Realtime 3D Computer Graphics & Virtual Reality

Bitmaps and Textures

Imaging and Raster Primitives

Vicki Shreiner

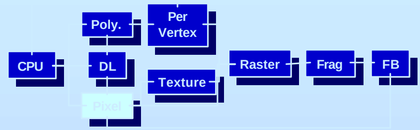
Imaging and Raster Primitives

- Describe OpenGL's raster primitives: bitmaps and image rectangles
- Demonstrate how to get OpenGL to read and render pixel rectangles

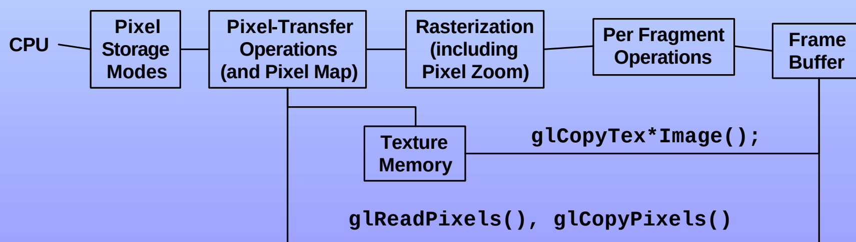
Pixel-based primitives

- Bitmaps
  - 2D array of bit masks for pixels
    - update pixel color based on current color
- Images
  - 2D array of pixel color information
    - complete color information for each pixel
- OpenGL doesn't understand image formats

Pixel Pipeline



- Programmable pixel storage and transfer operations



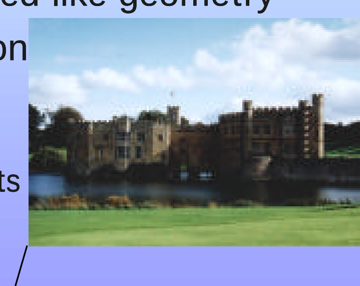
Positioning Image Primitives

`glRasterPos3f( x, y, z )`

– raster position transformed like geometry

– discarded if raster position outside of viewport

- may need to fine tune viewport for desired results

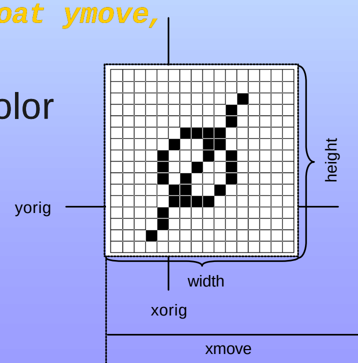


Raster Position

Rendering Bitmaps

```
glBitmap(  GLsizei width, GLsizei height,
           GLfloat xorig, GLfloat yorig,
           GLfloat xmove, GLfloat ymove,
           GLubyte *bitmap )
```

- render bitmap in current color at $\left(x - x_{orig}, y - y_{orig}\right)$
- advance raster position by $\left(x_{move}, y_{move}\right)$ after rendering



Rendering Fonts using Bitmaps

- OpenGL uses bitmaps for font rendering
  - each character is stored in a display list containing a bitmap
  - window system specific routines to access system fonts
    - **`glXUseXFont()`**
    - **`wglUseFontBitmaps()`**

Rendering Images

glDrawPixels(width, height, format, type, pixels)

- render pixels with lower left of image at current raster position
- numerous formats and data types for specifying storage in memory
  - best performance by using format and type that matches hardware

Reading Pixels

glReadPixels(x, y, width, height, format, type, pixels)

- read pixels form specified *(x,y)* position in framebuffer
- pixels automatically converted from framebuffer format into requested format and type

- Framebuffer pixel copy

  glCopyPixels(x, y, width, height, type)

Pixel Zoom

`glPixelZoom( x, y )`

– expand, shrink or reflect pixels around current raster position

– fractional zoom supported

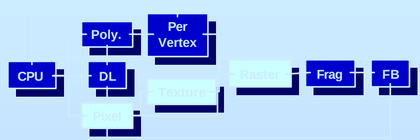
Storage and Transfer Modes

- Storage modes control accessing memory
  – byte alignment in host memory
  – extracting a subimage
- Transfer modes allow modify pixel values
  – scale and bias pixel component values
  – replace colors using pixel maps

