mapping



Perspective Correct Texture Mapping

a) Definition of a linear relation

b) Linear interpolation and linear relations

c) 1/z buffering

d) Perspective correct texture mapping

a) Definition of a Linear Relation

- When we say for example that x is a linear function of z, we mean that x could be written in terms of z as follows:

$$x = az + b$$

where a and b are two real numbers (constants).

b) Linear Interpolation and Linear Relations

- Let A(xA, yA, zA) and B(xB, yB, zB) be two points in the camera space.
- Let A'(xA', yA') and B'(xB', yB') be the projections of A and B in the screen space.
- zA and zB are the respective depths of A and B in camera space.

b) Linear Interpolation and Linear Relations *(cont'd)*

- One way to find the depth of the points between A and B we perform a linear interpolation in terms of x' and y' of each point between A and B.
- This most likely leads to incorrect results because when we move from the camera space to the screen space we multiply x by the focal distance and divide it by z.
- Basically, the projection transformation is NOT linear.
- The same thing applies to y.

b) Linear Interpolation and Linear Relations *(cont'd)*

- In other words,

$$xA' = f*xA/zA$$
$$yA' = f*yA/zA$$
$$xB' = f*xB/zB$$
$$yB' = f*yB/zB$$

- For a point M(x', y') between A' and B', if M is the projection of a point C(x,y,z) that belongs to the edge AB, then:

$$x' = f*x/z$$
$$y' = f*y/z$$

b) Linear Interpolation and Linear Relations *(cont'd)*

- So, as you can see, the relation between x' and z is not linear.
- In this case, a linear interpolation of z in terms of x' will not lead to a correct result.
- Though when dealing with Gouraud shading and lighting and even in depth calculation, the results are **GOOD**, still they are incorrect.

c) 1/z Buffering

- The rasterizer provides us with the screen coordinates of a point or a vertex (x', y') which are related to the depth of the vertex by the following relations:

$$x' = f * x * (1 / z) \qquad \text{(I)}$$

$$y' = f * y * (1 / z) \qquad \text{(II)}$$

- Where:
  - 'f' is the focal distance
  - (x, y, z) are the vertex coordinate in camera space

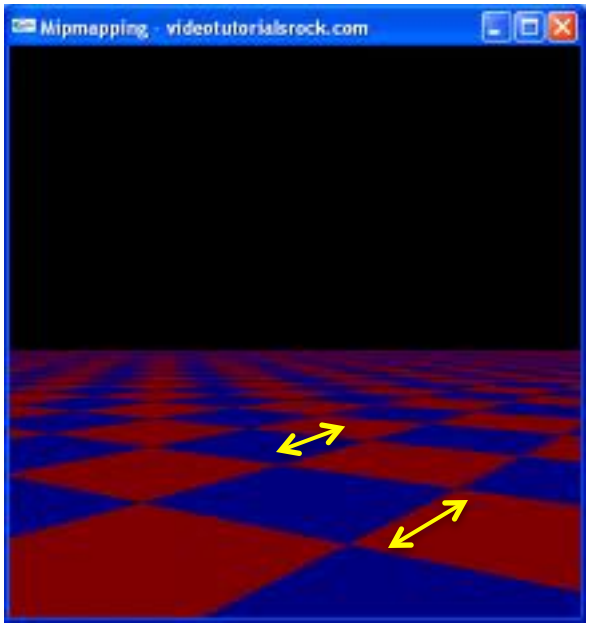
c) 1/z Buffering *(cont'd)*

- On the other hand, let the point (x,y,z) belong to the segment [AB] where the coordinates of A and B in the camera space are respectively A(xA, yA, zA) and B(xB, yB, zB).

- A relation between x, y and z can be found very easily

c) 1/z Buffering *(cont'd)*

- Forward and backward mapping
  - Forward mapping : Implementation of the graphics pipeline
    - Given (x,y,z), find appropriate (x', y') on the screen
    - Is not optimal for texture mapping



c) 1/z Buffering *(cont'd)*

- We need an "invariant" quantity that does not change with depth!
- Try to reverse the problem
  - Given (x', y') in screen space, what point (x, y, z) on the surface of the object does it represent?
  - Behavior found in modern hardware – fragment shader!
  - Since the rasterizer operates at pixel level accuracy, there will be no "missing texels"
  - Exactly 1 pixel width between adjacent points on the surface

c) 1/z Buffering *(cont'd)*

- Lets try "Backward Mapping"

$$x' = \frac{x}{z} \Rightarrow$$

$$x = x'z$$

Pixel location (given)

