8. Discrete Techniques

Lecture Overview

- Buffers
- Texture Mapping
- OpenGL Texture Mapping
- Compositing and Blending

- Reading: ANG Ch. 8, except 8.5, 8.9-8.10, and 8.12-8.13

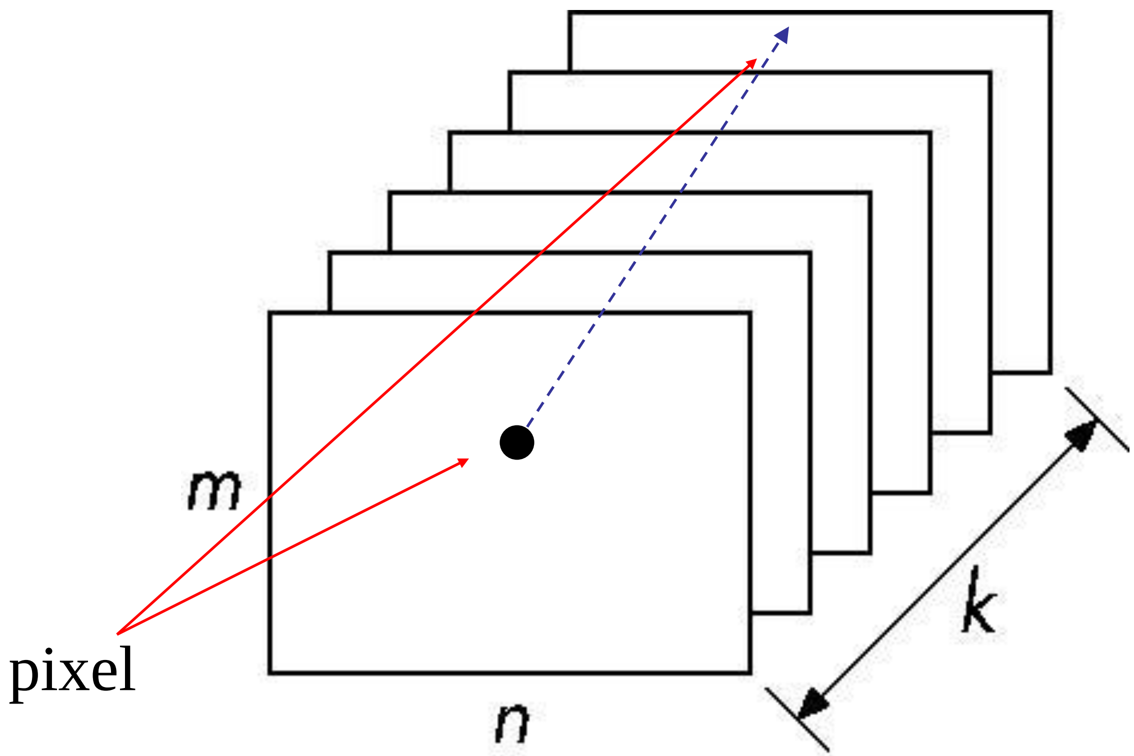
Buffers

Objectives

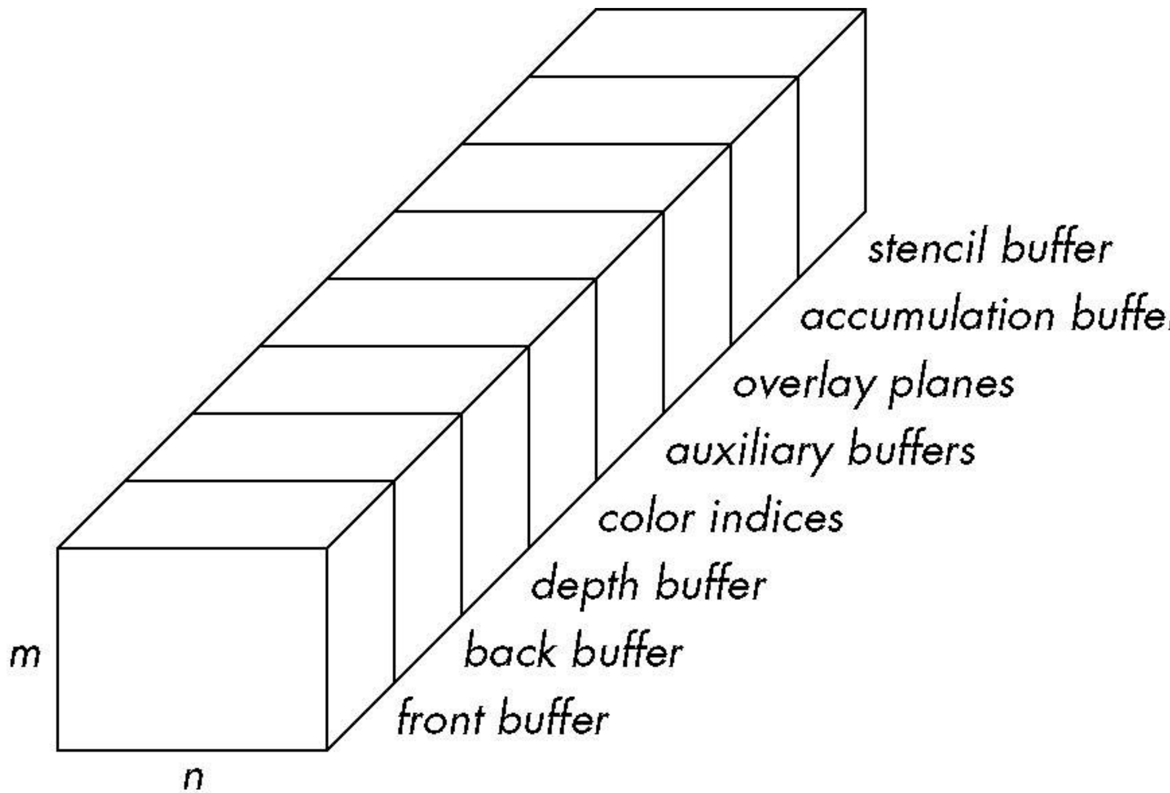
- Introduce additional OpenGL buffers
- Learn to read and write buffers
- Learn to use blending

Buffer

Define a buffer by its spatial resolution (n x m) and its depth (or precision) k, the number of bits/pixel



OpenGL Frame Buffer



OpenGL Buffers

- Color buffers can be displayed
  - Front
  - Back
  - Auxiliary
  - Overlay
- Depth
- Accumulation
  - High resolution buffer
- Stencil
  - Holds masks

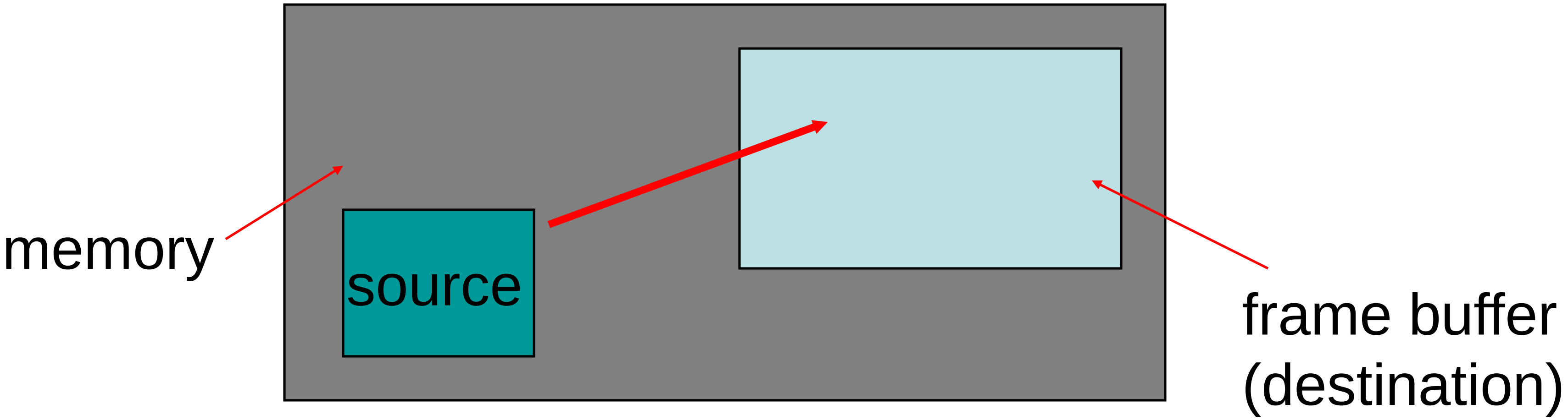
Digital Images

Creating a 512x512 image that consists of an 8x8 checkerboard of alternating *red* and *black* squares, such as we might use for a game.

```
GLubyte check[512][512][3];

int i, j;

for (i=0;i<512;i++)

  for (j=0;j<512;j++)

    {

      for (k=0;k<3;k++)

          if ((8*i+j/64)%64)  check[i][j][0]=255; // red

              else check[i][j][k]= 0;                  // black

     }
```

Writing in Buffers

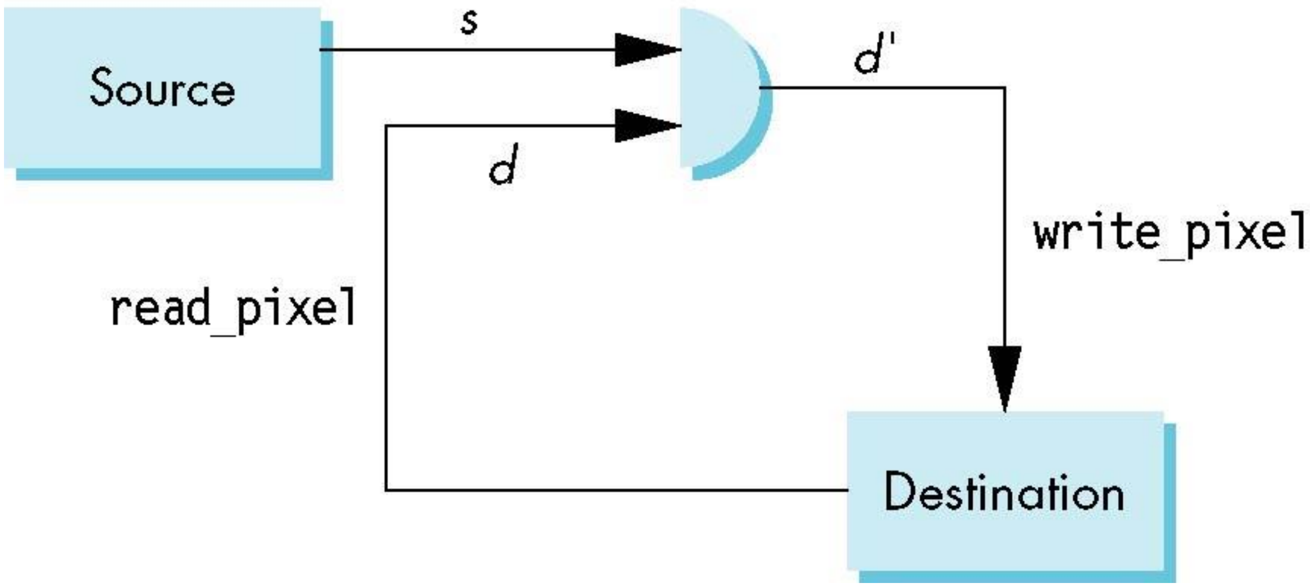
- Conceptually, we can consider all of memory as a large two-dimensional array of pixels
- We read and write rectangular block of pixels
  - *Bit block transfer* (*bitblt*) *operations*
- The frame buffer is part of this memory



writing into frame buffer

Writing Model

Read destination pixel before writing source



Bit Writing Modes

- Source and destination bits are combined bitwise
- 16 possible functions (one per column in table)

replace

XOR

OR

s	*d*	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

XOR mode

- Recall from Chapter 3 that we can use XOR by enabling logic operations and selecting the XOR write mode
- XOR is especially useful for swapping blocks of memory such as menus that are stored off screen

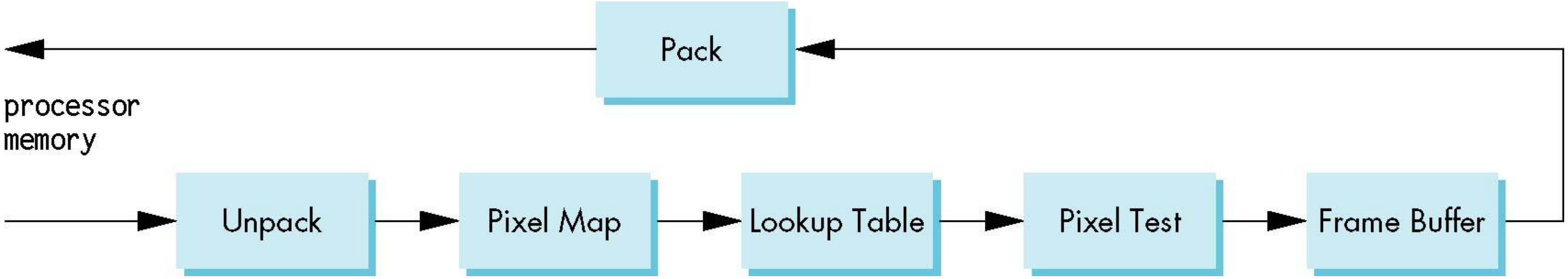
If S represents screen and M represents a menu the sequence

$$S \leftarrow S \oplus M$$
$$M \leftarrow S \oplus M$$
$$S \leftarrow S \oplus M$$

swaps the S and M

The Pixel Pipeline

- OpenGL has a separate pipeline for pixels
  - Writing pixels involves
    - Moving pixels from processor memory to the frame buffer
    - Format conversions
    - Mapping, Lookups, Tests
  - Reading pixels



Raster Position

- OpenGL maintains a *raster position* as part of the state
- Set by **`glRasterPos*()`**
  - **`glRasterPos3f(x, y, z);`**
- The raster position is a geometric entity
  - Passes through geometric pipeline
  - Eventually yields a 2D position in screen coordinates
  - This position in the frame buffer is where the next raster primitive is drawn

Buffer Selection

- OpenGL can draw into or read from any of the color buffers (front, back, auxiliary)
- Default to the back buffer
- Change with `glDrawBuffer` and `glReadBuffer`
- Note that format of the pixels in the frame buffer is different from that of processor memory and these two types of memory reside in different places
  - Need packing and unpacking
  - Drawing and reading can be slow

Bitmaps

- OpenGL treats 1-bit pixels (*bitmaps*) differently from multi-bit pixels (*pixelmaps*)
- Bitmaps are masks that determine if the corresponding pixel in the frame buffer is drawn with the *present raster color*
  - 0 ⇒ color unchanged
  - 1 ⇒ color changed based on writing mode
- Bitmaps are useful for raster text
  - GLUT font: **`GLUT_BIT_MAP_8_BY_13`**

Raster Color

- Same as drawing color set by **`glColor*()`**
- Fixed by last call to **`glRasterPos*()`**

```
glColor3f(1.0, 0.0, 0.0);
glRasterPos3f(x, y, z);
glColor3f(0.0, 0.0, 1.0);
glBitmap(…….
glBegin(GL_LINES);
  glVertex3f(…..)
```

- Geometry drawn in blue
- Ones in bitmap use a drawing color of red

Drawing Bitmaps

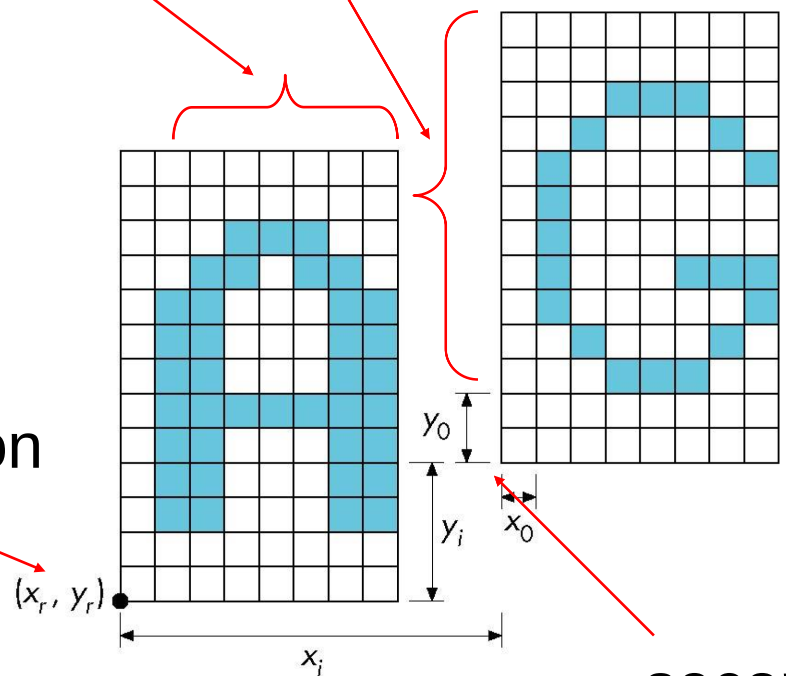
`glBitmap(width, height, x0, y0, xi, yi, bitmap)`

offset from raster position

increments in raster position after bitmap drawn

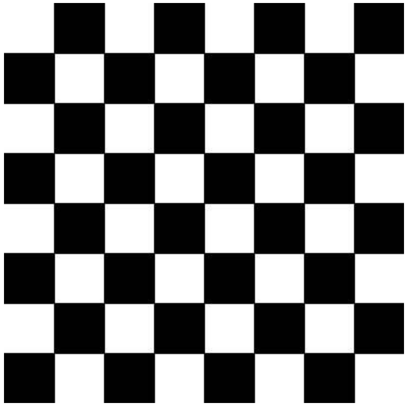
first raster position

second raster position



Example: Checker Board

```
GLubyte wb[2] = {0 x 00, 0 x ff};
GLubyte check[512];
int i, j;
for(i=0; i<64; i++) for (j=0; j<64, j++)
   check[i*8+j] = wb[(i/8+j)%2];

glBitmap( 64, 64, 0.0, 0.0, 0.0, 0.0, check);
```



Define a 128-character 8x13 font using a display list for each character.

```
GLubyte my_font[128][13];
:
Base=glGenLists(128)
for (i=0; i<128;i++)
{
   glNewList(base+i, GL_COMPILE);
   glBitMap(8, 13,0.0, 2.0, 10.0, 0.0, my_font[i]);
   glEndList();
}
```