Texture Mapping

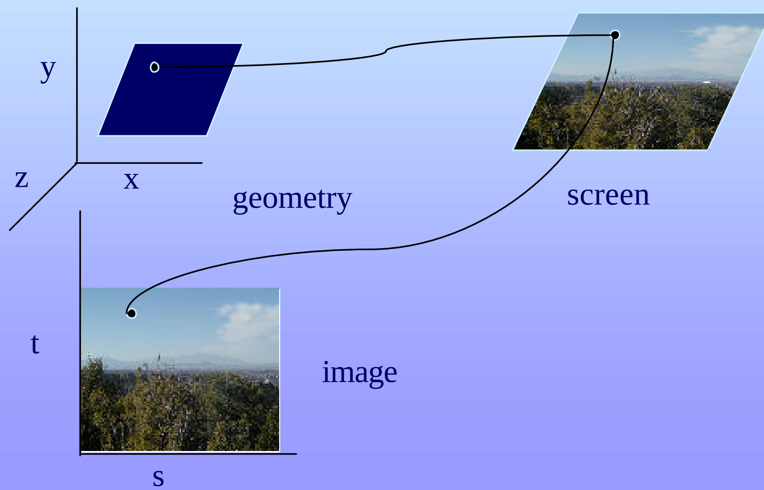
*(some taken from Ed Angel)

Texture Mapping



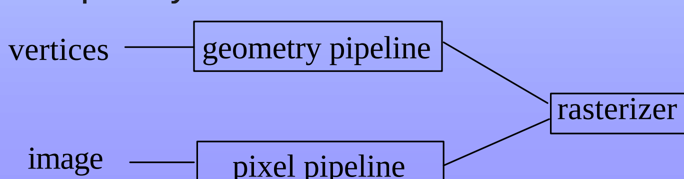
- Apply a 1D, 2D, or 3D image to geometric primitives
- Uses of Texturing
  - simulating materials
  - reducing geometric complexity
  - image warping
  - reflections

Texture Mapping

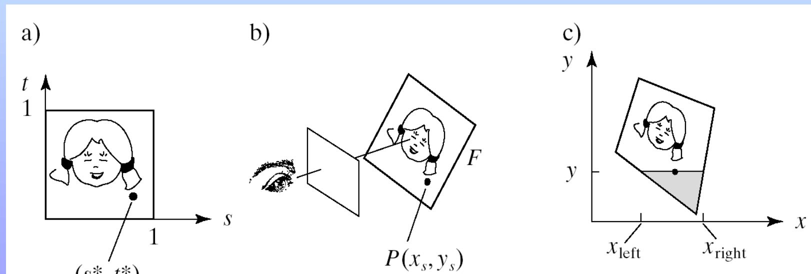


Texture Mapping and the OpenGL Pipeline

- Images and geometry flow through separate pipelines that join at the rasterizer
  – “complex” textures do not affect geometric complexity