Depth from camera (given for vertices)

c) 1/z Buffering *(cont'd)*

- To find: A relationship between x' and z

- In other words
  - If we step by one pixel in X direction (on screen), how does the depth of the points on the surface change?
  - It would help us (immensely, no doubt) if this relationship would also be a linear function
    - Projection transformation IS NOT

c) 1/z Buffering *(cont'd)*

- Consider

  $x = az + b$

  The line joining points A(xA, yA,zA) and B(xB, yB, zB) on the surface.

  What space does this line lie in?

Substituting the value of x, we get

$$x'z = az + b$$

$$b = z(x' - a)$$

$$z = \frac{b}{(x' - a)}$$

NOT Linear!!

Consider the reciprocal of this equation

$$\frac{1}{z} = \frac{1}{b}x' - \frac{a}{b}$$

LINEAR relationship

c) 1/z Buffering *(cont'd)*

- x/z is linear in screen space
  - By definition of projection transformation
- What about the texture coordinate u?
  - Yes!
  - Texture coordinates vary linearly along the surface as a function of x
- By transitivity
  - u/z is also linear in screen space!

c) 1/z Buffering *(cont'd)*

- We can interpolate u/z directly without calculating the x values

- Affine mapping interpolates the (u, v) values in screen space which leads to errors
- Perspective correct mapping interpolates (u/z, v/z) in screen space for correct results

Implementation

1. Project **object vertices** & texture coordinates into screen space,
   $xA' = xA/zA$, $uA' = uA/zA$
   $xB' = xB/zB$, $uB' = uB/zB$
2. Let $zA' = 1/zA$ and $zB' = 1/zB$
3. Interpolate u' and z' along the X direction, stepping by one pixel in the loop
   1. At each new value of x', calculate the texture space coordinate $u = u'/z'$
   2. Similarly for y and v ($v = v'/z'$)

c) 1/z Buffering *(cont'd)*

- In contrast of z-buffering where we compare the values of z and take the smallest to put in the z-buffer, in 1/z buffering we compare the 1/z values of the points and keep the greatest value of 1/z in the array because it will correspond to the smallest value of z

d) Perspective Correct Texture Mapping *(cont'd)*

- Knowing the texture coordinates of A and B, we can interpolate linearly to find the texture coordinates of each point on the segment.

- In fact, the relation in the camera space between u and z is linear because we have not done any perspective projection yet.

d) Perspective Correct Texture Mapping *(cont'd)*

- In other words, we can interpolate texture coordinates u and v along horizontal scan lines in the **camera space** once we know the texture coordinates of the extremities A and B.

- Similarly and for the same reasons, the relation between v and z is also linear

Do I have to do this?

- NO
- Current hardware implements this functionality

- `varying / out` variables in GLSL are perspective-correct interpolated

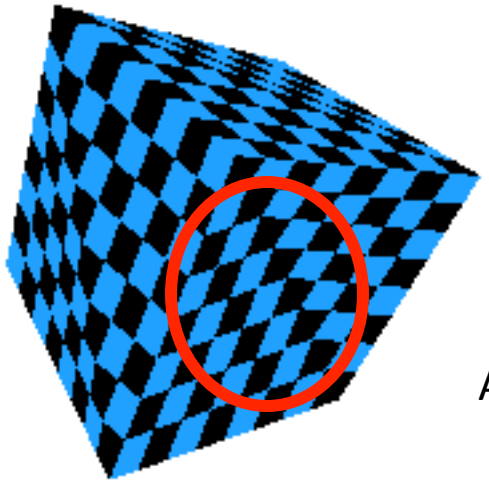
- OpenGL 3.0+ -
  - If you prefer affine interpolation, use qualifier `flat`

- You should know and understand how texture coordinate interpolation works!