Pixel Maps

- OpenGL works with rectangular arrays of pixels called pixel maps or images
- Pixels are in one byte (8 bit) chunks
  - Luminance (gray scale) images 1 byte/pixel
  - RGB 3 bytes/pixel
- Three functions
  - Draw pixels: processor memory to frame buffer
  - Read pixels: frame buffer to processor memory
  - Copy pixels: frame buffer to frame buffer

OpenGL Pixel Functions

```
glReadPixels(x,y,width,height,format,type,myimage)
```

start pixel in frame buffer

size

type of image

type of pixels

pointer to processor memory

```
GLubyte myimage[512][512][3];
glReadPixels(0,0, 512, 512, GL_RGB,
      GL_UNSIGNED_BYTE, myimage);

glDrawPixels(width,height,format,type,myimage)
```

starts at raster position

Image Formats

- We often work with images in a standard format (JPEG, TIFF, GIF)
- How do we read/write such images with OpenGL?
- No support in OpenGL
  - OpenGL knows nothing of image formats
  - Some code available on Web
  - Can write readers/writers for some simple formats in OpenGL

Displaying a PPM Image

- PPM is a very simple format
- Each image file consists of a header followed by all the pixel data
- Header

P3
comment 1
comment 2
.
#comment n
rows columns maxvalue
pixels

Reading the Header

```
FILE *fd;
int  k, nm;
char c;
int i;
char b[100];
float s;
int red, green, blue;
printf("enter file name\n");
scanf("%s", b);
fd = fopen(b, "r");
fscanf(fd,"%[^\n] ",b);
if(b[0]!='P'|| b[1] != '3'){
      printf("%s is not a PPM file!\n", b);
      exit(0);
}
printf("%s is a PPM file\n",b);
```

check for "P3" in first line

Reading the Header (cont)

```
fscanf(fd, "%c",&c);
while(c == '#')
{
      fscanf(fd, "%[^\n] ", b);
      printf("%s\n",b);
      fscanf(fd, "%c",&c);
}
      ungetc(c,fd);
```

skip over comments by
looking for # in first column

Reading the Data

```
fscanf(fd, "%d %d %d", &n, &m, &k);
printf("%d rows  %d columns  max value= %d\n",n,m,k);

nm = n*m;
image=malloc(3*sizeof(GLuint)*nm);
s=255./k;
```

scale factor

```
for(i=0;i<nm;i++)
{
      fscanf(fd,"%d %d %d",&red, &green, &blue );
      image[3*nm-3*i-3]=red;
      image[3*nm-3*i-2]=green;
      image[3*nm-3*i-1]=blue;
}
```

Scaling the Image Data

We can scale the image in the pipeline

```
glPixelTransferf(GL_RED_SCALE, s);
glPixelTransferf(GL_GREEN_SCALE, s);
glPixelTransferf(GL_BLUE_SCALE, s);
```

We may have to swap bytes when we go from processor memory to the frame buffer depending on the processor. If so, we can use

```
glPixelStorei(GL_UNPACK_SWAP_BYTES,GL_TRUE);
```

The display callback

```
void display()
{
 glClear(GL_COLOR_BUFFER_BIT);
 glRasterPos2i(0,0);
 glDrawPixels(n,m,GL_RGB,
    GL_UNSIGNED_INT, image);
 glFlush();
}
```

Texture Mapping

Objectives

- Introduce Mapping Methods
  - Texture Mapping
  - Environment Mapping
  - Bump Mapping
- Consider basic strategies
  - Forward vs backward mapping
  - Point sampling vs area averaging

The Limits of Geometric Modeling

- Although graphics cards can render over 10 million polygons per second, that number is insufficient for many phenomena
  - Clouds
  - Grass
  - Terrain
  - Skin

Modeling an Orange

- Consider the problem of modeling an orange (the fruit)
- Start with an orange-colored sphere
  - Too simple
- Replace sphere with a more complex shape
  - Does not capture surface characteristics (small dimples)
  - Takes too many polygons to model all the dimples

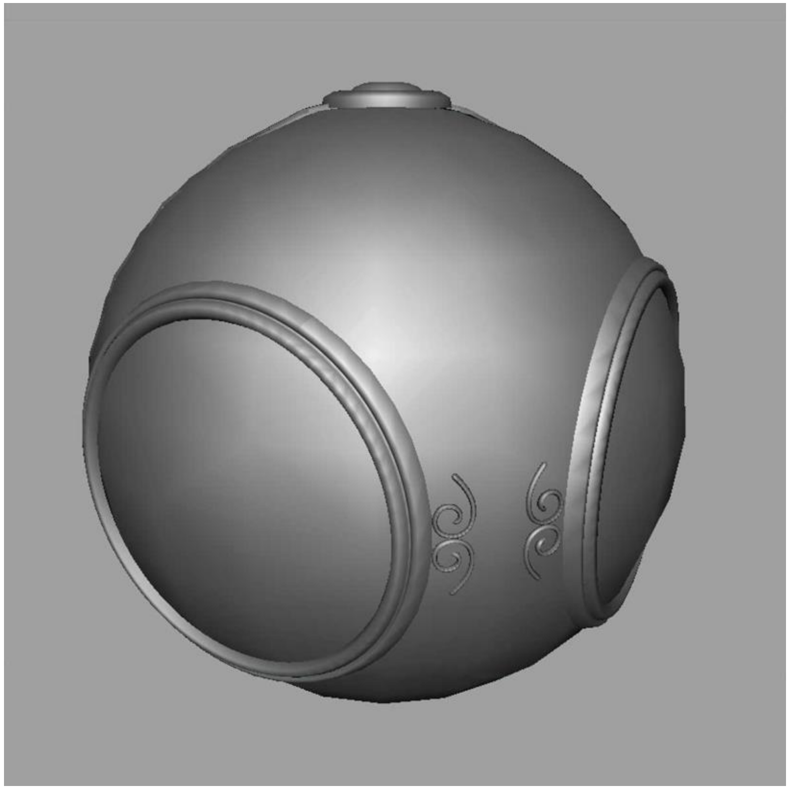
Modeling an Orange (2)

- Take a picture of a real orange, scan it, and "paste" onto simple geometric model
  - This process is known as texture mapping
- Still might not be sufficient because resulting surface will be smooth
  - Need to change local shape
  - Bump mapping

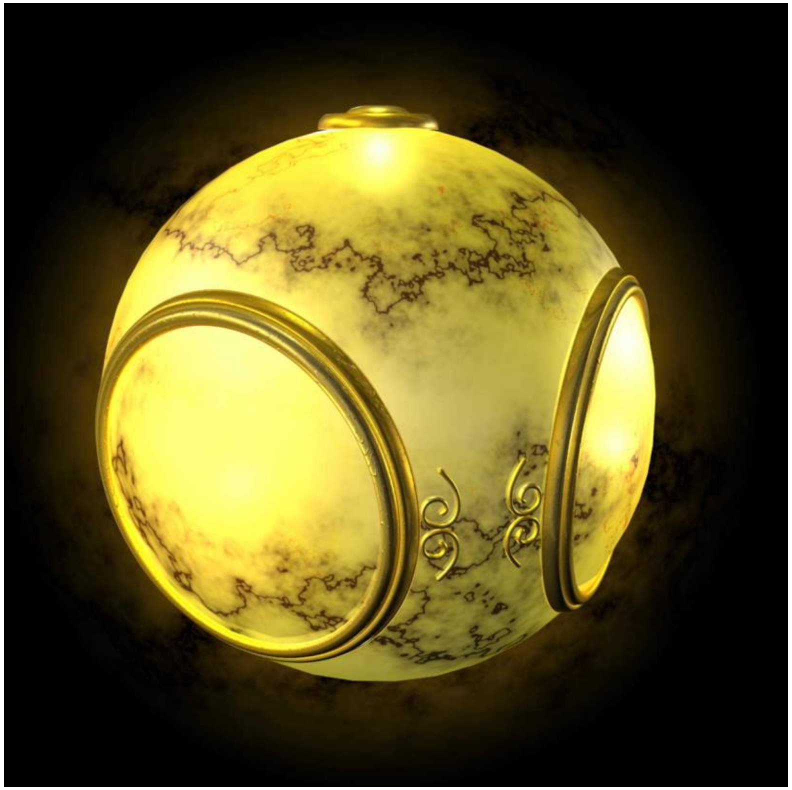
Three Types of Mapping

- Texture Mapping
  - Uses images to fill inside of polygons
- Environment (reflection mapping)
  - Uses a picture of the environment for texture maps
  - Allows simulation of highly specular surfaces
- Bump mapping
  - Emulates altering normal vectors during the rendering process

Texture Mapping



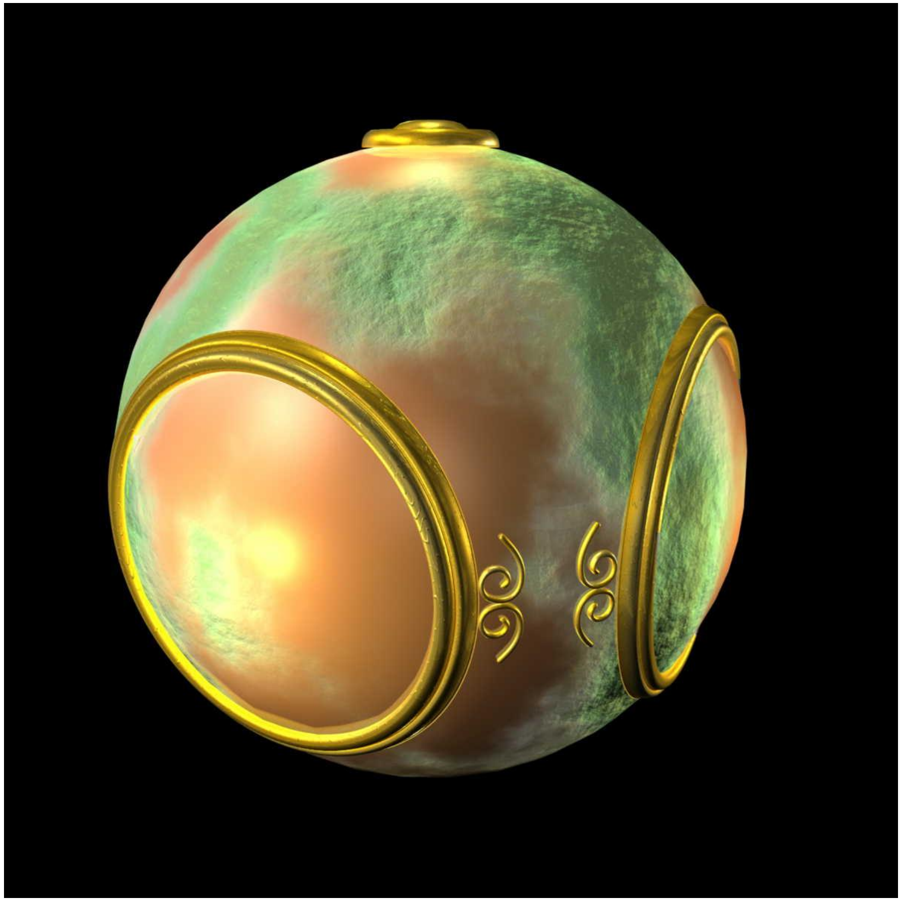
geometric model



texture mapped

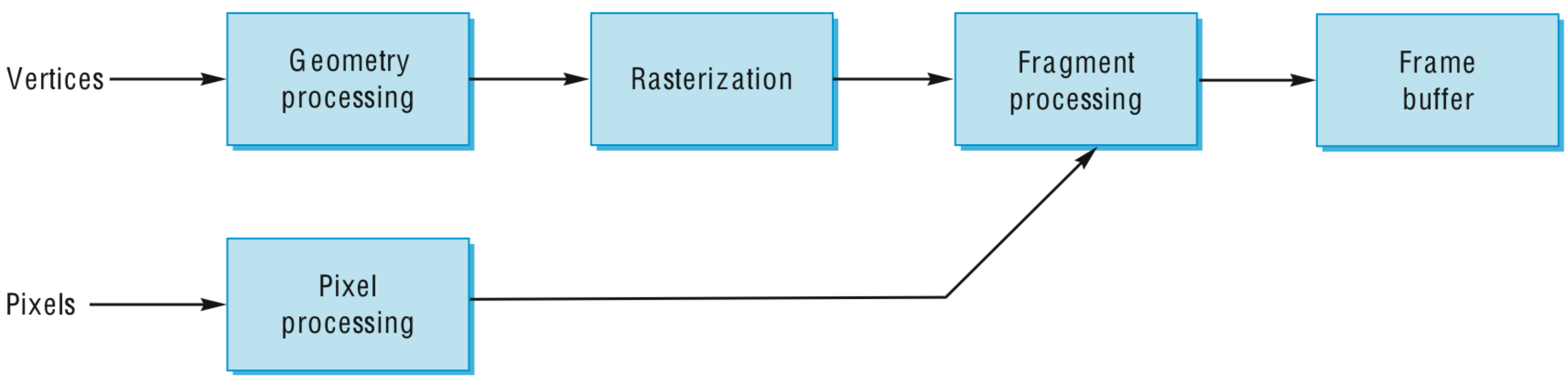
Environment Mapping

Bump Mapping



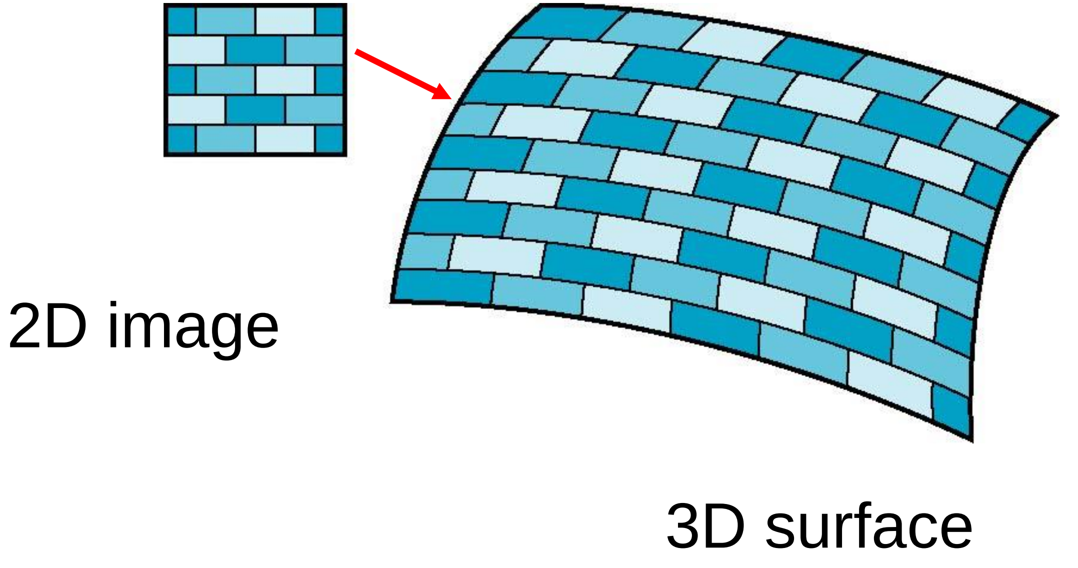
Where does mapping take place?

- Mapping techniques are implemented at the end of the rendering pipeline
  - Very efficient because few polygons make it past the clipper



Is it simple?

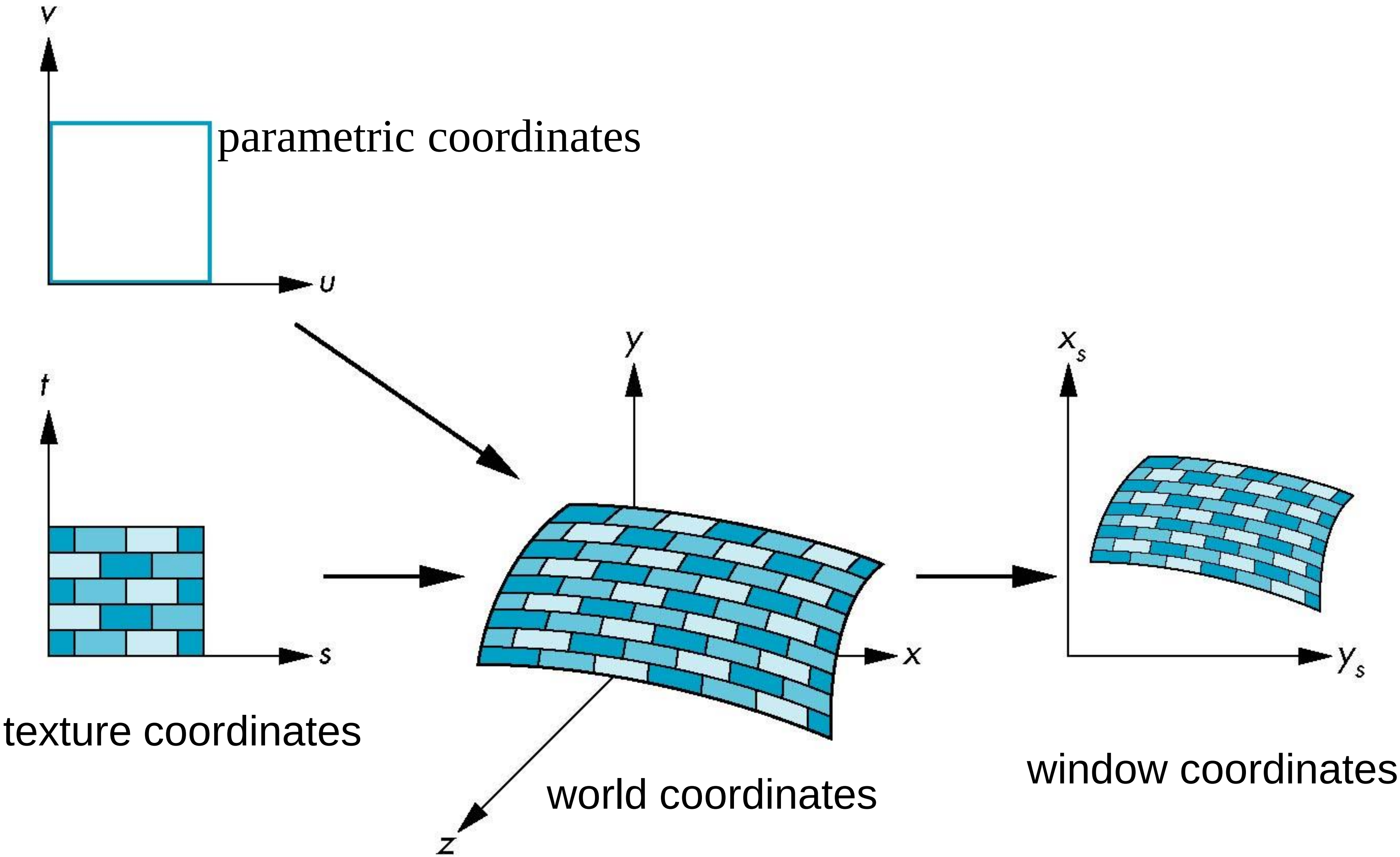
- Although the idea is simple---map an image to a surface---there are 3 or 4 coordinate systems involved