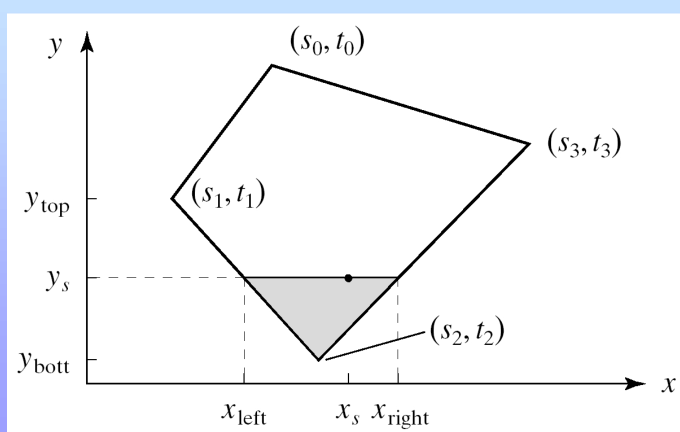


Rendering a texture

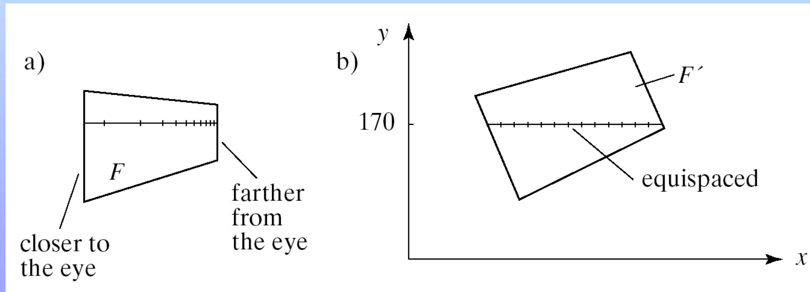


*Following pictures are courtesy of F.S.Hill Jr. “Computer Graphics using OpenGL 2nd”

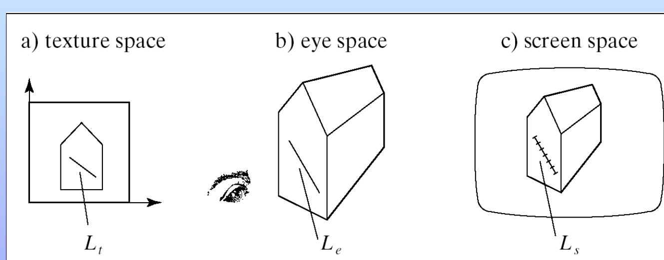
Rendering a texture: scanline



Rendering a texture: linear interpolation



Rendering a texture



- Affine and projective transformations preserve straightness.
- If moving equal steps across Ls, how to step in Lt?

Affine combination of two points

Given the linear combination of points $A = (A_1, A_2, A_3, 1)$ and $B = (B_1, B_2, B_3, 1)$ using the scalars s and t:

$$sA + tB = (sA_1 + tB_1, sA_2 + tB_2, sA_3 + tB_3, s + t)$$

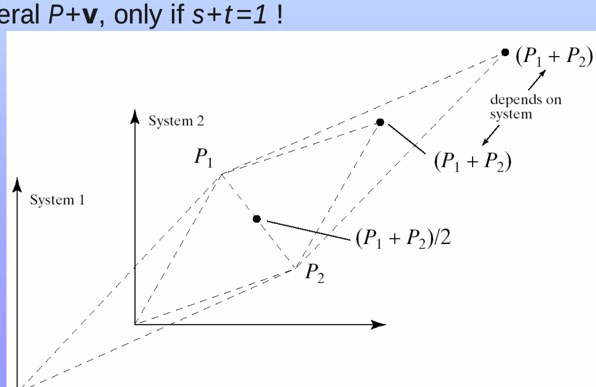
This is a valid vector if s+t=0. It is a valid point if s+t=1.

1. If the coefficients of a linear combination sum to unity we call it an affine combination.
2. Any affine combination of points is a legitimate point.

Why not building any linear combination of points $P = sA + tB$ if s +t do not sum to unity?

Affine combination of two points

-> A shift of the origin is the problem, let's shift it by vector **v**, so *A* is shifted to *A*+**v** and *B* is shifted to *B*+**v**. If *P* is a valid point it must be shifted to *P'*=*P*+**v**! But we have *P'*=*sA*+*tB*+*(s*+*t)***v**. This is not in general *P*+**v**, only if *s*+*t*=*1* !



Linear interpolation of two points

Let *P* be a point defined by a point *A* and a vector **v** scaled by *s* and substitute **v** with the difference of a point *B* and *A*.

$$P = A + s\mathbf{v} \Leftrightarrow P = A + s(B - A)$$

This can be rewritten as an affine combination of points as:

$$P = A + s(B - A) \Leftrightarrow P = sB + (1 - s)A$$

This performs a linear combination between points *A* and *B* !