Optimization

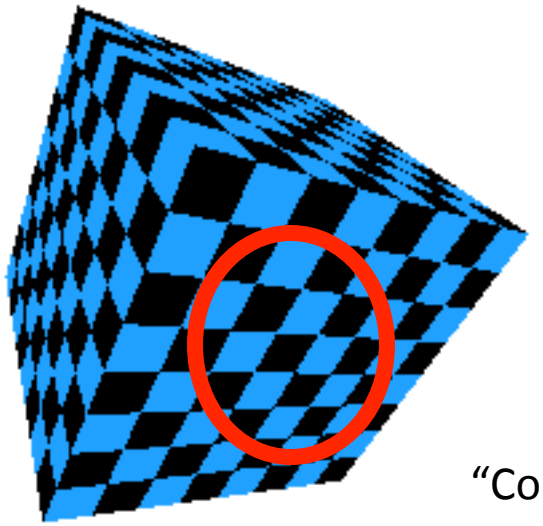
- Division is expensive
- Perform Perspective-correct division after a fixed interval (8/16/24 pixels)
- Linearly interpolate the values in between the "correct" values

- In most cases, there is no visible difference

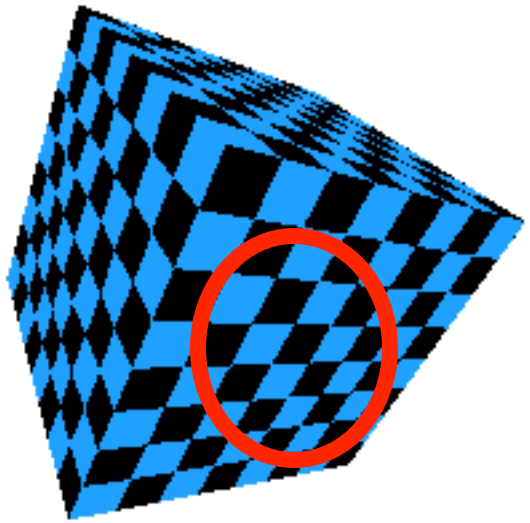


Affine

“Correct” every 16 pixels



“Correct”



References

- http://chrishecker.com/Miscellaneous_Technical_Articles

- Chris Hecker's articles for the Game Developer Magazine (PDF format)

DigiPen INSTITUTE OF TECHNOLOGY

Appendix

Deriving Linearity between u/z and x/z

c) 1/z Buffering *(cont'd)*

- To simplify the problem we will assume that [AB] belongs to the xz-plane, that is the y-coordinate of any point of the segment is equal to zero and obviously, yA = yB = 0.
- The equation of the segment [AB] is given by:

$$x = az + b \qquad \text{(III)}$$

where a and b are constants that we can determine using the fact that A and B are known and belong to the segment.

c) 1/z Buffering *(cont'd)*

- If A and B belong to the segment then their coordinates verify equation (III):

$$zA = a.xA + b \quad (1)$$

$$zB = a.xB + b \quad (2)$$

- What we just got is a system of two equations in two unknowns a and b. Solving this system to find a and b can be done by applying **Kramer's Rule**

c) 1/z Buffering *(cont'd)*

- Kramer's Rule:

$$a = \frac{\begin{vmatrix} zA & b \\ zB & b \end{vmatrix}}{\begin{vmatrix} xA & b \\ xB & b \end{vmatrix}} = \frac{b(zB - zA)}{b(xB - xA)}$$

$$a = \frac{zB - zA}{xB - xA}$$

c) 1/z Buffering *(cont'd)*

- At this point, we have relation (I) between x_{screen}, z, and x and relation (III) between z and x. we can use them to get a relation only between x_{screen} and z, i.e. getting z from x_{screen}:

(III) $\Rightarrow x = (z - b)/a \Rightarrow 1/x = a/(z - b)$

$$(I) \Rightarrow x\_screen = (fx)\left(\frac{1}{z}\right) \Rightarrow \frac{x\_screen}{fx} = \frac{1}{z} \Rightarrow \frac{1}{z} = \frac{x\_screen}{f}\left(\frac{1}{x}\right)$$

c) 1/z Buffering *(cont'd)*

- Replacing 1/x by its value in terms of z implies

$$\frac{1}{z} = \frac{x\_screen}{f}\left(\frac{a}{z-b}\right) \Rightarrow \frac{z-b}{z} = \frac{a}{f}x\_screen \Rightarrow 1-\frac{b}{z} = \frac{a}{f}x\_screen$$

$$\Rightarrow \frac{b}{z} = 1-\frac{a}{f}x\_screen \Rightarrow \frac{1}{z} = \left(-\frac{a}{bf}\right)x\_screen + \left(\frac{1}{b}\right) \quad (IV)$$

c) 1/z Buffering *(cont'd)*

- Now, knowing A'(xA',yA') and B'(xB',yB') to be the perspective projections of A(xA,yA,zA) and B(xB,yB,zB) on the screen, we can interpolate the 1/z value (and then calculate the depth z when needed) of every intermediate point between A and B using relation IV

c) 1/z Buffering *(cont'd)*

- Instead of calculating each value using relation IV, we can use forward differencing to find the value of 1/z.

