

Development of Games

Lecture 10

Introduction to OpenGL

Based on lecture of Dave Shreiner, Ed Angel,
Vicki Shreiner

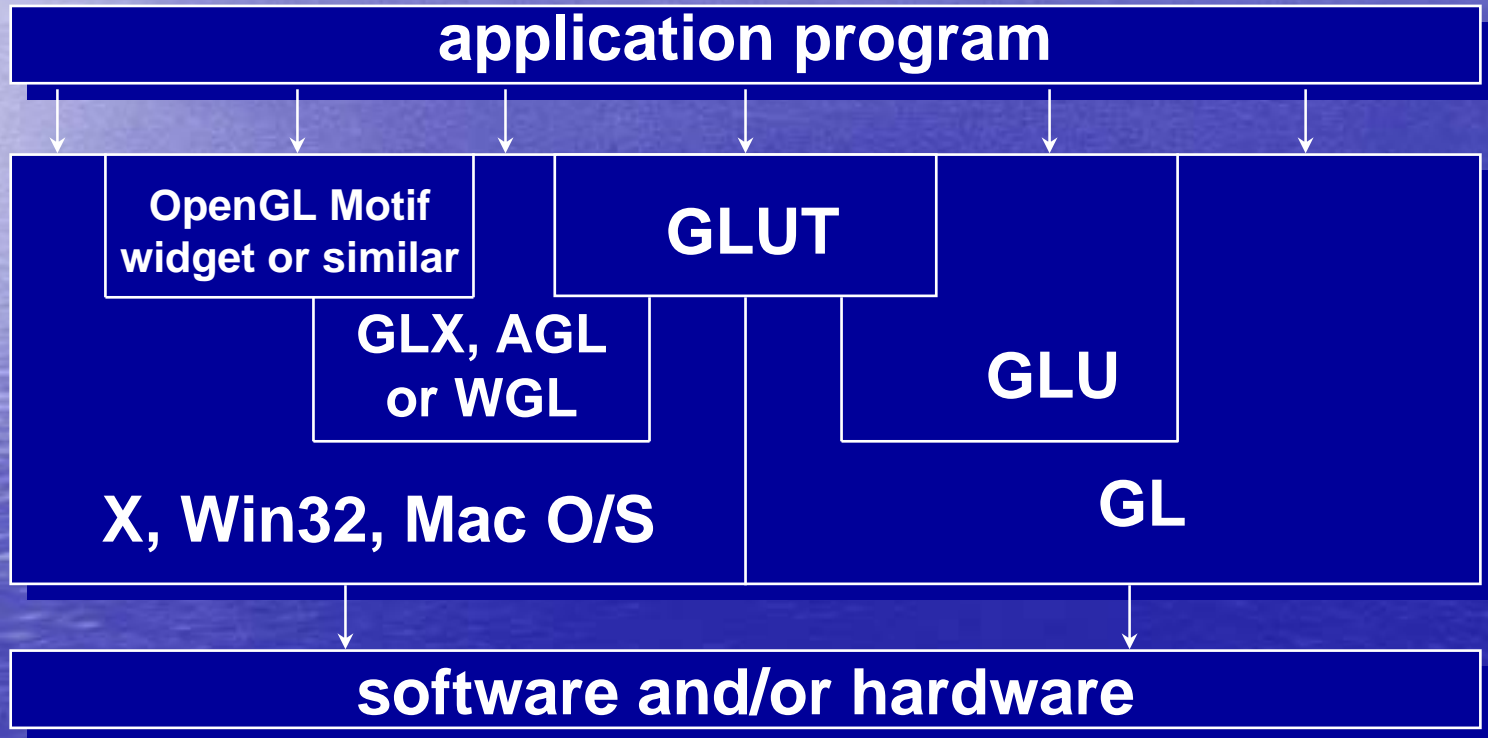
OpenGL as a Renderer

- Geometric primitives
 - points, lines and polygons
- Image Primitives
 - images and bitmaps
 - separate pipeline for images and geometry
 - linked through texture mapping
- Rendering depends on state
 - colors, materials, light sources, etc.