For each component *c* of *P*, *Pc(s)* is the value which is the fraction *s* between *Ac* and *Bc*. This important operation has its own popular name ***lerp()*** (*linear interpolation*).

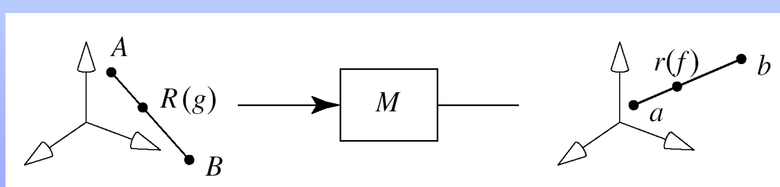
```
double lerp( double a, double b, double s){
  return ( a + (b – a) * s );}
```

Correspondence of motion along transformed lines

Let M be an affine or general perspective transformation.

The points A and B of a segment map to a and b.

The point R(g) maps to a point r(f).



How does g vary if f changes?

Why in the direction f->g? ->The process is to be embedded in the raster stage of the rendering pipeline!

Correspondence of motion along transformed lines

Let $\tilde{a} = (a_1, a_2, a_3, a_4)$ be the homogenous rep of *a, therefore*

$$a = \left(\frac{a_1}{a_4}, \frac{a_2}{a_4}, \frac{a_3}{a_4}\right)$$ is calculated by perspective division.

M maps *A* to *a* => $\tilde{a} = M(A,1)^T$ and $\tilde{b} = M(B,1)^T$

$R(g) = lerp(A, B, g)$ *maps to* $M(lerp(A, B, g),1)^T = lerp(\tilde{a}, \tilde{b}, g)$

$$= (lerp(a_1, b_1, g), lerp(a_2, b_2, g), lerp(a_3, b_3, g), lerp(a_4, b_4, g))$$

The latter being the homogenous coordinate version $\tilde{r}(f)$ of the point $r(f)$.

Correspondence of motion along transformed lines

$$(lerp(a_1, b_1, g), lerp(a_2, b_2, g), lerp(a_3, b_3, g), lerp(a_4, b_4, g))$$

Component wise (for one comp.) perspective division results in

$$r_1(f) = \frac{lerp(a_1, b_1, g)}{lerp(a_4, b_4, g)}$$ *but we also have* $r(f) = lerp(a, b, f)$

and hence (for one comp.) $$r_1(f) = lerp\left(\frac{a_1}{a_4}, \frac{b_1}{b_4}, f\right)$$

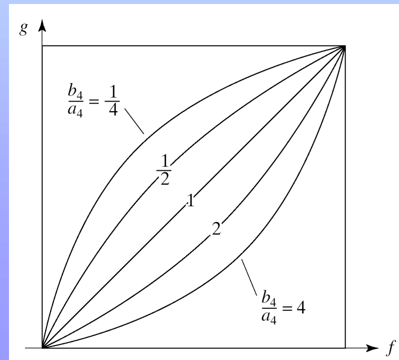
resulting in $$g = \frac{f}{lerp\left(\frac{b_4}{a_4}, 1, f\right)}$$

Correspondence of motion along transformed lines

$$g = \frac{f}{lerp\left(\frac{b_4}{a_4}, 1, f\right)}$$

R(g) maps to *r(f)* with different fractions for *f* and *g* !

$$g = f \Rightarrow f = 0 \vee f = 1 \vee b_4 = a_4$$



Finding the point

The point *R(g)* that maps to *r(f)* is as follows (for one comp.):

$$R_1 = \frac{lerp\left(\frac{A_1}{a_4}, \frac{B_1}{b_4}, f\right)}{lerp\left(\frac{1}{a_4}, \frac{1}{b_4}, f\right)}$$

Is there a difference if the matrix M is an affine or a perspective projection?

In the affine case, a_4 and b_4 are both unity and the relation between *R(g)* and *r(f)* degenerates to a linear dependency (remember the lerp() definition) and equal steps along *ab* correspond to equal steps along *AB*.