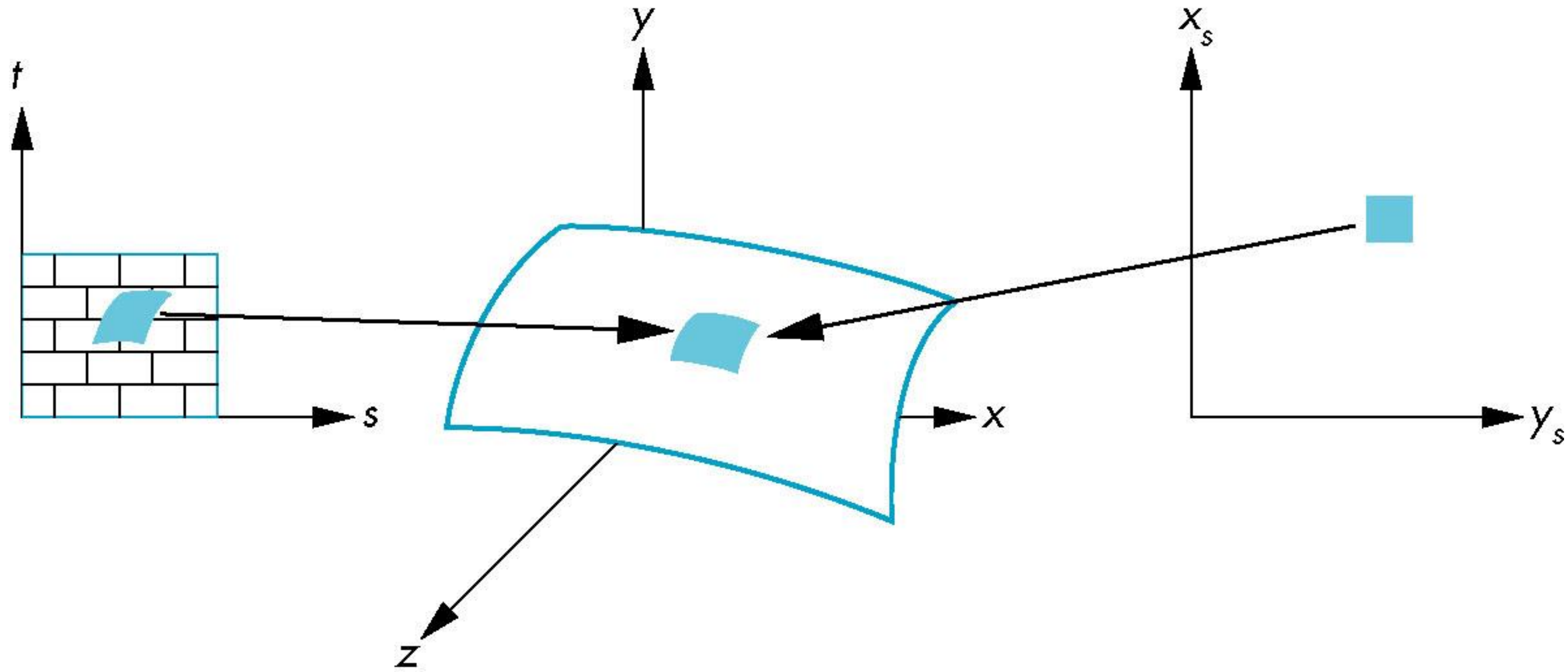
Coordinate Systems

- Parametric coordinates
  - –May be used to model curves and surfaces
- Texture coordinates
  - –Used to identify points in the image to be mapped
- Object or World Coordinates
  - –Conceptually, where the mapping takes place
- Window Coordinates
  - –Where the final image is really produced

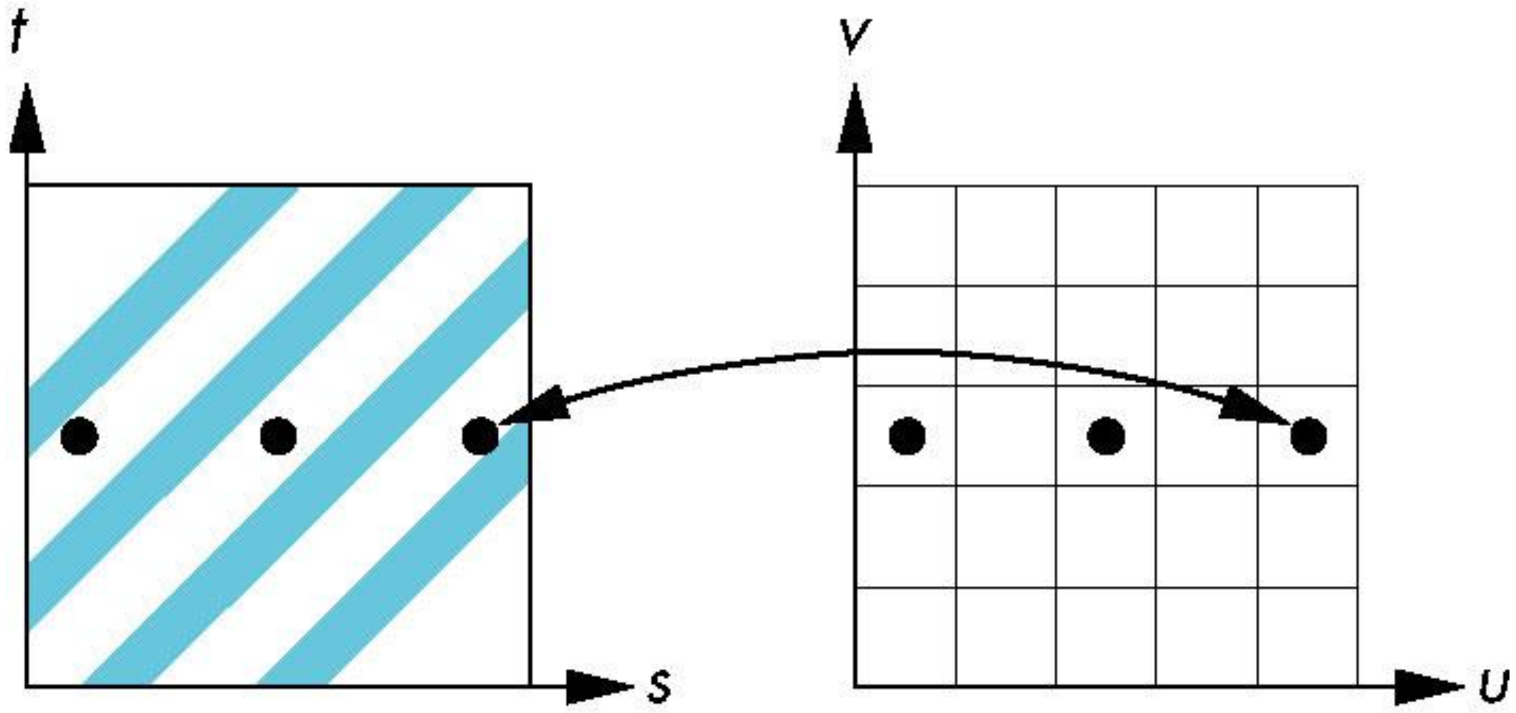
Texture Mapping



Preimages of a pixel



Aliasing in texture generation



Mapping Functions

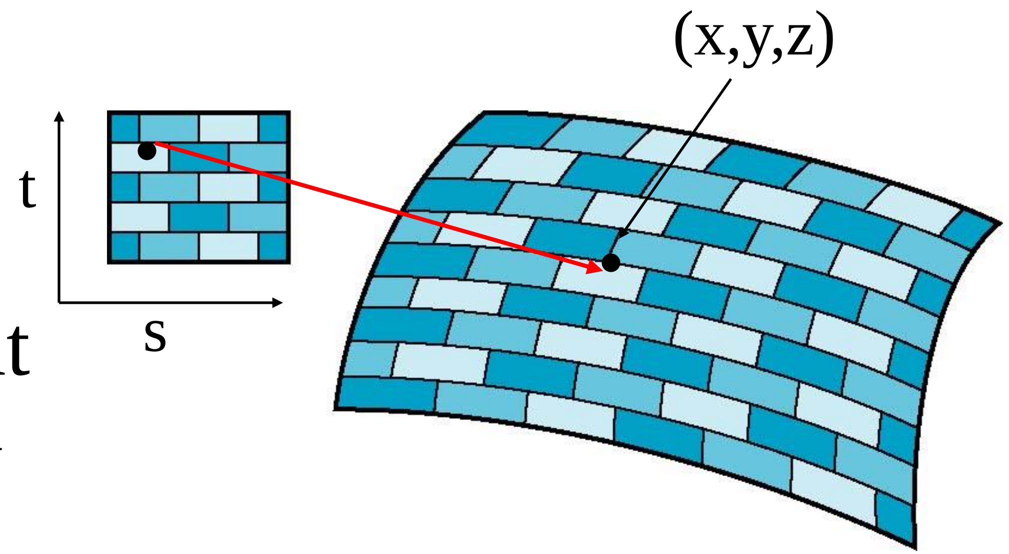
- Basic problem is how to find the maps
- Consider mapping from texture coordinates to a point a surface
- Appear to need three functions

$x = x(s,t)$

$y = y(s,t)$

$z = z(s,t)$

- But we really want to go the other way



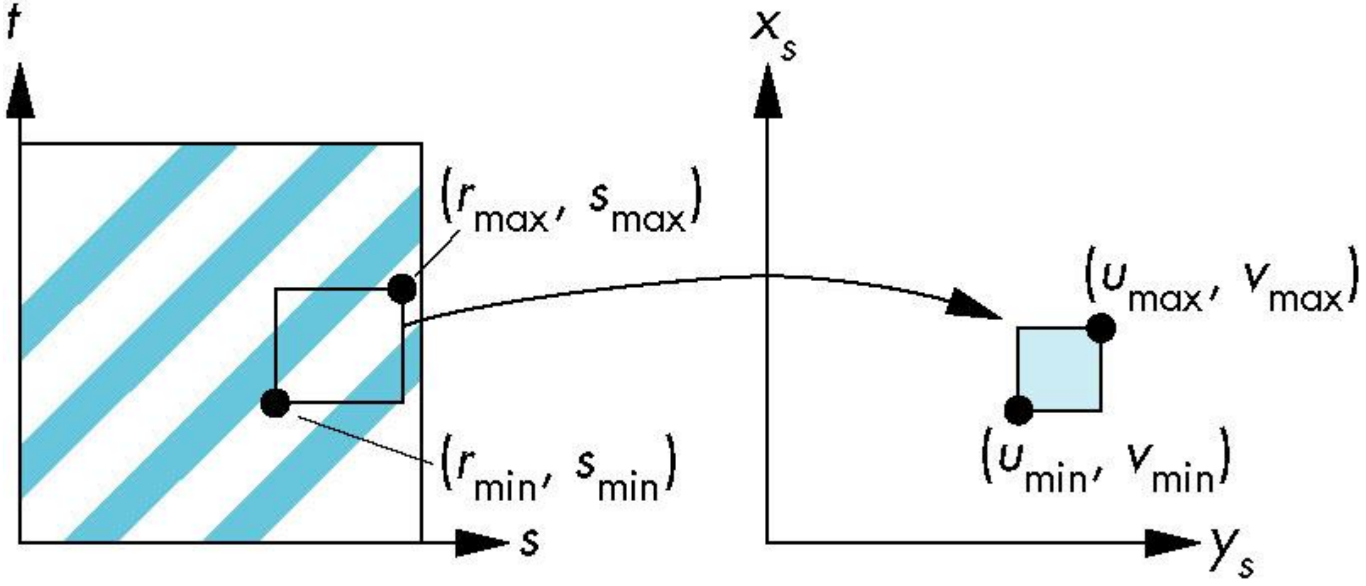
Backward Mapping

- We really want to go backwards
  - Given a pixel, we want to know to which point on an object it corresponds
  - Given a point on an object, we want to know to which point in the texture it corresponds
- Need a map of the form

  $s = s(x,y,z)$

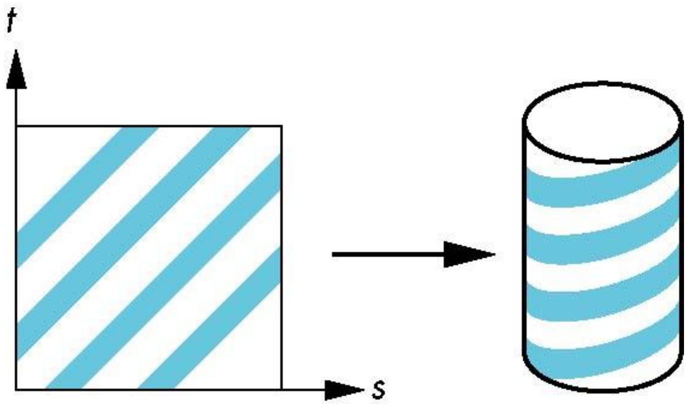
  $t = t(x,y,z)$
- Such functions are difficult to find in general

Linear Texture Mapping



Two-part mapping

- One solution to the mapping problem is to first map the texture to a simple intermediate surface
- Example: map to cylinder



Cylindrical Mapping

parametric cylinder

x = r cos 2p u

y = r sin 2pu

z = v/h

maps rectangle in u,v space to cylinder of radius r and height h in world coordinates

s = u

t = v

maps from texture space

Spherical Map

We can use a parametric sphere

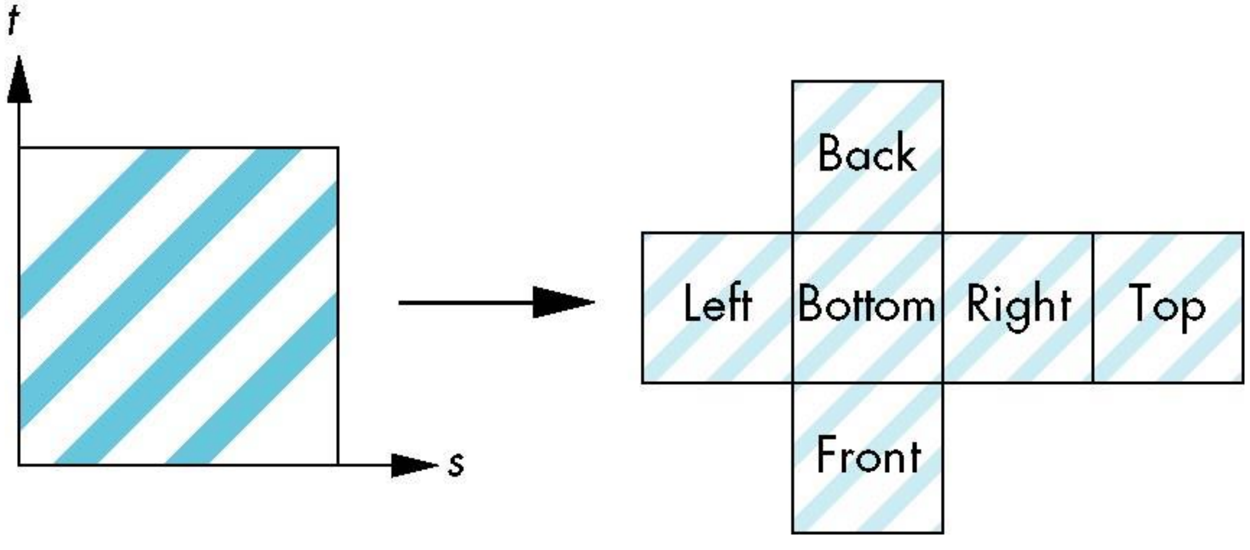
$$x = r \cos 2pu$$
$$y = r \sin 2pu \cos 2pv$$
$$z = r \sin 2pu \sin 2pv$$

in a similar manner to the cylinder
but have to decide where to put
the distortion

Spheres are used in environmental maps

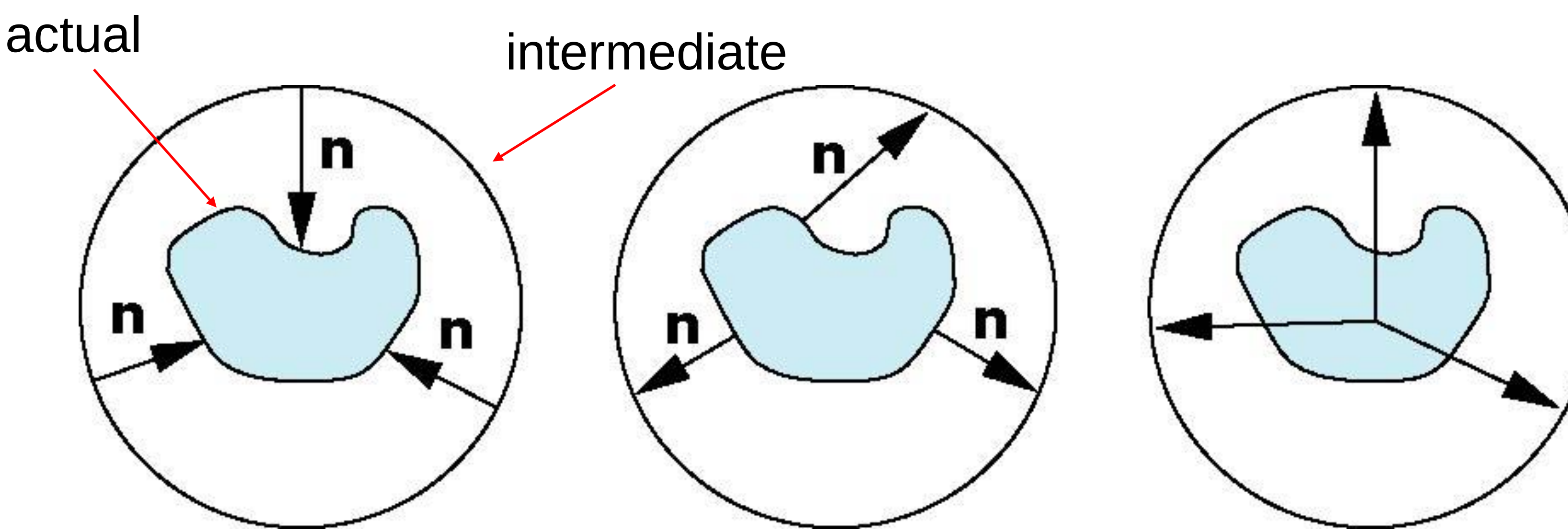
Box Mapping

- Easy to use with simple orthographic projection
- Also used in environment maps