Related APIs

- AGL, GLX, WGL
 - glue between OpenGL and windowing systems
- GLU (OpenGL Utility Library)
 - part of OpenGL
 - NURBS, tessellators, quadric shapes, etc.
- GLUT (OpenGL Utility Toolkit)
 - portable windowing API
 - not officially part of OpenGL

OpenGL and Related APIs



Preliminaries

- Headers Files
 - `#include <GL/gl.h>`
 - `#include <GL/glu.h>`
 - `#include <GL/glut.h>`
- Libraries
- Enumerated Types
 - OpenGL defines numerous types for compatibility
 - `GLfloat`, `GLint`, `GLenum`, etc.

GLUT Basics

- Application Structure
 - Configure and open window
 - Initialize OpenGL state
 - Register input callback functions
 - render
 - resize
 - input: keyboard, mouse, etc.
 - Enter event processing loop

Sample Program

```
void main( int argc, char** argv )
{
    int mode = GLUT_RGB|GLUT_DOUBLE;
    glutInitDisplayMode( mode );
    glutCreateWindow( argv[0] );
    init();

    glutDisplayFunc( display );
    glutReshapeFunc( resize );
    glutKeyboardFunc( key );
    glutIdleFunc( idle );
    glutMainLoop();
}
```

OpenGL Initialization

- Set up whatever state you're going to use

```
void init( void )
{
    glClearColor( 0.0, 0.0, 0.0, 1.0
);
    glClearDepth( 1.0 );

    glEnable( GL_LIGHT0 );
    glEnable( GL_LIGHTING );
    glEnable( GL_DEPTH_TEST );
}
```


GLUT Callback Functions

- Routine to call when something happens
 - window resize or redraw
 - user input
 - animation

- “Register” callbacks with GLUT

```
glutDisplayFunc( display );  
glutIdleFunc( idle );  
glutKeyboardFunc( keyboard );
```

Rendering Callback

- Do all of your drawing here

```
glutDisplayFunc( display );
```

```
void display( void )  
{  
    glClear( GL_COLOR_BUFFER_BIT );  
    glBegin( GL_TRIANGLE_STRIP );  
        glVertex3fv( v[0] );  
        glVertex3fv( v[1] );  
        glVertex3fv( v[2] );  
        glVertex3fv( v[3] );  
    glEnd();  
    glutSwapBuffers();  
}
```

Idle Callbacks

- Use for animation and continuous update

```
glutIdleFunc( idle );
```

```
void idle( void )  
{  
    t += dt;  
    glutPostRedisplay();  
}
```

User Input Callbacks

- Process user input

```
    glutKeyboardFunc( keyboard );
```

```
void keyboard( char key, int x, int y )
{
    switch( key ) {
        case 'q' : case 'Q' :
            exit( EXIT_SUCCESS );
            break;

        case 'r' : case 'R' :
            rotate = GL_TRUE;
            break;
    }
}
```

GL functions

Function Name	Usage	Purpose	General Notes
<code>glBegin</code>	<code>glBegin(type_of_shape)</code>	Starts drawing a shape.	The different types of shapes that can be drawn are documented in the "Drawing Paradigm" section.
<code>glColor</code>	<code>glColor(R, G, B[, A])</code>	Sets the color of the shape to be drawn.	The first three (required) arguments specify the color. The last argument specifies the opacity and is optional. If omitted, it defaults to 1 (opaque).
<code>glClear</code>	<code>glClear(bitfield)</code>	This clears a specified buffer.	If <code>GL_COLOR_BUFFER_BIT</code> is passed in, all the buffers being drawn on in color are reset to the color specified by <code>glClearColor</code>
<code>glClearColor</code>	<code>glClearColor(R,B,G,A)</code>	Sets the color to be used when a buffer is cleared.	The arguments work just as in <code>glColor</code> except the last argument is required.
<code>glEnd</code>	<code>glEnd()</code>	Sets up the program for using OpenGL.	You pass in <code>argc</code> and <code>argv</code> because you can change certain OpenGL settings at the command line.
<code>glFinish</code>	<code>glFinish()</code>	Lets OpenGL catch up with you.	This function returns when OpenGL is finished executing whatever jobs it has on its queue.
<code>glFlush</code>	<code>glFlush()</code>	Forces OpenGL to finish everything it's doing in a finite amount of time.	This works like <code>fflush</code> in that it ensures that all previous OpenGL commands will be executed. Unlike <code>glFinish</code> this is non-blocking.
<code>glLoadIdentity</code>	<code>glLoadIdentity()</code>	Effectively clears out the matrix to which you are drawing.	It really loads an identity matrix.
<code>glMatrixMode</code>	<code>glMatrixMode(bitfield)</code>	Sets which type of matrix we are editing.	You will probably just want to stick with <code>GL_PROJECTION</code> , used for editing in window coordinates, though <code>GL_MODELVIEW</code> might also be useful.
<code>glPopMatrix</code>	<code>glPopMatrix()</code>	Saves the current matrix by pushing it onto a LIFO queue.	You set which matrix will be duplicated with <code>glMatrixMode</code> .
<code>glPointSize</code>	<code>glPointSize(size)</code>	Sets the size of drawn points.	Size is the size in pixels.
<code>glPushMatrix</code>	<code>glPushMatrix()</code>	Retrieves the next matrix on a LIFO queue, discarding the top one.	You set which matrix will be duplicated with <code>glMatrixMode</code> .