Finding the point

The perspective transformation case, given a matrix *M* (here the one that transforms from eye coordinates to clip coordinates):

$$M = \begin{bmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & c & d \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Given a point *A* this leads to:

$$M(A,1)^T = (NA_1, NA_2, cA_3 + d, -A_3)$$

The last component $a_4 = -A_3$ is the position of the point along the z-axis –the view plane normal– in camera coordinates (depth of point in front of the eye).

a_4, b_4 interpreted as the depth represent a line parallel to the view plane if they are equal, hence there is no foreshortening effect.

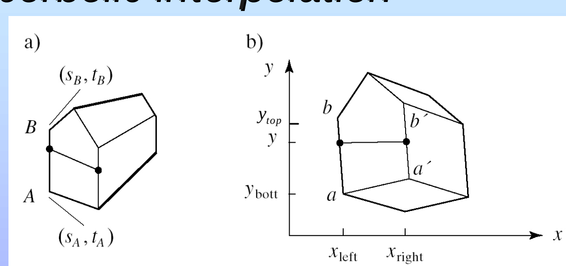
Texture processing during the scanline process: *hyperbolic interpolation*

We search for:

$(s_{left}, t_{left})\, and\, (s_{right}, t_{right})$

given:

$f = (y - y_{bott}) / \ (y_{top} - y_{bott})$



follows:

$$s_{(left)}(y) = \frac{lerp\left(\frac{s_A}{a_4}, \frac{s_B}{b_4}, f\right)}{lerp\left(\frac{1}{a_4}, \frac{1}{b_4}, f\right)}$$

How does it look like for $t_{(left)}$?

$$t_{(left)}(y) = \frac{lerp\left(\frac{t_A}{a_4}, \frac{t_B}{b_4}, f\right)}{lerp\left(\frac{1}{a_4}, \frac{1}{b_4}, f\right)}$$

Texture processing during the scanline process: *hyperbolic interpolation*

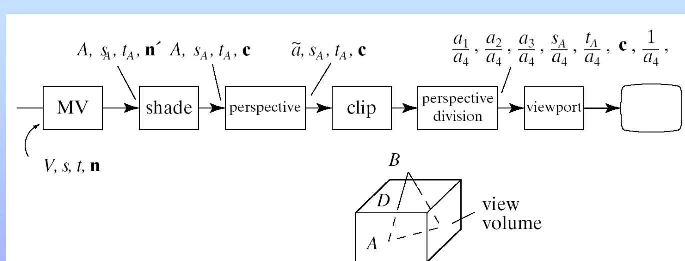
$$s_{(left)}(y) = \frac{lerp\left(\frac{s_A}{a_4}, \frac{s_B}{b_4}, f\right)}{lerp\left(\frac{1}{a_4}, \frac{1}{b_4}, f\right)}$$

- Same denominator for $s_{(left)}$ and $t_{(left)}$: A linear interpolation.
- Nominator is a linear interpolation of texture coordinates devided by a_4 and b_4.
- This is called *rational linear rendering* [Heckbert91] or *hyperbolic interpolation* [Blinn96].

Given y, the term $s_A/a_4, s_B/b_4, t_A/a_4, t_B/b_4, 1/a_4, 1/b_4$ is constant. Needed values for nominator and denominator can be found incrementally (see Gouraud shading).

Division is required for $s_{(left)}$ and $t_{(left)}$.

Scanline processing using *hyperbolic interpolation* in the pipeline



Each vertex *V* is associated with texture coords (*s*,*t*) and a normal **n**.

The modelview matrix M transforms into eye coords with $s_A = s, t_A = t$.

Perspective transformation alters only *A* which results into $\tilde{a}$.

What happens during clipping?

Hyperbolic interpolation in the pipeline

During clipping a new point $D=(d_1,d_2,d_3,d_4)$ is created by
$d_i = lerp\ (a_i,b_i,t), i=1,\dots,4,$ for some t.

The same is done for color and texture data. This creates a new vertex, for the given vertex we have the array

$$(a_1,a_2,a_3,a_4,s_A,t_A,c,1)$$

Which finally undergoes perspective division leading to

$$(x,y,z,1,s_A/a_4,t_A/a_4,c,1/a_4)$$

What is (x,y,z)?