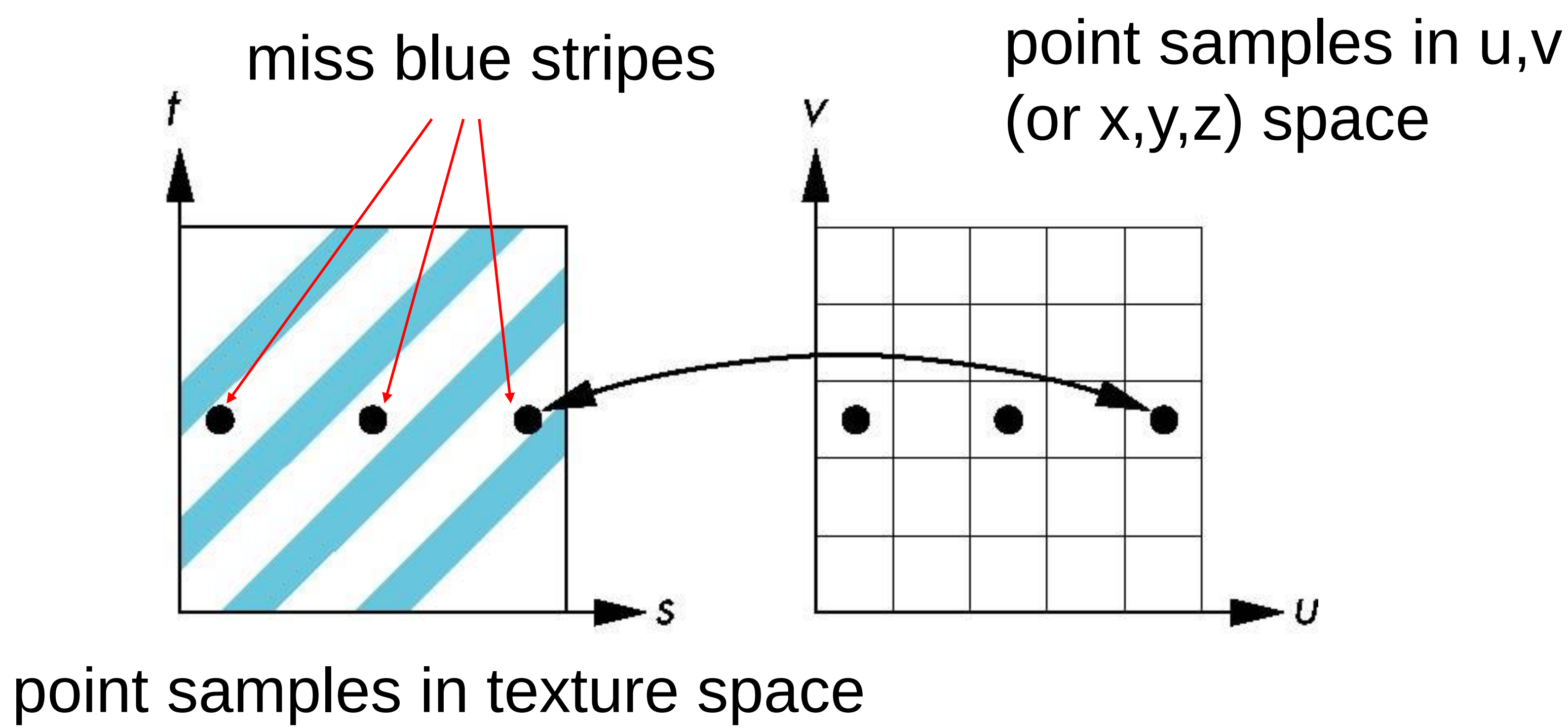
Second Mapping

- Map from intermediate object to actual object
  - –Normals from intermediate to actual
  - –Normals from actual to intermediate
  - –Vectors from center of intermediate



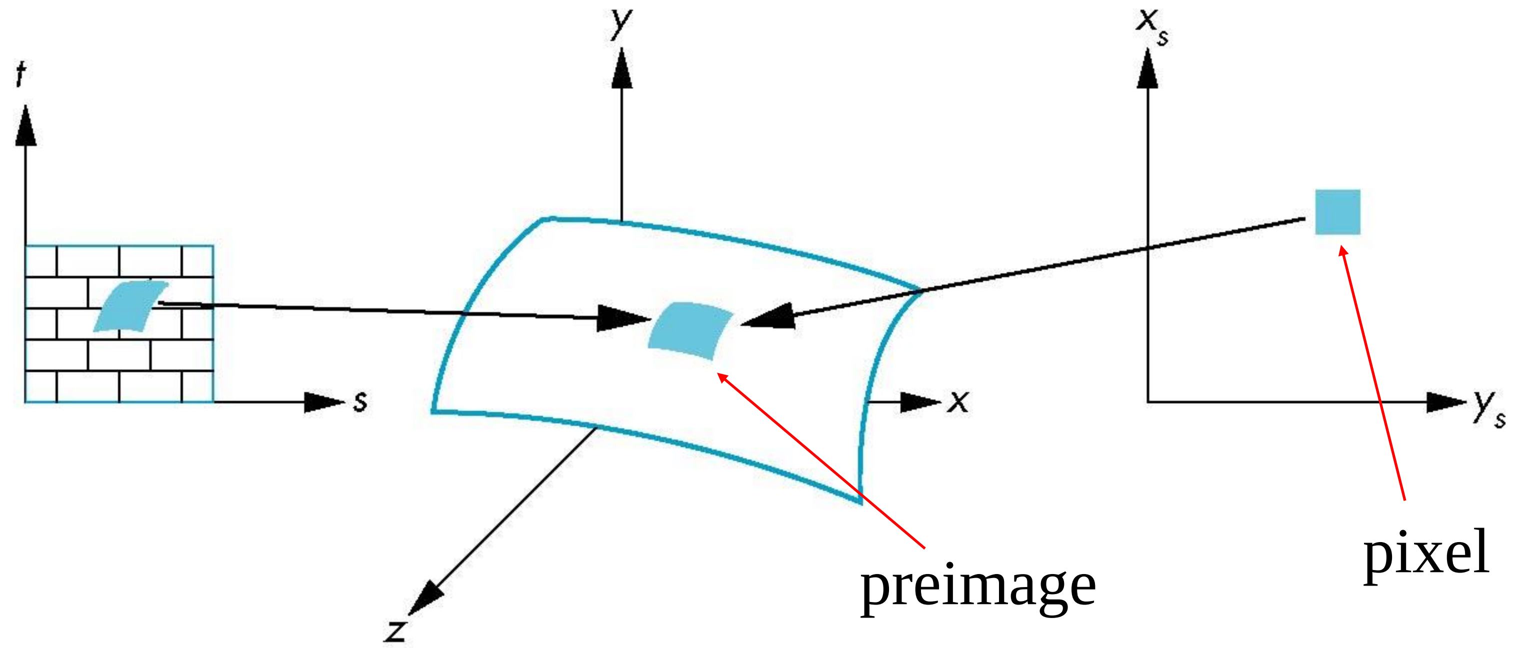
Aliasing

- Point sampling of the texture can lead to aliasing errors



Area Averaging

A better but slower option is to use *area averaging*



Note that *preimage* of pixel is curved

OpenGL Texture Mapping

Objectives

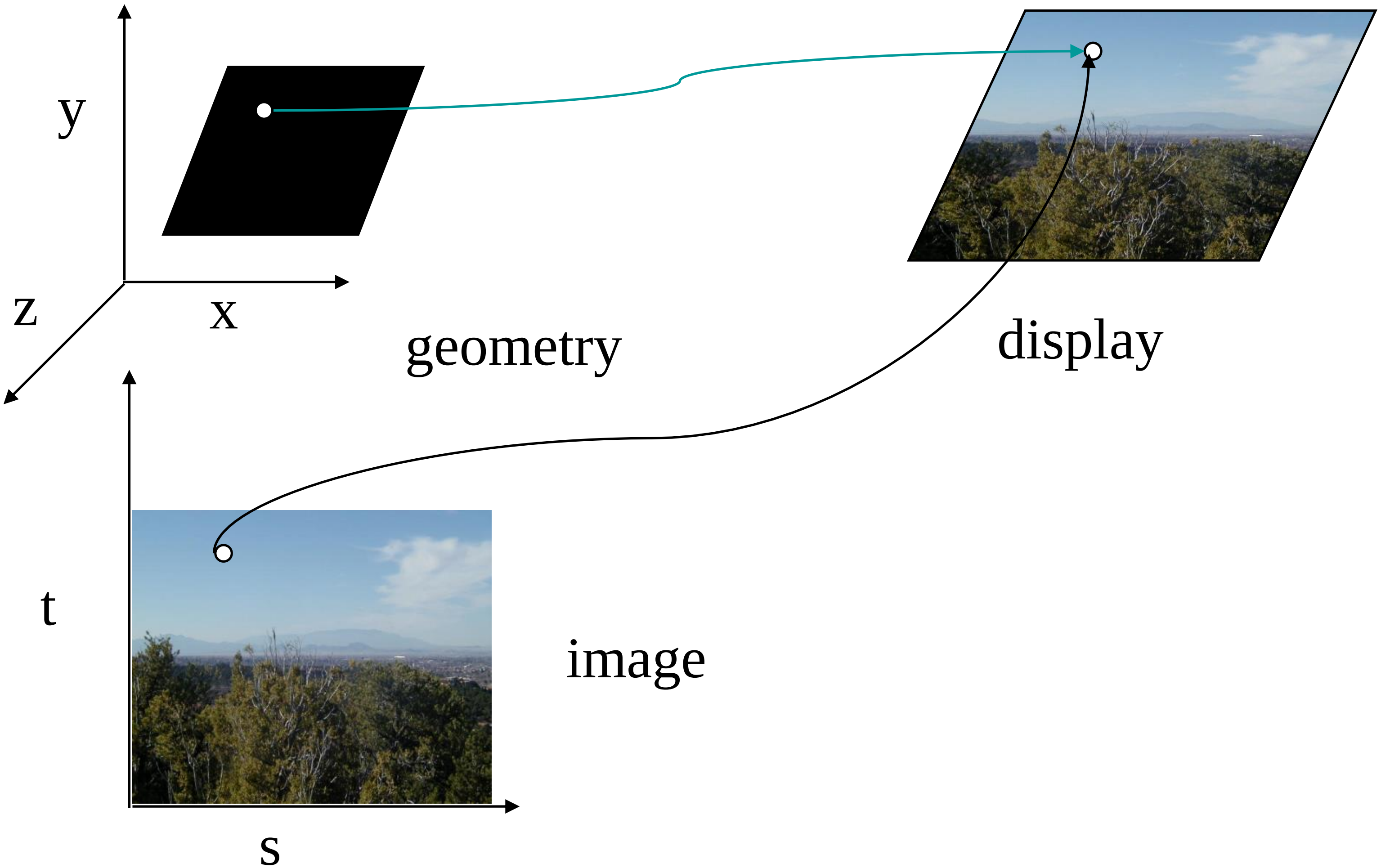
- Introduce the OpenGL texture functions and options

Basic Stragegy

Three steps to applying a texture

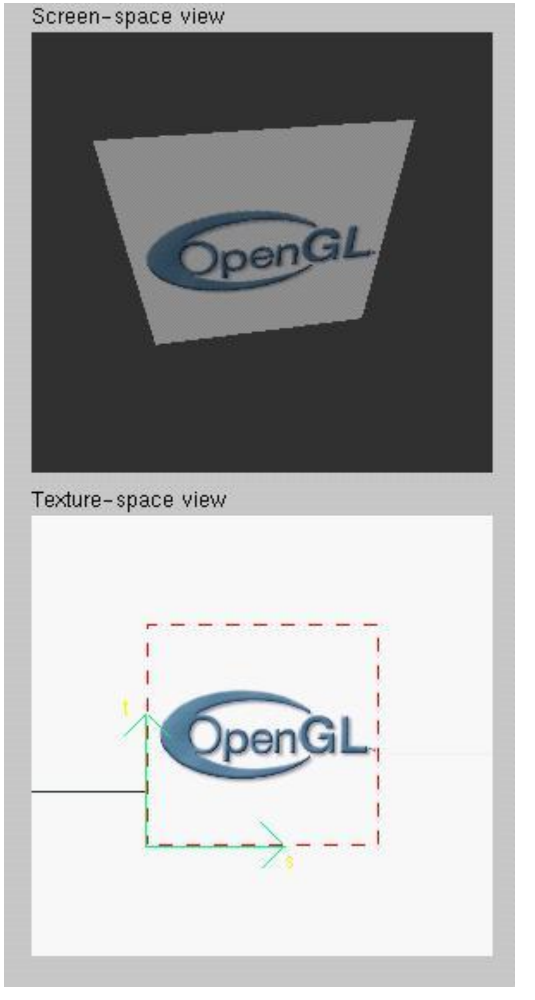
1. specify the texture
   - read or generate image
   - assign to texture
   - enable texturing
2. assign texture coordinates to vertices
   - Proper mapping function is left to application
3. specify texture parameters
   - wrapping, filtering

Texture Mapping



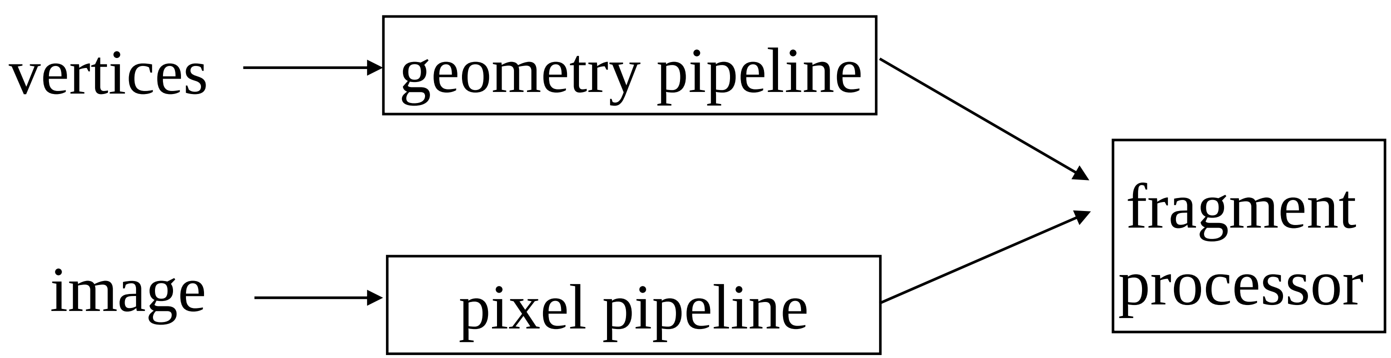
Texture Example

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



Texture Mapping and the OpenGL Pipeline

- Images and geometry flow through separate pipelines that join during fragment processing
  - “complex” textures do not affect geometric complexity



Specifying a Texture Image

- Define a texture image from an array of *texels* (texture elements) in CPU memory
  ```
  Glubyte my_texels[512][512];
  ```
- Define as any other pixel map
  - Scanned image
  - Generate by application code
- Enable texture mapping
  - `glEnable(GL_TEXTURE_2D)`
  - OpenGL supports **1-4 dimensional** texture maps

Define Image as a Texture

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

`target`: type of texture, e.g. **`GL_TEXTURE_2D`**

`level`: used for mipmapping (discussed later)

`components`: elements per texel

`w, h`: width and height of **`texels`** in pixels

`border`: used for smoothing (discussed later)

`format and type`: describe texels

`texels`: pointer to texel array

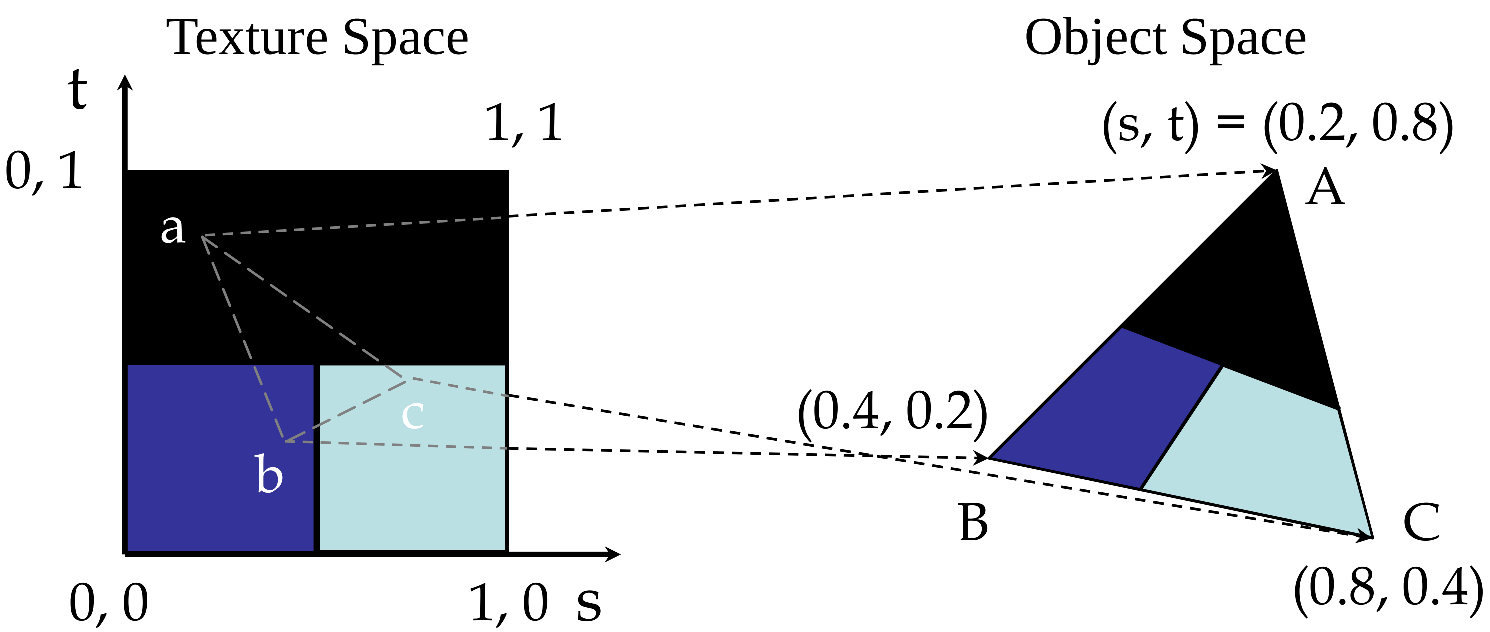
```
glTexImage2D(GL_TEXTURE_2D, 0, 3, 512, 512, 0,
 GL_RGB, GL_UNSIGNED_BYTE, my_texels);
```