GL functions (cont.)

<code>glRasterPos</code>	<code>glRasterPos(x, y, z)</code>	Specify the x, y, and z coordinates of the raster.	You will primarily use this to position text on the screen. This would be used in conjunction with <code>glutBitmapCharacter</code> .
<code>glRect</code>	<code>glRect(x1, y1, x2, y2)</code>	Draws a rectangle.	Remember to specify the type of variable being passed in!!
<code>glRotate</code>	<code>glRotate(angle, x, y, z)</code>	Rotates a matrix.	angle is the angle through which to rotate. x, y, and z specify the vector about which to rotate the matrix.
<code>glScale</code>	<code>glScale(x, y, z)</code>	Scale.	There are different parameters for the x, y, and z, axes so you can scale them independently.
<code>glShadeModel</code>	<code>glShadeModel(shading_mode)</code>	Specifies the shading mode.	Just use <code>GL_SMOOTH</code> as the argument.
<code>glTranslate</code>	<code>glTranslate(x, y, z)</code>	Translate a matrix.	x, y, and z specify the vector along which to translate the matrix.
<code>glVertex</code>	<code>glVertex(x, y[, z, w])</code>	Sets a vertex to be used by <code>glBegin</code> for drawing.	The coordinates this function uses are set by <code>glOrtho</code> .
<code>glViewport</code>	<code>glViewport(x, y, width, height)</code>	Sets up where the viewport is in the OpenGL window.	x and y are the coordinates of the lower-left corner of the viewport in the window. width and height are the width and height of the viewport. <i>This should be called whenever the window is resized!!</i>

GLU functions

<code>gluOrtho</code>	<code>gluOrtho(left, right, bottom, top)</code>	Set up a coordinate system for you to use in the window.	This function technically gets you an "orthographic projection" of the world from the camera's perspective. If you are doing stuff in 2D, you do not need to know what this means, and can just think of this as setting up your coordinate system in world-coordinates.
-----------------------	---	--	--