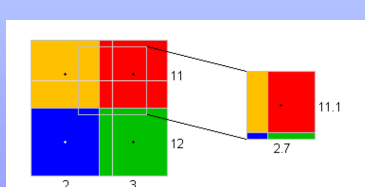
-> (x,y,z) = position of point in normalized device coordinates.

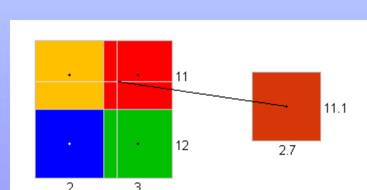
Why don’t we devide c?

Hyperbolic interpolation in the pipeline

What happens now if we calculate (s,t) in the described way?

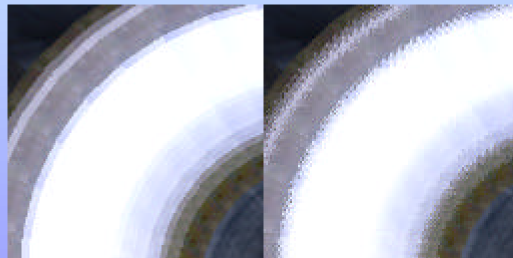
->Referencing to “arbitrary” points into (s,t) space might further produce visual artifacts due to sampling errors.





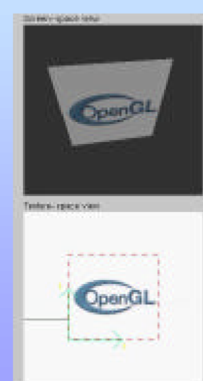
Hyperbolic interpolation in the pipeline

Point sampling vs. bilinear filtering.



Texture Example

- The texture (below) is a 256 x 256 image that has been mapped to a rectangular polygon which is viewed in perspective



Applying Textures I

- **Three steps**
  - ① specify texture
    - read or generate image
    - assign to texture
  - ② assign texture coordinates to vertices
  - ③ specify texture parameters
    - wrapping, filtering

Applying Textures II

- specify textures in texture objects
- set texture filter
- set texture function
- set texture wrap mode
- set optional perspective correction hint
- bind texture object
- enable texturing
- supply texture coordinates for vertex
  - coordinates can also be generated

Texture Objects

- Like display lists for texture images
  – one image per texture object
  – may be shared by several graphics contexts
- Generate texture names

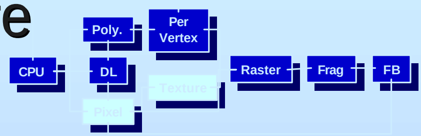
```
glGenTextures( n, *texIds );
```

Texture Objects (cont.)

- Create texture objects with texture data and state
- Bind textures before using

```
glBindTexture( target, id );
```

Specify Texture Image



- Define a texture image from an array of texels in CPU memory

```
glTexImage2D( target, level, components,
    w, h, border, format, type, *texels );
```

  - dimensions of image must be powers of 2
- Texel colors are processed by pixel pipeline
  - pixel scales, biases and lookups can be done

Converting A Texture Image

- If dimensions of image are not power of 2

```
gluScaleImage( format,
           w_in, h_in,type_in, *data_in,
           w_out, h_out,type_out, *data_out );
```

  - *data_in ***is for source image***
  - *data_out ***is for destination image***
- Image interpolated and filtered during scaling

Example

```
class RGB{ // holds a color triple – each with 256 possible
   intensities
    public: unsigned char r,g,b;
};

//The RGBpixmap class stores the number of rows and columns
//in the pixmap, as well as the address of the first pixel
//in memory:

class RGBpixmap{
  public:
   int nRows, nCols; // dimensions of the pixmap
   RGB* pixel; // array of pixels
   int readBMPFile(char * fname); // read BMP file into this
   pixmap
   void makeCheckerboard();
   void setTexture(GLuint textureName);
};
```