Converting A Texture Image

- OpenGL requires texture dimensions to be powers of 2
- If dimensions of image are not powers of 2
  - **`gluScaleImage( format, w_in, h_in, type_in, *data_in, w_out, h_out, type_out, *data_out );`**
  - **`data_in`** is source image
  - **`data_out`** is for destination image
- Image interpolated and filtered during scaling

Mapping a Texture

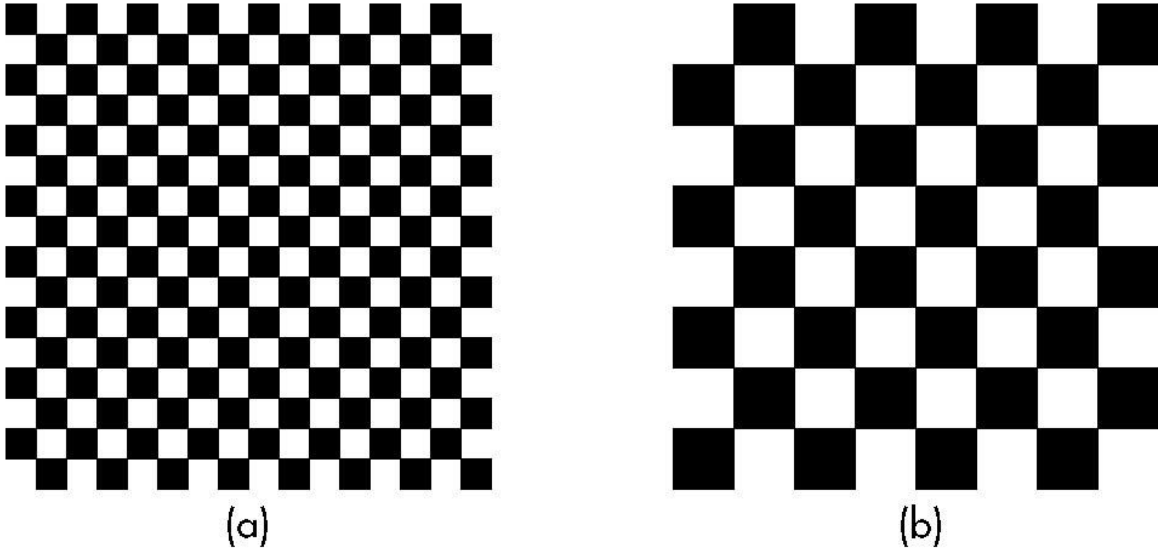
- Based on parametric texture coordinates
- **`glTexCoord*()`** specified at each vertex



Typical Code

```
glBegin(GL_POLYGON);
 glColor3f(r0, g0, b0); //if no shading used
 glNormal3f(u0, v0, w0); // if shading used
 glTexCoord2f(s0, t0);
 glVertex3f(x0, y0, z0);
 glColor3f(r1, g1, b1);
 glNormal3f(u1, v1, w1);
 glTexCoord2f(s1, t1);
 glVertex3f(x1, y1, z1);
      .
      .
glEnd();
```

Note that we can use vertex arrays to increase efficiency



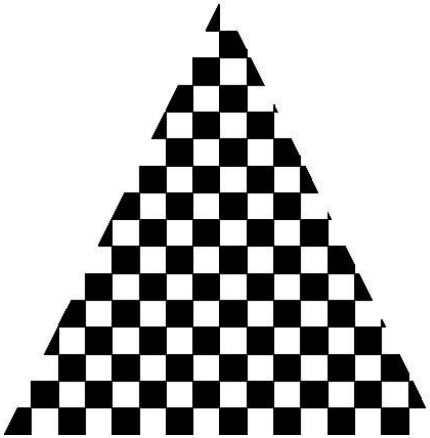
Mapping of checkerboard texture to a quadrilateral. (a) Using the entire texel array. (b) Using part of the texel array.

Interpolation

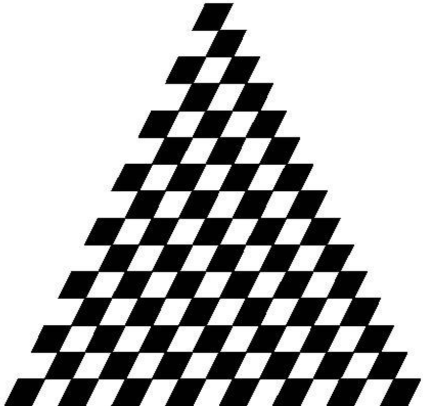
OpenGL uses interpolation to find proper texels from specified texture coordinates

Can be distortions

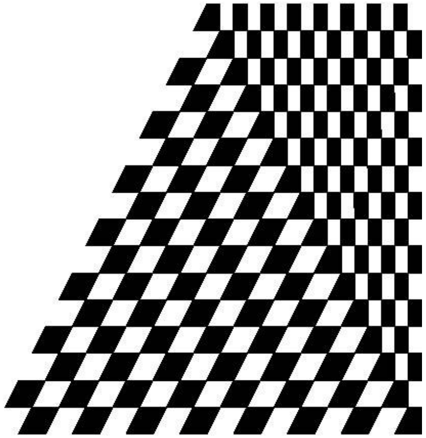
good selection of tex coordinates



poor selection of tex coordinates



texture stretched over trapezoid showing effects of bilinear interpolation



Texture Parameters

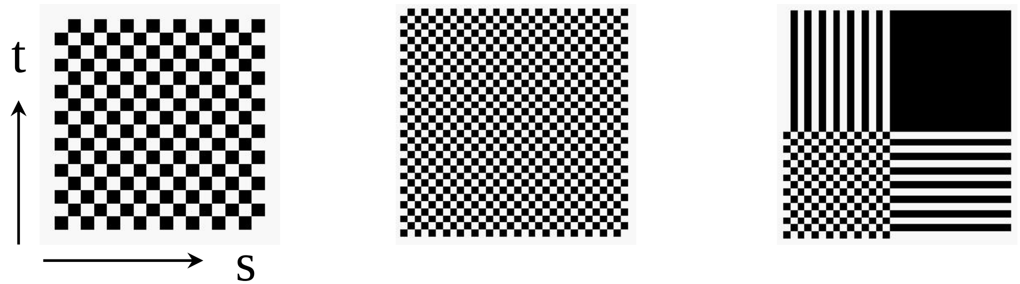
- OpenGL has a variety of parameters that determine how texture is applied
  - Wrapping parameters determine what happens if s and t are outside the (0,1) range
  - Filter modes allow us to use area averaging instead of point samples
  - Mipmapping allows us to use textures at multiple resolutions
  - Environment parameters determine how texture mapping interacts with shading

Wrapping Mode

Clamping: if s,t > 1 use 1, if s,t <0 use 0

Wrapping: use s,t modulo 1

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



texture

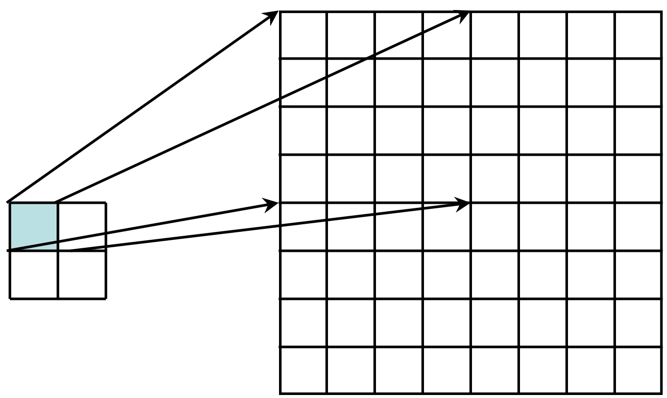
GL_REPEAT wrapping

GL_CLAMP wrapping

Magnification and Minification

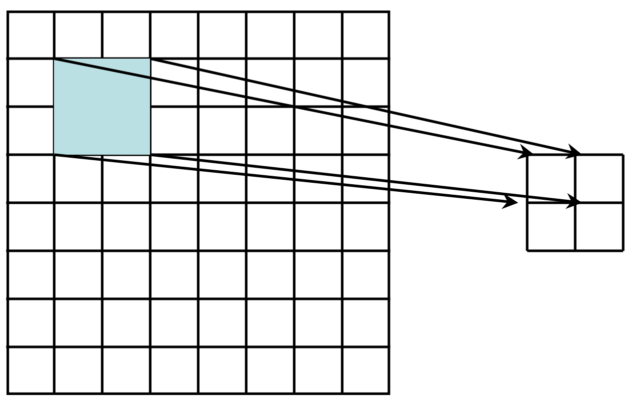
More than one texel can cover a pixel (*minification*) or more than one pixel can cover a texel (*magnification*)

Can use point sampling (nearest texel) or linear filtering (2 x 2 filter) to obtain texture values



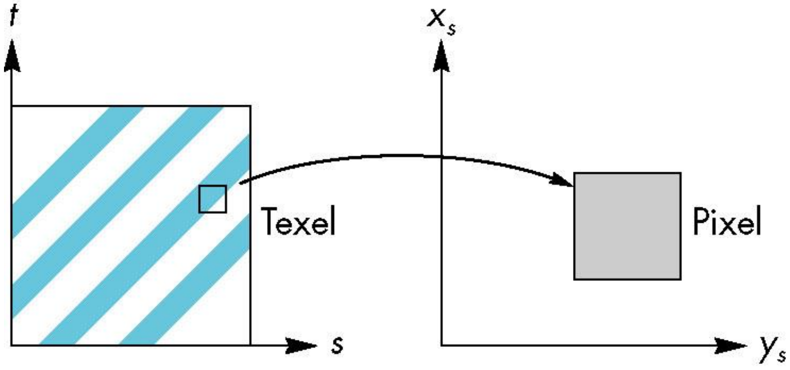
Texture Polygon

Magnification

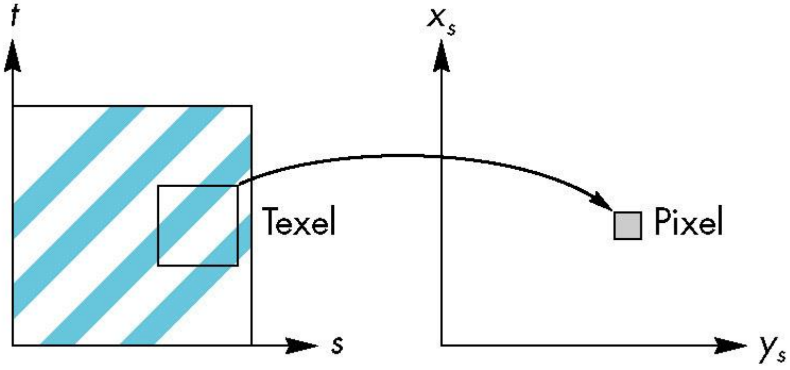


Texture Polygon

Minification



(a)



(b)

Minification

Magnification

Filter Modes

Modes determined by

- **`glTexParameteri( target, type, mode )`**

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXURE_MAG_FILTER,
                GL_NEAREST);

glTexParameteri(GL_TEXTURE_2D, GL_TEXURE_MIN_FILTER,
                GL_LINEAR);
```

Note that **linear filtering** requires a border of an extra texel for filtering at edges (border = 1)

Mipmapped Textures

- *Mipmapping* allows for prefiltered texture maps of decreasing resolutions
- Lessens interpolation errors for smaller textured objects
- Declare mipmap level during texture definition

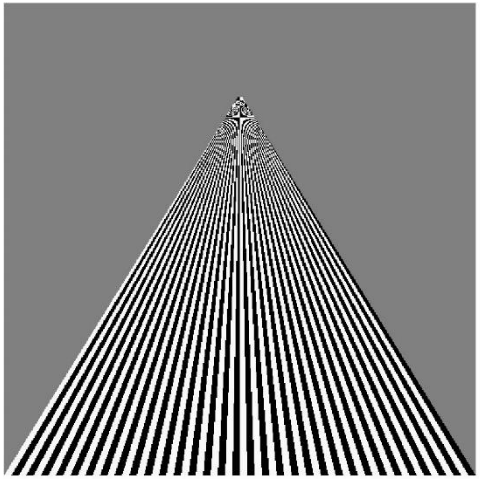
```
glTexImage2D( GL_TEXTURE_*D, level, … )
```

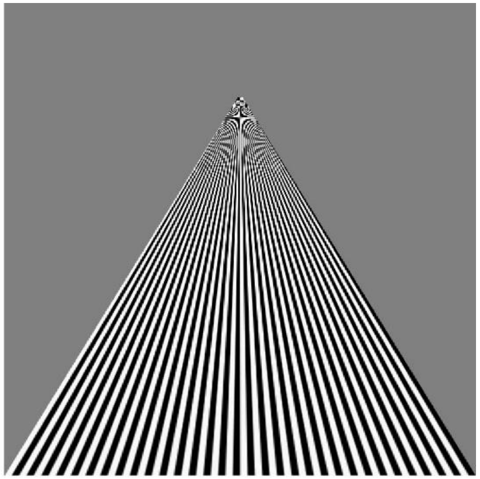
- GLU mipmap builder routines will build all the textures from a given image