- Let us see by which value 1/z increases if x_screen increases by 1

$$\left(\frac{1}{z}\right)' = \left(-\frac{a}{bf}\right)(x\_screen + 1) + \left(\frac{1}{b}\right) = \left(-\frac{a}{bf}\right)(x\_screen) + \left(\frac{1}{b}\right) + \left(-\frac{a}{bf}\right) = \frac{1}{z} + \left(-\frac{a}{bf}\right)$$

c) 1/z Buffering *(cont'd)*

Therefore, for each increment of x_screen by 1 corresponds an increment by $\left(-\frac{a}{bf}\right)$ in 1/z

d) Perspective Correct Texture Mapping

- Given the coordinates of the vertices A and B in camera space, let us add to their corresponding coordinates the texture coordinates u and v.

- We will get A(xA,yA,zA,uA,vA) and B(xB,yB,zB,uB,vB).

d) Perspective Correct Texture Mapping *(cont'd)*

- Mathematically, what we tried to say is that u and v are linear functions of z:

$$u = cz + m \quad (A)$$

$$v = dz + n \quad (B)$$

- To find c, d, m, and n. we use the texture coordinates and the depth value (z) of A and B

d) Perspective Correct Texture Mapping *(cont'd)*

- The system below allows us to find c and m:

$$uA = czA + m$$

$$uB = czB + m$$

- The system below allows us to find d and n:

$$vA = dzA + n$$

$$vB = dzB + n$$

d) Perspective Correct Texture Mapping *(cont'd)*

- Now, we know the following formula from the previous section:

$$\frac{1}{z} = \left(-\frac{a}{bf}\right) x\_screen + \left(\frac{1}{b}\right)$$

- which can be rewritten as follows:

$$\frac{1}{z} = \frac{-\frac{a(x\_screen)}{f} + 1}{b} = \frac{-a(x\_screen) + f}{bf} \Rightarrow z = \frac{bf}{-a(x\_screen) + f}$$

d) Perspective Correct Texture Mapping *(cont'd)*

- Also, equations (A) and (B) can imply:

$$u = cz + m \text{ (A)} \Rightarrow \quad \frac{u}{z} = c + \frac{m}{z} \qquad \text{(C)}$$

$$v = dz + n \text{ (B)} \Rightarrow \quad \frac{v}{z} = d + \frac{n}{z} \qquad \text{(D)}$$

d) Perspective Correct Texture Mapping *(cont'd)*

- Replacing the value of z in equation (C) and (D) leads to the following:

$$\frac{u}{z} = c + \frac{m(-a(x\_screen) + f)}{bf} = \frac{-ma}{bf} x\_screen + \left(c + \frac{m}{b}\right) \qquad (I)$$

$$\frac{v}{z} = d + \frac{n(-a(x\_screen) + f)}{bf} = \frac{-na}{bf} x\_screen + \left(d + \frac{n}{b}\right) \qquad (II)$$

d) Perspective Correct Texture Mapping *(cont'd)*

- As you can see, we wrote u/z and v/z as linear equations of x_screen which allows us to interpolate correct texture coordinates along x_screen.

- Afterwards, we divide u/z and v/z by 1/z (which we found in 1/z buffering) in order to get the texture coordinates u and v

d) Perspective Correct Texture Mapping *(cont'd)*

- Again, we use forward differencing to find the values of u/z. If x_screen increases by 1, what would be the change in u/z?

$$\left(\frac{u}{z}\right)' = \frac{-ma}{bf}(x\_screen+1)+\left(c+\frac{m}{b}\right) = \frac{-ma}{bf}x\_screen+\left(c+\frac{m}{b}\right)+\left(\frac{-ma}{bf}\right) = \frac{u}{z}+\left(\frac{-ma}{bf}\right)$$

d) Perspective Correct Texture Mapping *(cont'd)*

- Thus, when x_screen increments by 1, u/z increments by

$$\left(-\frac{ma}{bf} \right)$$