Example cont.

```
void RGBpixmap:: makeCheckerboard()
{  // make checkerboard patten
   nRows = nCols = 64;
   pixel = new RGB[3 * nRows * nCols];
I  f(!pixel){cout << ”out of memory!”;return;}
   long count = 0;
   for(int i = 0; i < nRows; i++)
        for(int j = 0; j < nCols; j++)
        {
                int c = (((i/8) + (j/8)) %2) * 255;
                pixel[count].r = c; // red
                pixel[count].g = c; // green
                pixel[count++].b = 0;      // blue
        }
}
```

Example cont.

```
void RGBpixmap :: setTexture(GLuint textureName)
{
   glBindTexture(GL_TEXTURE_2D,textureName);
   glTexParameteri(GL_TEXTURE_2D,
   GL_TEXTURE_MAG_FILTER,GL_NEAREST);
   glTexParameteri(GL_TEXTURE_2D,
   GL_TEXTURE_MIN_FILTER,GL_NEAREST);
   glTexImage2D(GL_TEXTURE_2D, 0,
   GL_RGB,nCols,nRows,0, GL_RGB,
                    GL_UNSIGNED_BYTE, pixel);
}
```

Specifying a Texture: Other Methods

- Use frame buffer as source of texture image
  - uses current buffer as source image

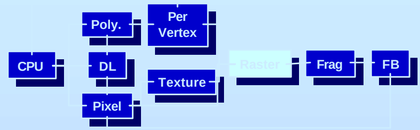
    `glCopyTexImage2D(...)`

    `glCopyTexImage1D(...)`
- Modify part of a defined texture

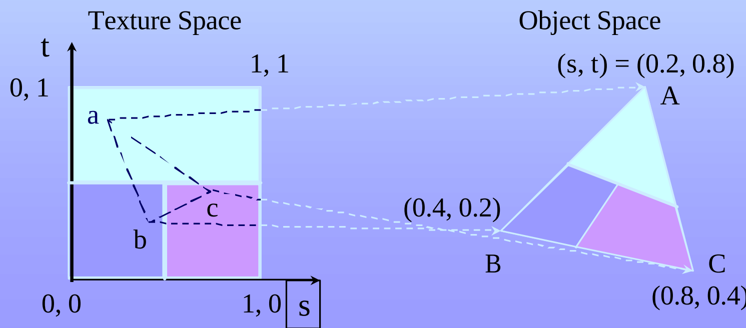
    `glTexSubImage2D(...)`

    `glTexSubImage1D(...)`
- Do both with *`glCopyTexSubImage2D(...)`*, etc.

Mapping a Texture



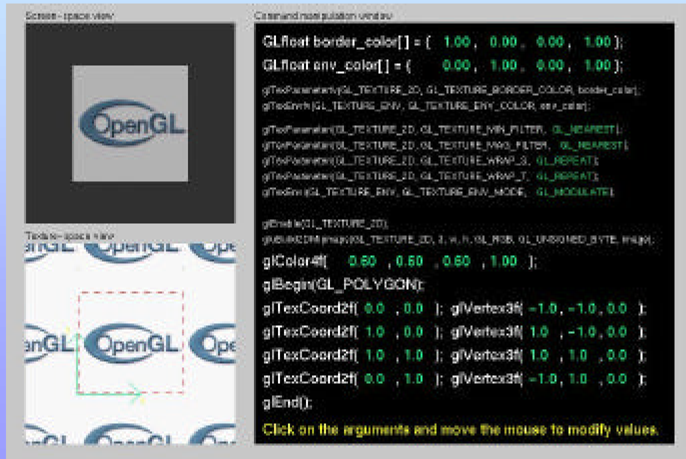
- Based on parametric texture coordinates
- `glTexCoord*()` specified at each vertex



Generating Texture Coordinates

- Automatically generate texture coords
  `glTexGen{ifd}[v]()`
- specify a plane
  - generate texture coordinates based upon distance from plane $Ax + By + Cz + D = 0$
- generation modes
  - `GL_OBJECT_LINEAR`
  - `GL_EYE_LINEAR`
  - `GL_SPHERE_MAP`