```
gluBuild*DMipmaps( … )
```

Example

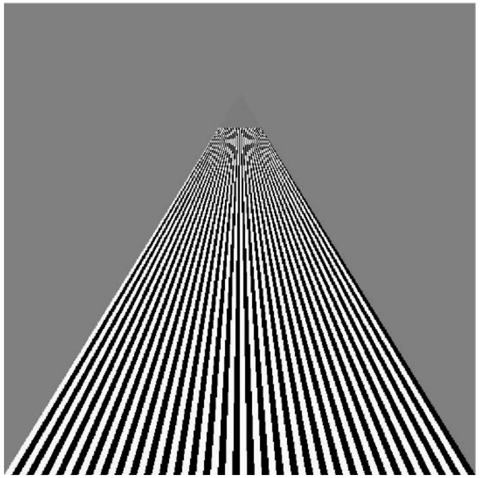
point sampling

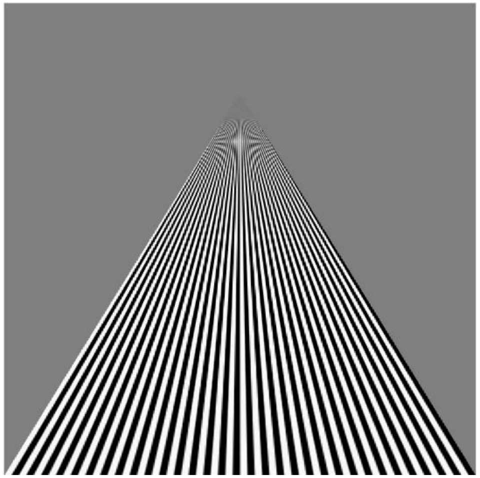




linear filtering

mipmapped point sampling





mipmapped linear filtering

Texture Functions

- Controls how texture is applied
  - **glTexEnv{fi}[v](GL_TEXTURE_ENV, prop, param)**
- **GL_TEXTURE_ENV_MODE** modes
  - **GL_MODULATE:** modulates with computed shade
  - **GL_BLEND:** blends with an environmental color
  - **GL_REPLACE:** use only texture color
  - **GL(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);**
- 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`**

Generating Texture Coordinates

- OpenGL can generate texture coordinates automatically

  glTexGen{ifd}[v]()

- specify a plane
  - generate texture coordinates based upon distance from the plane
- generation modes
  - **GL_OBJECT_LINEAR**
  - **GL_EYE_LINEAR**
  - **GL_SPHERE_MAP** (used for environmental maps)

Texture Objects

- Texture is part of the OpenGL state
  - If we have different textures for different objects, OpenGL will be moving large amounts data from processor memory to texture memory
- Recent versions of OpenGL have *texture objects*
  - one image per texture object
  - Texture memory can hold multiple texture objects

Applying Textures II

1. specify textures in texture objects
2. set texture filter
3. set texture function
4. set texture wrap mode
5. set optional perspective correction hint
6. bind texture object
7. enable texturing
8. supply texture coordinates for vertex
   - coordinates can also be generated

Other Texture Features

- Environment Maps
  - Start with image of environment through a wide angle lens
    - Can be either a real scanned image or an image created in OpenGL
  - Use this texture to generate a spherical map
  - Use automatic texture coordinate generation
- Multitexturing
  - Apply a sequence of textures through cascaded texture units

Compositing and Blending

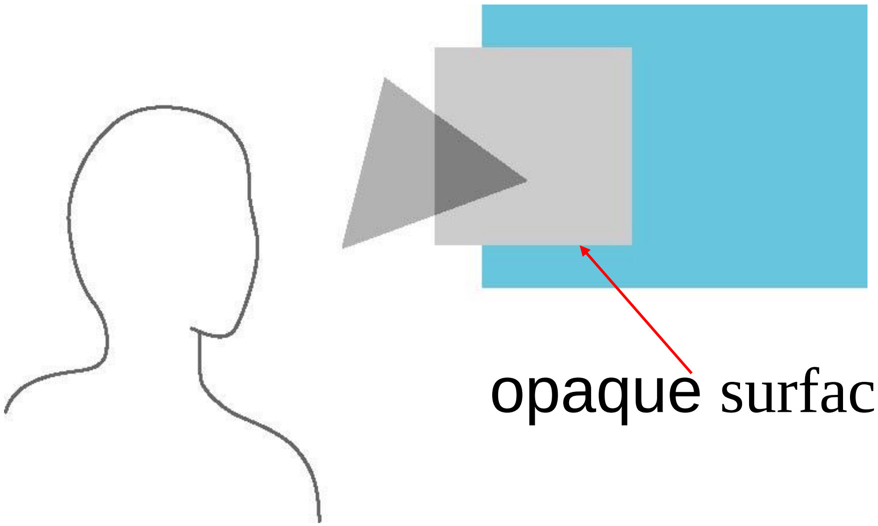
Objectives

- Learn to use the A component in RGBA color for
  - Blending for translucent surfaces
  - Compositing images
  - Antialiasing

Opacity and Transparency

- Opaque surfaces permit no light to pass through
- Transparent surfaces permit all light to pass
- Translucent surfaces pass some light

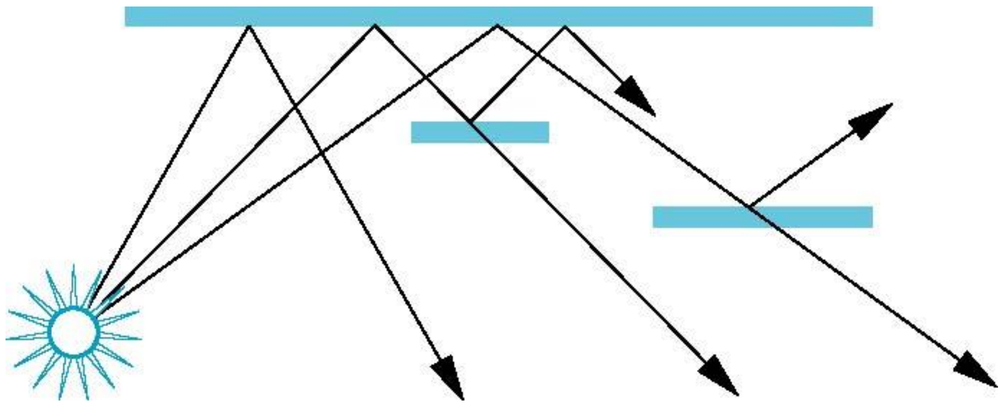
translucency = 1 – opacity (α)



opaque surface α =1

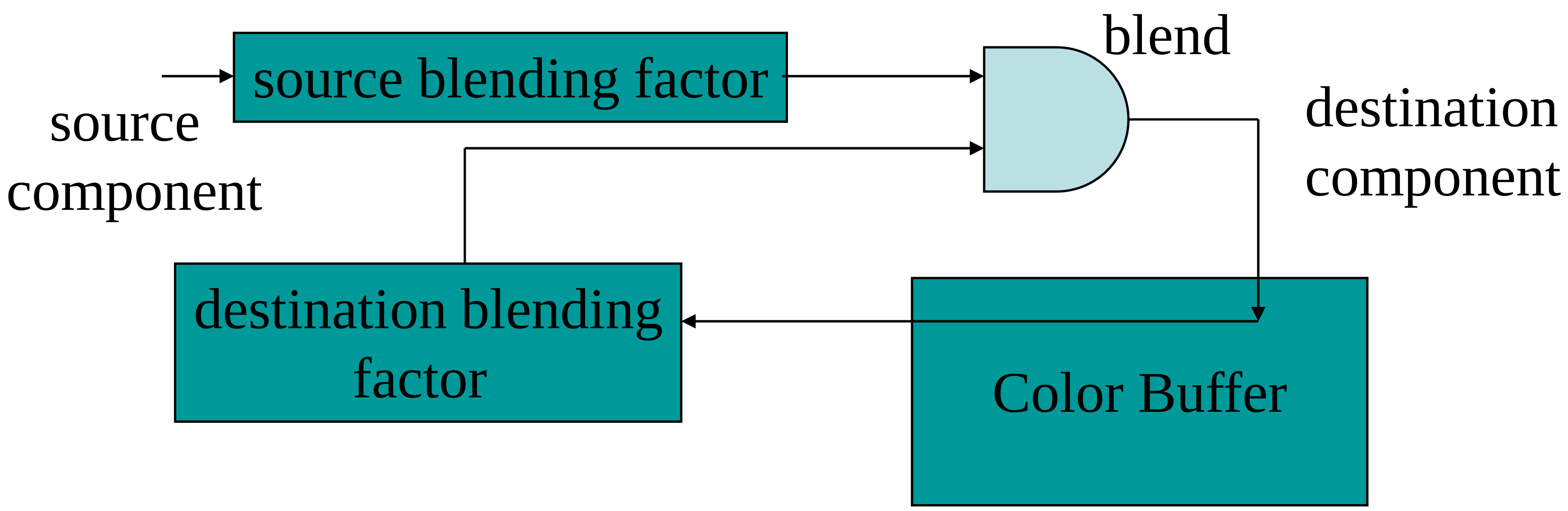
Physical Models

- Dealing with translucency in a physically correct manner is difficult due to
  - –the complexity of the internal interactions of light and matter
  - –Using a pipeline renderer



Writing Model

- Use A component of RGBA (or RGBα) color to store opacity
- During rendering we can expand our writing model to use RGBA values



Blending Equation

- We can define source and destination blending factors for each RGBA component

  $\mathbf{s} = [s_r, s_g, s_b, s_\alpha]$

  $\mathbf{d} = [d_r, d_g, d_b, d_\alpha]$

Suppose that the source and destination colors are

  $\mathbf{b} = [b_r, b_g, b_b, b_\alpha]$

  $\mathbf{c} = [c_r, c_g, c_b, c_\alpha]$

Blend as

$$\mathbf{c}' = [b_r\, s_r + c_r\, d_r,\ b_g\, s_g + c_g\, d_g,\ b_b\, s_b + c_b\, d_b,\ b_\alpha\, s_\alpha + c_\alpha\, d_\alpha]$$

OpenGL Blending and Compositing

- Must enable blending and pick source and destination factors

```
glEnable(GL_BLEND)
glBlendFunc(source_factor,
    destination_factor)
```

- Only certain factors supported
  - `GL_ZERO, GL_ONE`
  - `GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA`
  - `GL_DST_ALPHA, GL_ONE_MINUS_DST_ALPHA`
  - See Redbook for complete list

Example

- Suppose that we start with the opaque background color $(R_0,G_0,B_0,1)$
  - This color becomes the initial destination color
- We now want to blend in a translucent polygon with color (R_1,G_1,B_1,α_1)
- Select **`GL_SRC_ALPHA`** and **`GL_ONE_MINUS_SRC_ALPHA`** as the source and destination blending factors

$$R'_1 = \alpha_1 R_1 + (1- \alpha_1) R_0, \ldots\ldots$$

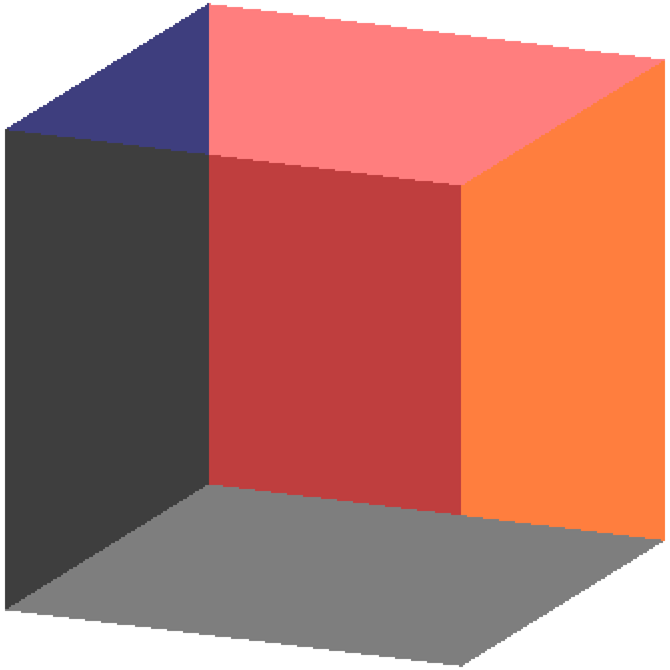
- Note this formula is correct if polygon is either opaque or transparent

Clamping and Accuracy

- All the components (RGBA) are clamped and stay in the range (0,1)
- However, in a typical system, RGBA values are only stored to 8 bits
  - Can easily loose accuracy if we add many components together
  - Example: add together n images
    - Divide all color components by n to avoid clamping
    - Blend with source factor = 1, destination factor = 1
    - But division by n loses bits

Order Dependency

- Is this image correct?
  - –Probably not
  - –Polygons are rendered in the order they pass down the pipeline
  - –Blending functions are order dependent



Opaque and Translucent Polygons

- Suppose that we have a group of polygons some of which are opaque and some translucent
- How do we use hidden-surface removal?
- Opaque polygons block all polygons behind them and affect the depth buffer
- Translucent polygons should not affect depth buffer
  - Render with **`glDepthMask(GL_FALSE)`** which makes depth buffer read-only
- Sort polygons first to remove order dependency

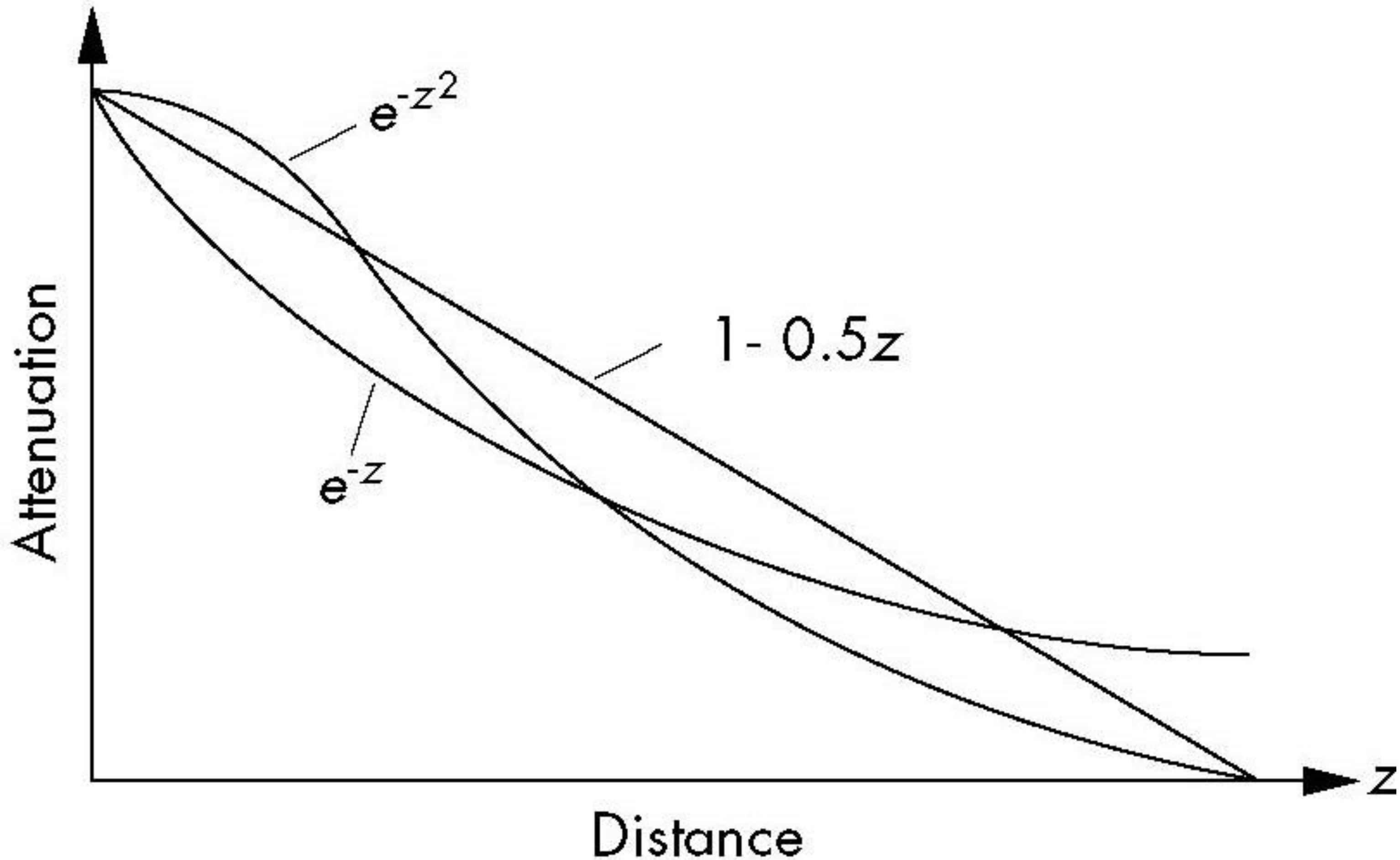
Fog

- We can composite with a fixed color and have the blending factors depend on depth
  - Simulates a fog effect
- Blend source color C_s and fog color C_f by

$$C_s' = f\,C_s + (1-f)\,C_f$$

- f is the *fog factor*
  - Exponential
  - Gaussian
  - Linear (depth cueing)

Fog Functions



OpenGL Fog Functions

```
GLfloat fcolor[4] = {……}:

glEnable(GL_FOG);
glFogf(GL_FOG_MODE, GL_EXP);
glFogf(GL_FOG_DENSITY, 0.5);
glFOgv(GL_FOG, fcolor);
```

Line Aliasing

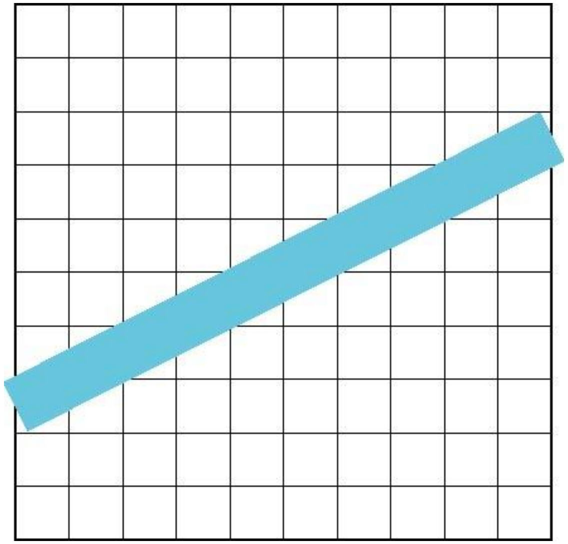
- Ideal raster line is one pixel wide
- All line segments, other than vertical and horizontal segments, partially cover pixels
- Simple algorithms color

only whole pixels

- Lead to the “jaggies”

or aliasing

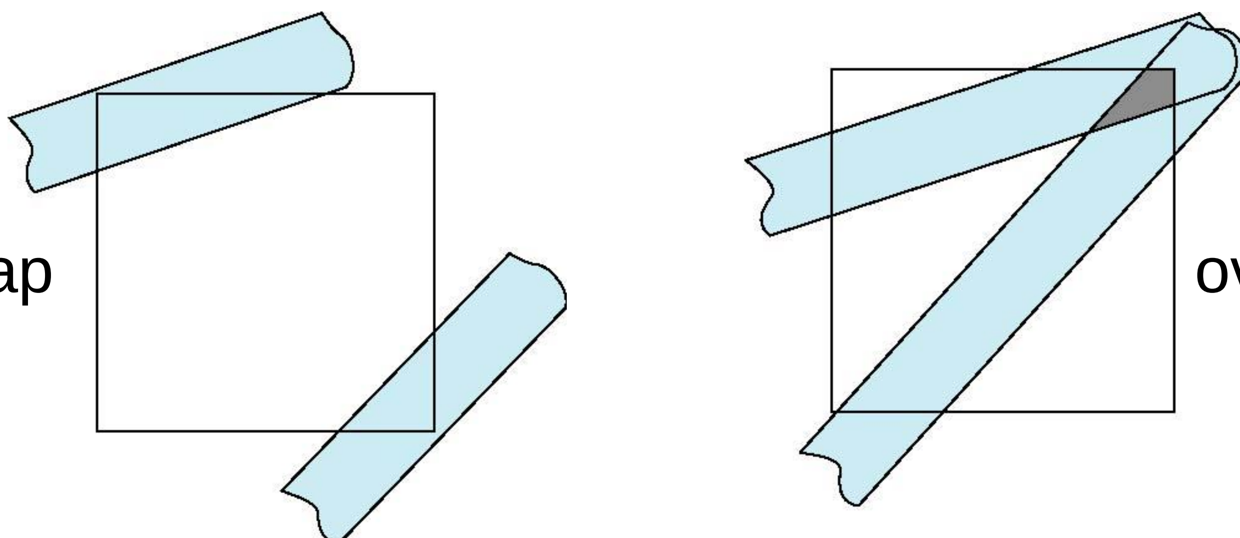
- Similar issue for polygons



Antialiasing

- Can try to color a pixel by adding a fraction of its color to the frame buffer
  - Fraction depends on percentage of pixel covered by fragment
  - Fraction depends on whether there is overlap

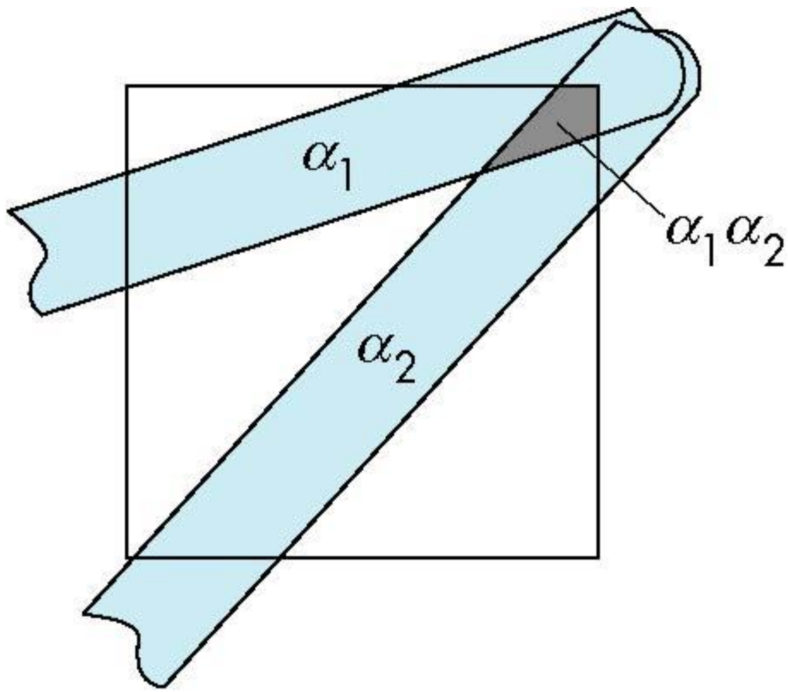
no overlap



overlap

Area Averaging

- Use average area $\alpha_1+\alpha_2-\alpha_1\alpha_2$ as blending factor



OpenGL Antialiasing

- Can enable separately for points, lines, or polygons