GLUT functions

glutBitmapCharacter	glutBitmapCharacter(font, character)	Write a character to the screen.	See documentation for a full list of fonts. A few useful ones are GLUT_BITMAP_TIMES_ROMAN_10 and GLUT_BITMAP_HELVETICA_10
glutCreateWindow	glutCreateWindow(name)	Creates a window.	The parameter is the name of the window.
glutDestroyWindow	glutDestroyWindow(window)	Closes a window.	The parameter indicates which window to close. Every window is associated with a number, and you can find out what this is with .
glutDisplayFunc	glutDisplayFunc(display)	Registers the function to be used if the window is to be redisplayed.	-
glutGet	glutGet(what_to_get)	Gets variables local to OpenGL, such as the window size or the number of red pixels.	Calling this with GLUT_WINDOW_HEIGHT and GLUT_WINDOW_WIDTH could be useful for transforming windows coordinates to a more useful coordinate system, i.e. your coordinate system.
glutGetWindow	glutGetWindow()	Call this and it will return the integer identifier of the current window.	-
glutIdleFunc	glutIdleFunc(idle)	Registers the function to be used if there are no events.	-
glutInit	glutInit(&argc, argv)	Sets up the program for using OpenGL.	You pass in argc and argv because you can change certain OpenGL settings at the command line.
glutInitDisplayMode	glutInitDisplayMode(bitfield)	Sets up the environment in which OpenGL will run.	For this project, you can usually pass GLUT_RGBA GLUT_DOUBLE GLUT_DEPTH in as the argument.
glutInitWindowPosition	glutInitWindowPosition(x, y)	Sets the initial window position in pixels.	Passing in negative values lets the windowing program choose where to put the window.
glutInitWindowSize	glutInitWindowSize(width, height)	Sets the initial window size in pixels.	The parameters have to be positive.
glutKeyboardFunc	glutKeyboardFunc(keyFunc)	Registers the function to be used if a key is pressed.	-
glutMainLoop	glutMainLoop()	Enters the main event-loop.	-
glutMouseFunc	glutMouseFunc(mouseFunc)	Registers the function to be used if mouse-button is clicked or unclicked.	-

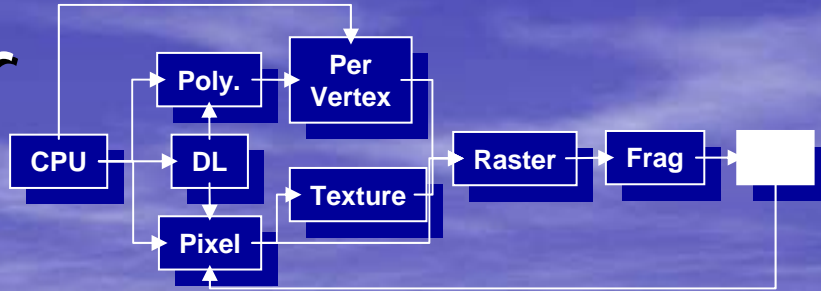
GLU functions (cont.)

<code>glutPostRedisplay</code>	<code>glutPostRedisplay()</code>	Force <code>glutDisplayFunc</code> to be called on the next iteration of the main loop.	-
<code>glutReshapeFunc</code>	<code>glutReshapeFunc(reshape)</code>	Registers the function to be used if the window is resized.	-
<code>glutSpecialFunc</code>	<code>glutSpecialFunc(specialKeyFunc)</code>	Registers the function to be used if special keys (arrows, function keys, etc.) are pressed.	-
<code>glutSwapBuffers</code>	<code>glutSwapBuffers()</code>	Prints the buffer that has been edited to the screen.	This is where the double-buffering mechanism is used. You still need to redisplay after you swap!
<code>glutTimerFunc</code>	<code>glutTimerFunc(time, timerFunc, value)</code>	Registers the function to be called at specified intervals.	Time is the time in milliseconds to wait before calling this function. Value is passed into the registered function as a parameter. Also note, once this function is called, it is no longer registered. You have to call <code>glutTimerFunc</code> from within the registered function to regularly have this called.