Tutorial: Texture



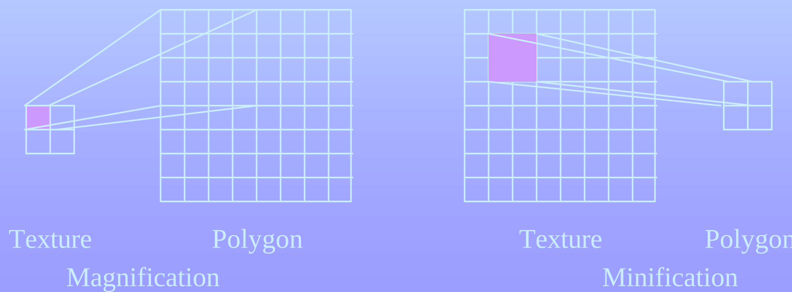
Texture Application Methods

- Filter Modes
  - minification or magnification
  - special mipmap minification filters
- Wrap Modes
  - clamping or repeating
- Texture Functions
  - how to mix primitive's color with texture's color
    - blend, modulate or replace texels

Filter Modes

Example:

`glTexParameteri( target, type, mode );`



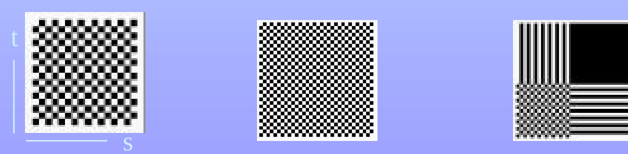
Mipmapped Textures

- Mipmap allows for prefiltered texture maps of decreasing resolutions
- Lessens interpolation errors for smaller textured objects
- Declare mipmap level during texture definition
  `glTexImage*D( GL_TEXTURE_*D, level, … )`
- GLU mipmap builder routines
  `gluBuild*DMipmaps( … )`
- OpenGL 1.2 introduces advanced LOD controls

Wrapping Mode

- Example:

```
glTexParameteri( GL_TEXTURE_2D,
     GL_TEXTURE_WRAP_S, GL_CLAMP )
glTexParameteri( GL_TEXTURE_2D,
     GL_TEXTURE_WRAP_T, GL_REPEAT )
```



Texture Functions

- Controls how texture is applied

```
glTexEnv{fi}[v]( GL_TEXTURE_ENV, prop,
                 param )
```

- ***GL_TEXTURE_ENV_MODE*** modes
  - **GL_MODULATE**
  - **GL_BLEND**
  - **GL_REPLACE**
- Set blend color with ***GL_TEXTURE_ENV_COLOR***

Perspective Correction Hint

- Texture coordinate and color interpolation
  - either linearly in screen space
  - or using depth/perspective values (slower)
- Noticeable for polygons "on edge"

```
glHint( GL_PERSPECTIVE_CORRECTION_HINT, hint )
```

where ***hint*** is one of

- *GL_DONT_CARE*
- *GL_NICEST*
- *GL_FASTEST*

Is There Room for a Texture?

- Query largest dimension of texture image
  - typically largest square texture
  - doesn't consider internal format size

```
glGetIntegerv( GL_MAX_TEXTURE_SIZE, &size )
```

- Texture proxy
  - will memory accommodate requested texture size?
  - no image specified; placeholder
  - if texture won't fit, texture state variables set to 0
    - doesn't know about other textures
    - only considers whether this one texture will fit all of memory

Texture Residency

- Working set of textures
  - high-performance, usually hardware accelerated
  - textures must be in texture objects
  - a texture in the *working set* is *resident*
  - for residency of current texture, check **GL_TEXTURE_RESIDENT** state
- If too many textures, not all are resident
  - can set priority to have some kicked out first
  - establish 0.0 to 1.0 priorities for texture objects

Advanced OpenGL Topics

Dave Shreiner

Advanced OpenGL Topics

- Display Lists and Vertex Arrays
- Alpha Blending and Antialiasing
- Using the Accumulation Buffer
- Fog
- Feedback & Selection
- Fragment Tests and Operations
- Using the Stencil Buffer