```
glEnable(GL_POINT_SMOOTH);
glEnable(GL_LINE_SMOOTH);
glEnable(GL_POLYGON_SMOOTH);

glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Accumulation Buffer

- Compositing and blending are limited by resolution of the frame buffer
  - Typically 8 bits per color component
- The *accumulation buffer* is a high resolution buffer (16 or more bits per component) that avoids this problem
- Write into it or read from it with a scale factor
- Slower than direct compositing into the frame buffer

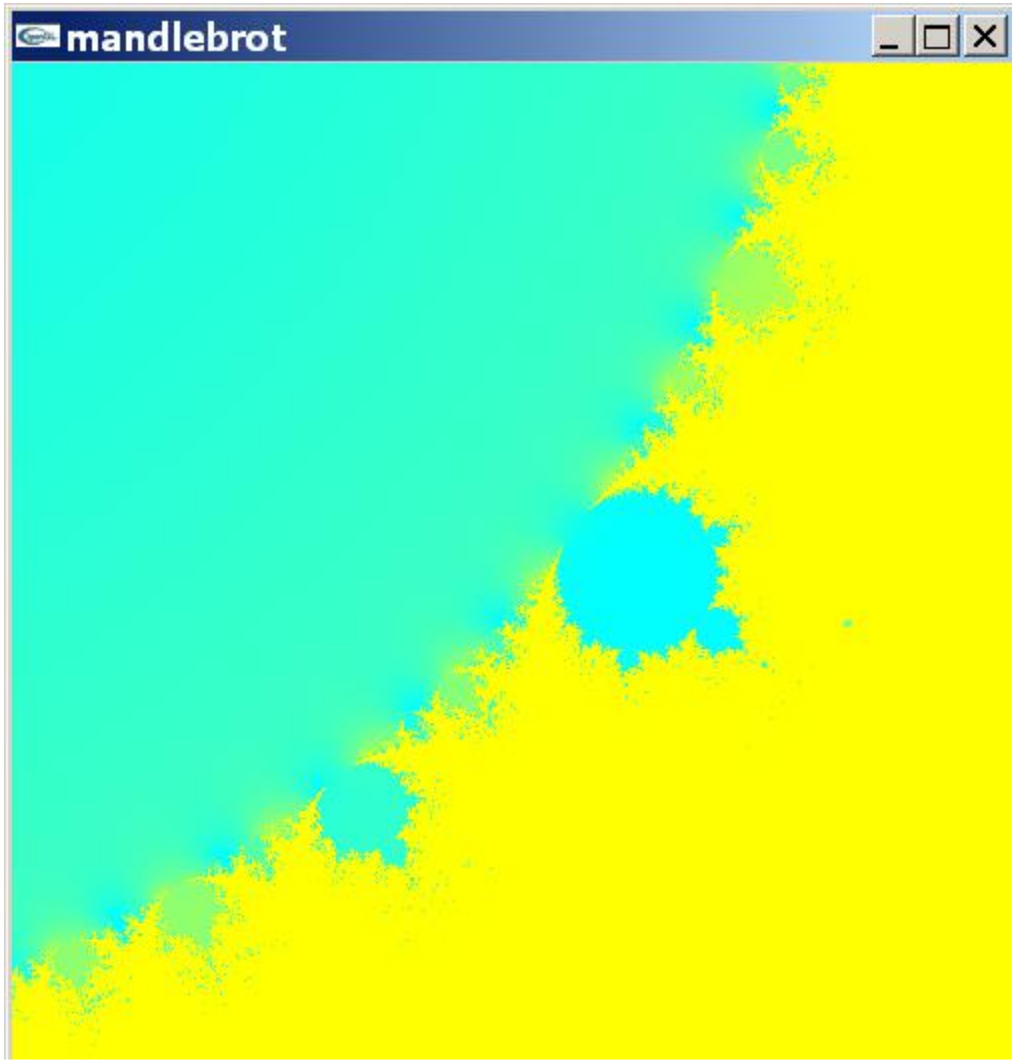
Applications

- Compositing
- Image Filtering (convolution)
- Whole scene antialiasing
- Motion effects

Sample Programs

- A.13 Mandelbrot set program
- A.14 Line drawing using Bresenham's algorithm
- A.15 A rotating cube with texture
- Rotating Utah teapot with wireframe
- Rotating Utah teapot with texture

A.13 mandelbrot.c



A.13 mandelbrot.c (1/8)

```
#include <stdio.h>
#include <stdlib.h>

#ifdef __APPLE__
#include <GLUT/glut.h>
#else
#include <GL/glut.h>
#endif

/* default data*/
/* can enter other values via command line arguments */
#define CENTERX -0.5
#define CENTERY 0.5
#define HEIGHT 0.5
#define WIDTH 0.5
#define MAX_ITER 100

/* N x M array to be generated */
#define N 500
#define M 500
```

```
float height = HEIGHT; /* size of window in complex plane */
float width = WIDTH;
float cx = CENTERX; /* center of window in complex plane */
float cy = CENTERY;
int max = MAX_ITER; /* number of interations per point */

int n=N;
int m=M;

/* use unsigned bytes for image */

GLubyte image[N][M];

/* complex data type and complex add, mult, and magnitude functions
   probably not worth overhead */

typedef float complex[2];
```

A.13 mandelbrot.c (3/8)

```
void add(complex a, complex b, complex p)
{
   p[0]=a[0]+b[0];
   p[1]=a[1]+b[1];
}

void mult(complex a, complex b, complex p)
{
   p[0]=a[0]*b[0]-a[1]*b[1];
   p[1]=a[0]*b[1]+a[1]*b[0];
}

float mag2(complex a)
{
   return(a[0]*a[0]+a[1]*a[1]);
}

void form(float a, float b, complex p)
{
   p[0]=a;
   p[1]=b;
}
```

A.13 mandelbrot.c (4/8)

```
void display()
{
   glClear(GL_COLOR_BUFFER_BIT);
   glDrawPixels(n,m,GL_COLOR_INDEX, GL_UNSIGNED_BYTE, image);
   glFlush();
}


void myReshape(int w, int h)
{
   glViewport(0, 0, w, h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
      gluOrtho2D(0.0, 0.0, (GLfloat) n, (GLfloat) m* (GLfloat) h / (GLfloat) w);
   else
     gluOrtho2D(0.0, 0.0, (GLfloat) n * (GLfloat) w / (GLfloat) h, (GLfloat) m);
   glMatrixMode(GL_MODELVIEW);
}
```

```
void myinit()
{
   float redmap[256], greenmap[256],bluemap[256];
   int i;

   glClearColor (1.0, 1.0, 1.0, 1.0);
   gluOrtho2D(0.0, 0.0, (GLfloat) n, (GLfloat) m);

   /* define pseudocolor maps, ramps for red and blue,
  random for green */

   for(i=0;i<256;i++)
   {
      redmap[i]=i/255.;
      greenmap[i]=rand()%255;
      bluemap[i]=1.0-i/255.;
   }
   glPixelMapfv(GL_PIXEL_MAP_I_TO_R, 256, redmap);
   glPixelMapfv(GL_PIXEL_MAP_I_TO_G, 256, greenmap);
   glPixelMapfv(GL_PIXEL_MAP_I_TO_B, 256, bluemap);
}
```

A.13 mandelbrot.c (6/8)

```
main(int argc, char *argv[])
{
    int i, j, k;
    float x, y, v;
    complex c0, c, d;

/* uncomment to define your own parameters */
/*    scanf("%f", &cx); /* center x */
/*    scanf("%f", &cy); /* center y */
 /*   scanf("%f", &width); /* rectangle width */
 /*   height=width; /* rectangle height */
/*    scanf("%d",&max); /* maximum iterations */

    for (i=0; i<n; i++) for(j=0; j<m; j++)
    {
      /* starting point */
      x= i *(width/(n-1)) + cx -width/2;
      y= j *(height/(m-1)) + cy -height/2;

      form(0,0,c);
      form(x,y,c0);
```

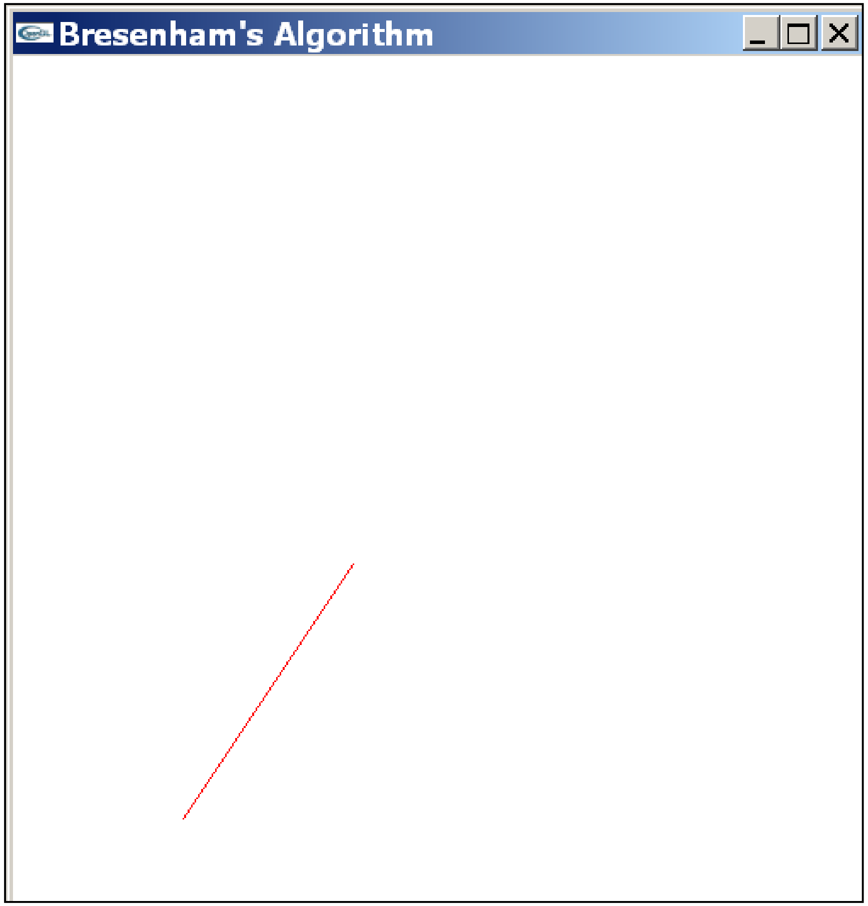
A.13 mandelbrot.c (7/8)

```
/* complex iteration */

    for(k=0; k<max; k++)
        {
        mult(c,c,d);
        add(d,c0,c);
        v=mag2(c);
        if(v>4.0) break; /* assume not in set if mag > 4 */
        }

/* assign gray level to point based on its magnitude */
        if(v>1.0) v=1.0; /* clamp if > 1 */
        image[i][j]=255*v;
    }
```

A.13 mandelbrot.c (8/8)

```
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB );
glutInitWindowSize(N, M);
glutCreateWindow("mandlebrot");
myinit();
glutReshapeFunc(myReshape);
glutDisplayFunc(display);

glutMainLoop();


}
```

A.14 Bresenham's algorithm



A.14 bres.c (1/6)

```
#define BLACK 0

#ifdef __APPLE__
#include <GLUT/glut.h>
#else
#include <GL/glut.h>
#endif

#include <stdio.h>


void draw_pixel(int ix, int iy, int value)
{
  glBegin(GL_POINTS);
    glVertex2i( ix, iy);
  glEnd();
}
```

A.14 bres.c (2/6)

```
bres(int x1,int y1,int x2,int y2)
{
  int dx, dy, i, e;
  int incx, incy, inc1, inc2;
  int x,y;

  dx = x2 - x1;
  dy = y2 - y1;

  if(dx < 0) dx = -dx;
  if(dy < 0) dy = -dy;
  incx = 1;
  if(x2 < x1) incx = -1;
  incy = 1;
  if(y2 < y1) incy = -1;
  x=x1;
  y=y1;
```

A.14 bres.c (3/6)

```
If (dx > dy)
  {
    draw_pixel(x,y, BLACK);
    e = 2*dy - dx;
    inc1 = 2*( dy -dx);
    inc2 = 2*dy;
    for (i = 0; i < dx; i++)
    {
      if (e >= 0)
      {
        y += incy;
        e += inc1;
      }
      else e += inc2;
      x += incx;
      draw_pixel(x,y, BLACK);
    }
  }
```

A.14 bres.c (4/6)

```
else  /* dx <=dy */
 {
   draw_pixel(x,y, BLACK);
   e = 2*dx - dy;
   inc1 = 2*( dx - dy);
   inc2 = 2*dx;
   for (i = 0; i < dy; i++)
   {
     if (e >= 0)
     {
        x += incx;
        e += inc1;
     }
     else e += inc2;
     y += incy;
     draw_pixel(x,y, BLACK);
  }
 }
}
```

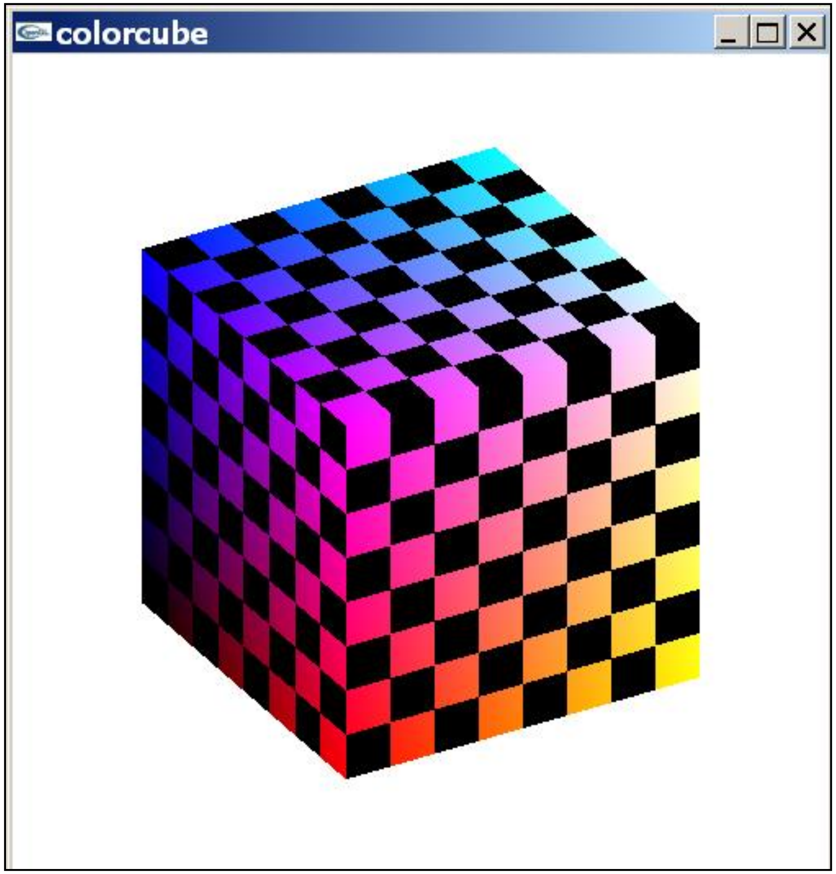
A.14 bres.c (5/6)

```
void display(void)
{
        glClear(GL_COLOR_BUFFER_BIT);
        bres(200, 200, 100, 50);
        glFlush();
}

void myinit()
{
        glClearColor(1.0, 1.0, 1.0, 1.0);
        glColor3f(1.0, 0.0, 0.0);
        glPointSize(1.0);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluOrtho2D(0.0, 499.0, 0.0, 499.0);
}
```

A.14 bres.c (6/6)

```
void main(int argc, char** argv)
{

/* Standard GLUT initialization */

    glutInit(&argc,argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
            /* default, not needed */
    glutInitWindowSize(500,500); /* 500 x 500 pixel window */
    glutInitWindowPosition(0,0); /* place window top left on display */
    glutCreateWindow("Bresenham's Algorithm"); /* window title */
    glutDisplayFunc(display);
           /* display callback invoked when window opened */

    myinit(); /* set attributes */

    glutMainLoop(); /* enter event loop */
}
```

A.15 Rotating cube with texture



A.15 tex_cube.c (1/7)

```
#include <stdlib.h>

#ifdef __APPLE__
#include <GLUT/glut.h>
#else
#include <GL/glut.h>
#endif

GLfloat planes[]= {-1.0, 0.0, 1.0, 0.0};
GLfloat planet[]= {0.0, -1.0,  0.0, 1.0};
GLfloat vertices[][3] = {{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},
   {1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-1.0,-1.0,1.0},
   {1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}};
GLfloat colors[][4] = {{0.0,0.0,0.0,0.5},{1.0,0.0,0.0,0.5},
   {1.0,1.0,0.0,0.5}, {0.0,1.0,0.0,0.5}, {0.0,0.0,1.0,0.5},
   {1.0,0.0,1.0,0.5}, {1.0,1.0,1.0,0.5}, {0.0,1.0,1.0,0.5}};
```

A.15 tex_cube.c (2/7)

```
void polygon(int a, int b, int c, int d)
{
   glBegin(GL_POLYGON);
   glColor4fv(colors[a]);
   glTexCoord2f(0.0,0.0);
   glVertex3fv(vertices[a]);
   glColor4fv(colors[b]);
   glTexCoord2f(0.0,1.0);
   glVertex3fv(vertices[b]);
   glColor4fv(colors[c]);
   glTexCoord2f(1.0,1.0);
   glVertex3fv(vertices[c]);
   glColor4fv(colors[d]);
   glTexCoord2f(1.0,0.0);
   glVertex3fv(vertices[d]);
   glEnd();
}
```