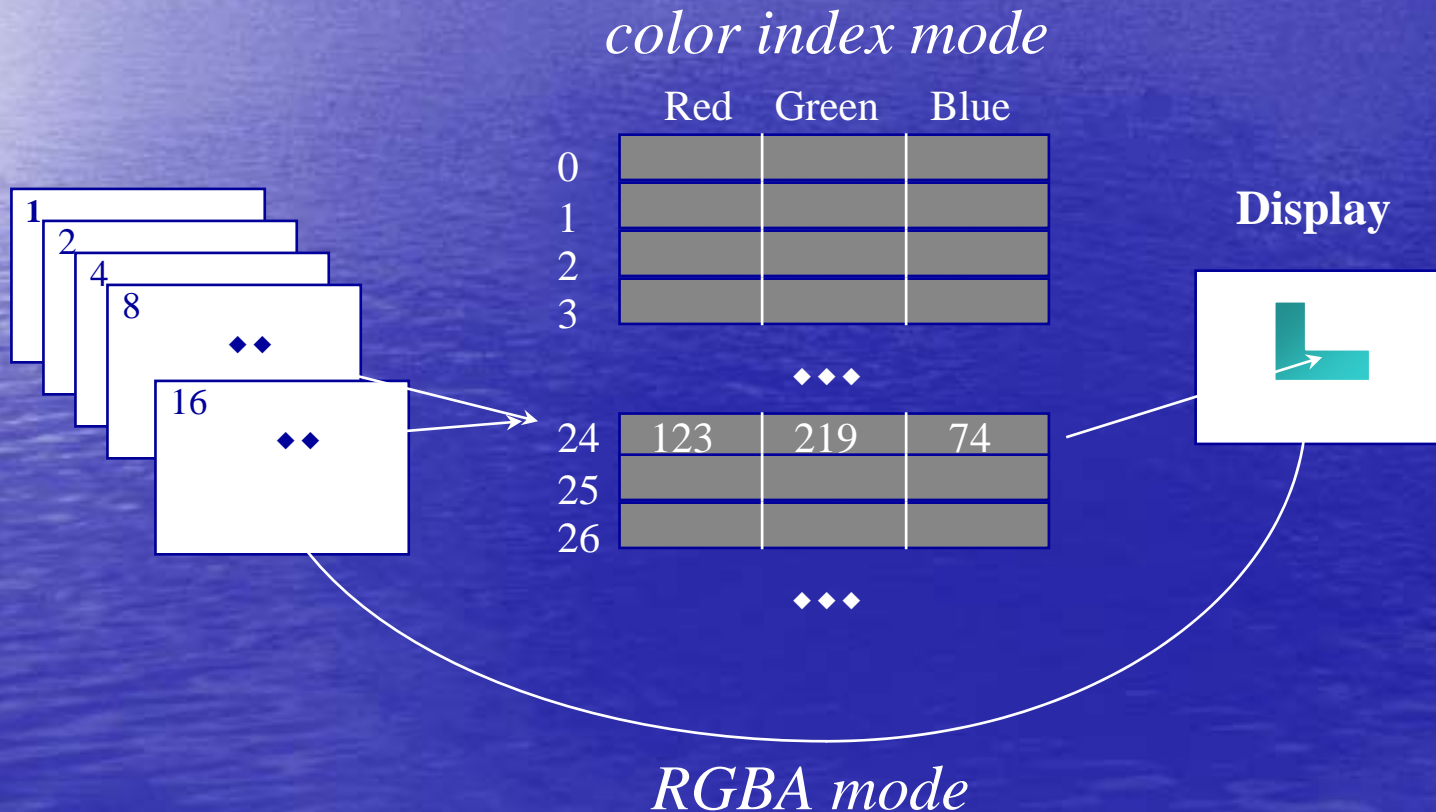
Event functions

display	<code>display()</code>	Called when the window needs to be redisplayed.	-
idle	<code>idle()</code>	Called when there are no events.	-
keyFunc	<code>keyFunc(unsigned char key, int x, int y)</code>	Called when a key is pressed.	The ASCII value of the depressed key is the <i>key</i> parameter. The window coordinates of the mouse are also passed in.
mouseFunc	<code>mouseFunc(button, state, int x, int y)</code>	Called when a mouse button is clicked or un-clicked.	The mouse button whose state has changed and the state of that button are passed in as arguments. The window coordinates of the mouse are also passed in.
reshape	<code>reshape(width, height)</code>	Called when the window is resized.	The new width and height are passed in so you can update the viewport.
specialKeyFunc	<code>specialKeyFunc(key, int x, int y)</code>	Called when a special key (arrow, function key, etc.) is pressed.	The #defined value of the depressed key is the <i>key</i> parameter. The window coordinates of the mouse are also passed in.
timerFunc	<code>timerFunc(value)</code>	Called when a specified amount of time has passed since the function was registered.	Value is the third argument passed in by <code>glutTimerFunc</code> . <i>You must re-register the function in the function if you want it called again after a specified amount of time!!</i>