A.15 tex_cube.c (3/7)

```
void colorcube()
{
/* map vertices to faces */
    polygon(0,3,2,1);
    polygon(2,3,7,6);
    polygon(0,4,7,3);
    polygon(1,2,6,5);
    polygon(4,5,6,7);
    polygon(0,1,5,4);
}
static GLfloat theta[] = {0.0,0.0,0.0};
static GLint axis = 2;
void display()
{   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glRotatef(theta[0], 1.0, 0.0, 0.0);
    glRotatef(theta[1], 0.0, 1.0, 0.0);
    glRotatef(theta[2], 0.0, 0.0, 1.0);
    colorcube();
    glutSwapBuffers();
}
```

A.15 tex_cube.c (4/7)

```
void spinCube()
{
   theta[axis] += 2.0;
   if ( theta[axis] > 360.0 ) theta[axis] -= 360.0;
   glutPostRedisplay();
}

void mouse(int btn, int state, int x, int y)
{
   if (btn==GLUT_LEFT_BUTTON && state == GLUT_DOWN) axis = 0;
   if (btn==GLUT_MIDDLE_BUTTON && state == GLUT_DOWN) axis = 1;
   if (btn==GLUT_RIGHT_BUTTON && state == GLUT_DOWN) axis = 2;
}
```

A.15 tex_cube.c (5/7)

```
void myReshape(int w, int h)
{
   glViewport(0, 0, w, h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
      glOrtho(-2.0, 2.0, -2.0 * (GLfloat) h / (GLfloat) w,
         2.0 * (GLfloat) h / (GLfloat) w, -10.0, 10.0);
   else
      glOrtho(-2.0 * (GLfloat) w / (GLfloat) h,
         2.0 * (GLfloat) w / (GLfloat) h, -2.0, 2.0, -10.0, 10.0);
   glMatrixMode(GL_MODELVIEW);
}

void key(unsigned char k, int x, int y)
{
   if (k == '1') glutIdleFunc(spinCube);
   if (k == '2') glutIdleFunc(NULL);
   if (k == 'q') exit(0);
}
```

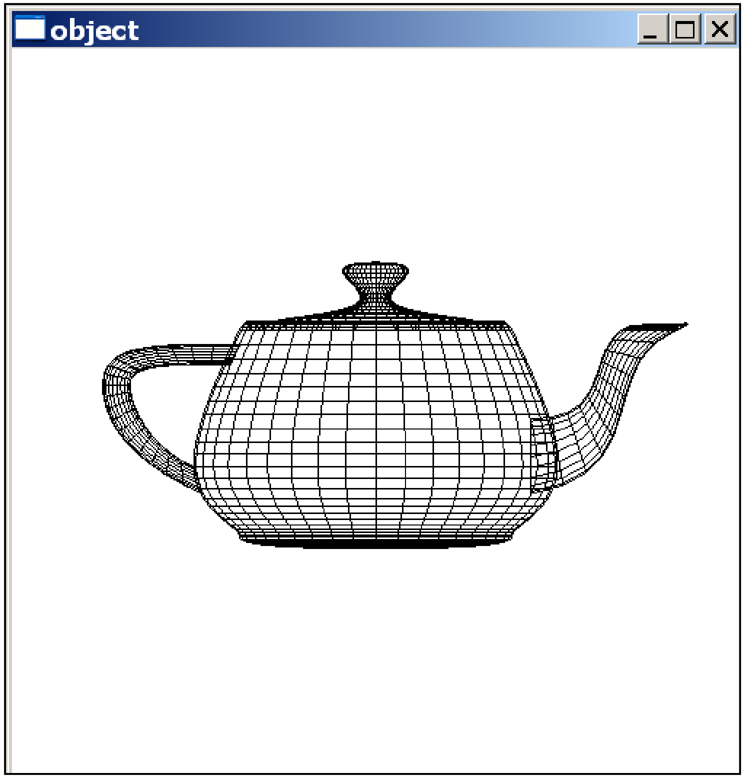
A.15 tex_cube.c (6/7)

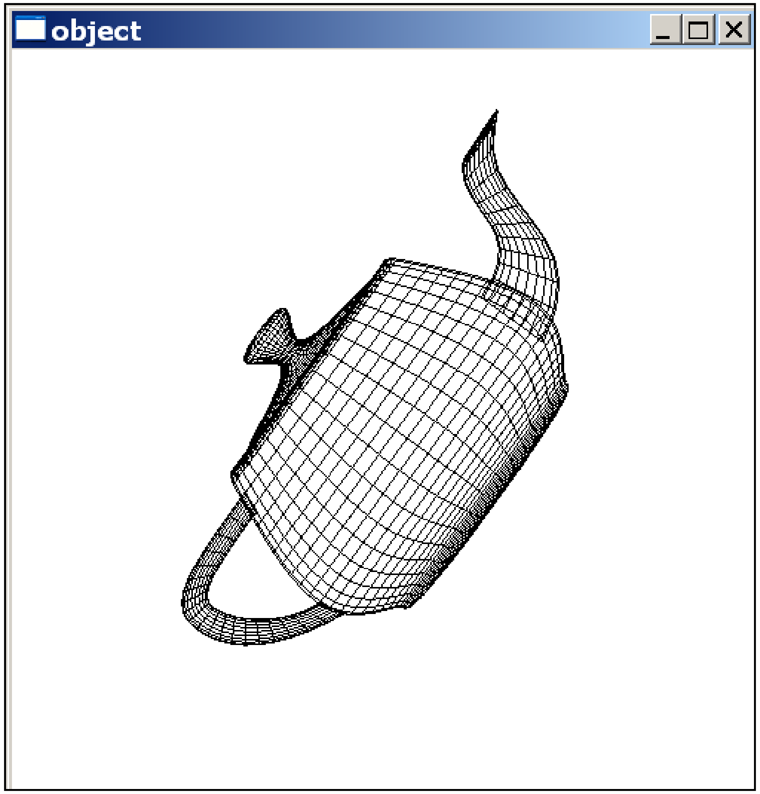
```
int main(int argc, char **argv)
{
   GLubyte image[64][64][3];
   int i, j,  c;
   for (i=0;i<64;i++)
   {
     for (j=0;j<64;j++)
     {
       c = ((((i&0x8)==0)^((j&0x8))==0))*255;
       image[i][j][0]= (GLubyte) c;
       image[i][j][1]= (GLubyte) c;
       image[i][j][2]= (GLubyte) c;
     }
   }
```

A.15 tex_cube.c (7/7)

```
 glutInit(&argc, argv);
 glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
 glutInitWindowSize(500, 500);
 glutCreateWindow("colorcube");
 glutReshapeFunc(myReshape);
 glutDisplayFunc(display);
 glutIdleFunc(spinCube);
 glutMouseFunc(mouse);
 glEnable(GL_DEPTH_TEST);
 glEnable(GL_TEXTURE_2D);
 glTexImage2D(GL_TEXTURE_2D, 0, 3, 64, 64, 0, GL_RGB, GL_UNSIGNED_BYTE,
image);
 glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_REPEAT);
 glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_REPEAT);
 glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);
 glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);
 glutKeyboardFunc(key);
 glClearColor(1.0,1.0,1.0,1.0);

 glutMainLoop();
}
```

Rotating Utah teapot with wireframe (object.c)





object.c (1/5)

```c
/* displays various glu objects */

#include <stdlib.h>
#include <GL/glut.h>

GLUquadricObj *obj;
static GLfloat theta[] = {0.0,0.0,0.0};
static GLint axis = 2;

void display()
{
/* display callback, clear frame buffer and z buffer,
   rotate object and draw, swap buffers */

 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
 glLoadIdentity();
 glRotatef(theta[0], 1.0, 0.0, 0.0);
 glRotatef(theta[1], 0.0, 1.0, 0.0);
 glRotatef(theta[2], 0.0, 0.0, 1.0);
```

object.c (2/5)

```
/*          glutWireIcosahedron();*/
/*          glutWireDodecahedron();*/
/*          gluSphere(obj, 1.0, 12, 12);*/
/*          gluCylinder(obj, 1.0, 0.5, 1.0, 12, 12); */
/*          gluDisk(obj, 0.5, 1.0, 10, 10);*/
/*          gluPartialDisk( obj, 0.5, 1.0, 10, 10, 0.0, 45.0);*/
            glutWireTeapot(1.0);
/*          glutWireTorus(0.5, 1.0, 10, 10);*/
/*          glutWireCone(1.0, 1.0, 10, 10);*/

            glutSwapBuffers();
}
```

object.c (3/5)

```
void spinObject()
{

/* Idle callback, spin cube 2 degrees about selected axis */

        theta[axis] += 2.0;
        if ( theta[axis] > 360.0 ) theta[axis] -= 360.0;
        glutPostRedisplay();
}

void mouse(int btn, int state, int x, int y)
{

/* mouse callback, selects an axis about which to rotate */

 if (btn==GLUT_LEFT_BUTTON && state == GLUT_DOWN) axis = 0;
 if (btn==GLUT_MIDDLE_BUTTON && state == GLUT_DOWN) axis = 1;
 if (btn==GLUT_RIGHT_BUTTON && state == GLUT_DOWN) axis = 2;
}
```

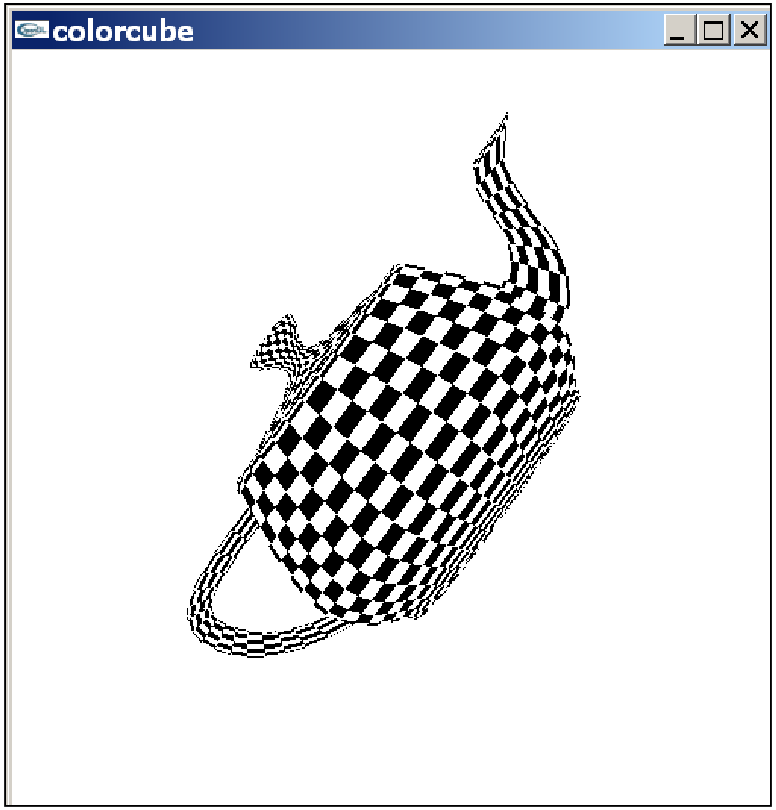
object.c (4/5)

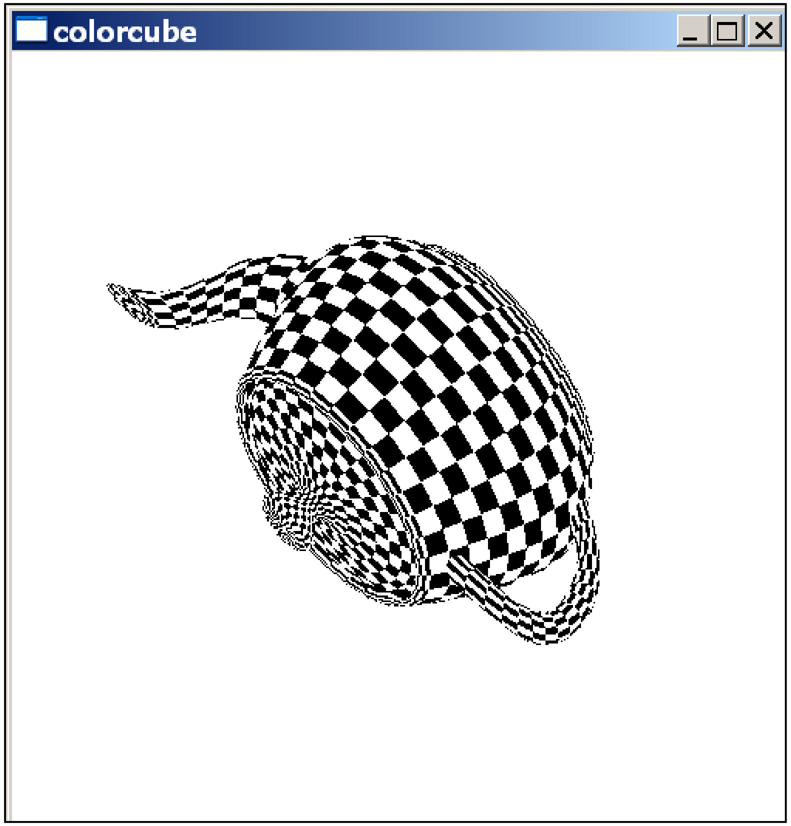
```
void myReshape(int w, int h)
{
   glViewport(0, 0, w, h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
      glOrtho(-2.0, 2.0, -2.0 * (GLfloat) h / (GLfloat) w,
         2.0 * (GLfloat) h / (GLfloat) w, -10.0, 10.0);
   else
      glOrtho(-2.0 * (GLfloat) w / (GLfloat) h,
         2.0 * (GLfloat) w / (GLfloat) h, -2.0, 2.0, -10.0, 10.0);
   glMatrixMode(GL_MODELVIEW);
}

void key(unsigned char key, int x, int y)
{
      if (key=='1') glutIdleFunc(NULL);
      if (key=='2') glutIdleFunc(spinObject);
}
```

object.c (5/5)

```
main(int argc, char **argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
   glutInitWindowSize(500, 500);
   glutCreateWindow("object");

  /* need both double buffering and z buffer */
   glutReshapeFunc(myReshape);
   glutDisplayFunc(display);
   glutIdleFunc(NULL);
   glutMouseFunc(mouse);
   glutKeyboardFunc(key);
   glClearColor(1.0, 1.0, 1.0, 1.0);
   glColor3f(0.0, 0.0, 0.0);
   obj = gluNewQuadric();
   gluQuadricDrawStyle(obj, GLU_LINE);

   glutMainLoop();
}
```

Rotating Utah teapot with texture (teatex.c)





teatex.c (1/6)

```
/* Utah teapot with texture map  */

#include <stdlib.h>
#include <GL/glut.h>

GLfloat planes[]= {-1.0, 0.0, 1.0, 0.0};
GLfloat planet[]= {0.0, -1.0,  0.0, 1.0};

GLfloat vertices[][3] = {{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},
        {1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-1.0,-1.0,1.0},
        {1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}};

GLfloat colors[][4] = {{0.0,0.0,0.0,0.5},{1.0,0.0,0.0,0.5},
        {1.0,1.0,0.0,0.5}, {0.0,1.0,0.0,0.5}, {0.0,0.0,1.0,0.5},
        {1.0,0.0,1.0,0.5}, {1.0,1.0,1.0,0.5}, {0.0,1.0,1.0,0.5}};

static GLfloat theta[] = {0.0,0.0,0.0};
static GLint axis = 2;
```

teatex.c (2/6)

```
void display(void)
{
/* display callback, clear frame buffer and z buffer,
   rotate cube and draw, swap buffers */

 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
 glLoadIdentity();
 glRotatef(theta[0], 1.0, 0.0, 0.0);
 glRotatef(theta[1], 0.0, 1.0, 0.0);
 glRotatef(theta[2], 0.0, 0.0, 1.0);

 glutSolidTeapot(1.0);


 glutSwapBuffers();
}
```

teatex.c (3/6)

```
void spinTeapot()
{

/* Idle callback, spin cube 2 degrees about selected axis */

            theta[axis] += 2.0;
            if ( theta[axis] > 360.0 ) theta[axis] -= 360.0;
            glutPostRedisplay();
}

void mouse(int btn, int state, int x, int y)
{

/* mouse callback, selects an axis about which to rotate */

 if (btn==GLUT_LEFT_BUTTON && state == GLUT_DOWN) axis = 0;
 if (btn==GLUT_MIDDLE_BUTTON && state == GLUT_DOWN) axis = 1;
 if (btn==GLUT_RIGHT_BUTTON && state == GLUT_DOWN) axis = 2;
}
```

teatex.c (4/6)

```
void myReshape(int w, int h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        glOrtho(-2.0, 2.0, -2.0 * (GLfloat) h / (GLfloat) w,
            2.0 * (GLfloat) h / (GLfloat) w, -10.0, 10.0);
    else
        glOrtho(-2.0 * (GLfloat) w / (GLfloat) h,
            2.0 * (GLfloat) w / (GLfloat) h, -2.0, 2.0, -10.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
}

void key(char k, int x, int y)
{
   if (k == '1') glutIdleFunc(spinTeapot);
   if (k == '2') glutIdleFunc(NULL);
}
```

teatex.c (5/6)

```
void
main(int argc, char **argv)
{
   GLubyte image[64][64][3];
   int i, j, r, c;
   for (i=0;i<64;i++)
   {
     for (j=0;j<64;j++)
     {
       c = ((((i&0x8)==0)^((j&0x8))==0))*255;
       image[i][j][0]= (GLubyte) c;
       image[i][j][1]= (GLubyte) c;
       image[i][j][2]= (GLubyte) c;
     }
   }
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(500, 500);
    glutCreateWindow("colorcube");
```

teatex.c (6/6)

```
/* need both double buffering and z buffer */

glutReshapeFunc(myReshape);
glutDisplayFunc(display);
glutIdleFunc(spinTeapot);
glutMouseFunc(mouse);
glEnable(GL_DEPTH_TEST);
glEnable(GL_TEXTURE_2D);
glTexImage2D(GL_TEXTURE_2D,0,3,64,64,0,GL_RGB,GL_UNSIGNED_BYTE, image);
glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_REPEAT);
glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_REPEAT);
glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);
glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);
glutKeyboardFunc(key);
glClearColor(1.0,1.0,1.0,1.0);
glutMainLoop();
}
```