OpenGL Color Models

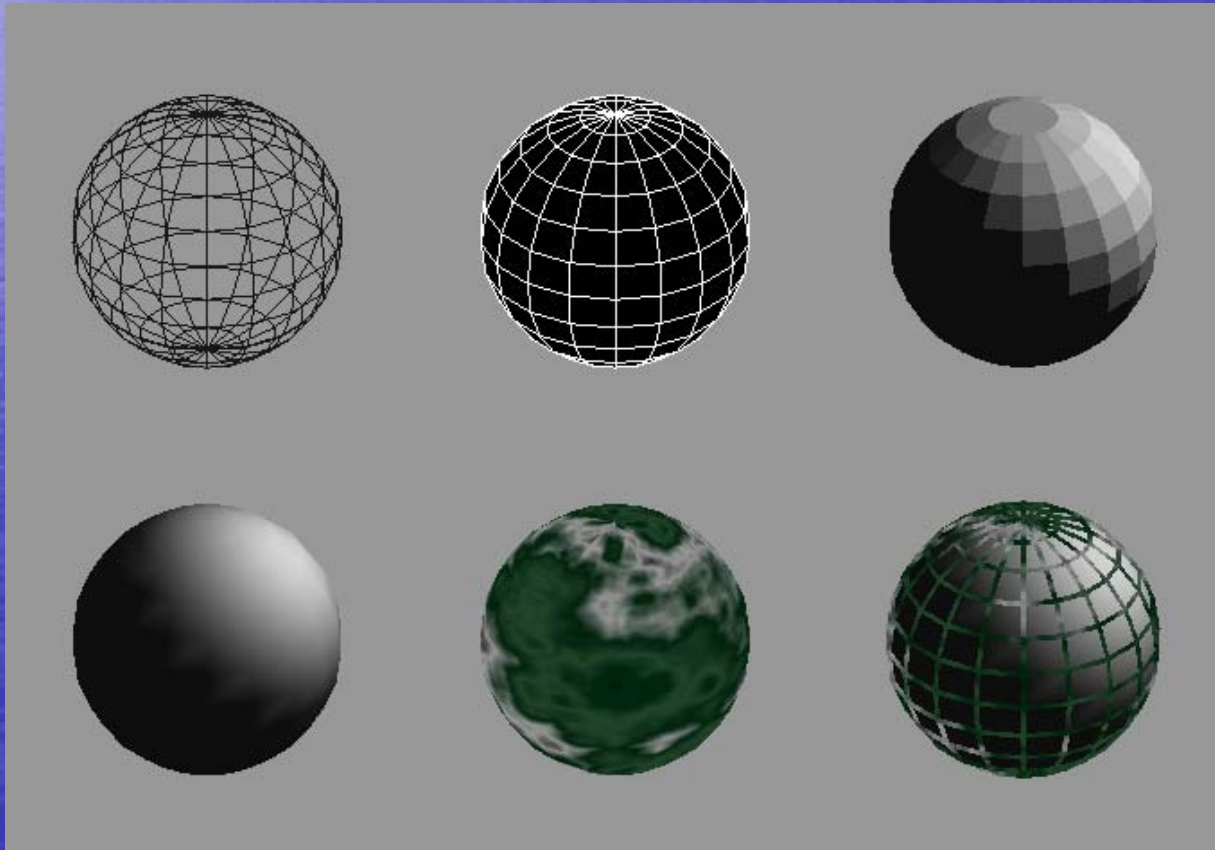


- **RGBA or Color Index**



Controlling Rendering Appearance

- From Wireframe to Texture Mapped



OpenGL's State Machine

- All rendering attributes are encapsulated in the OpenGL State
 - rendering styles
 - shading
 - lighting
 - texture mapping

Manipulating OpenGL State

- Appearance is controlled by current state for each (primitive to render) {
 update OpenGL state
 render primitive
}
- Manipulating vertex attributes is most common way to manipulate state
 glColor*() // **glIndex*()**
 glNormal*()
 glTexCoord*()

Controlling current state

- Setting State

```
glPointSize( size );
```

```
glLineStipple( repeat, pattern );
```

```
glShadeModel( GL_SMOOTH );
```

- Enabling Features

```
glEnable( GL_LIGHTING );
```

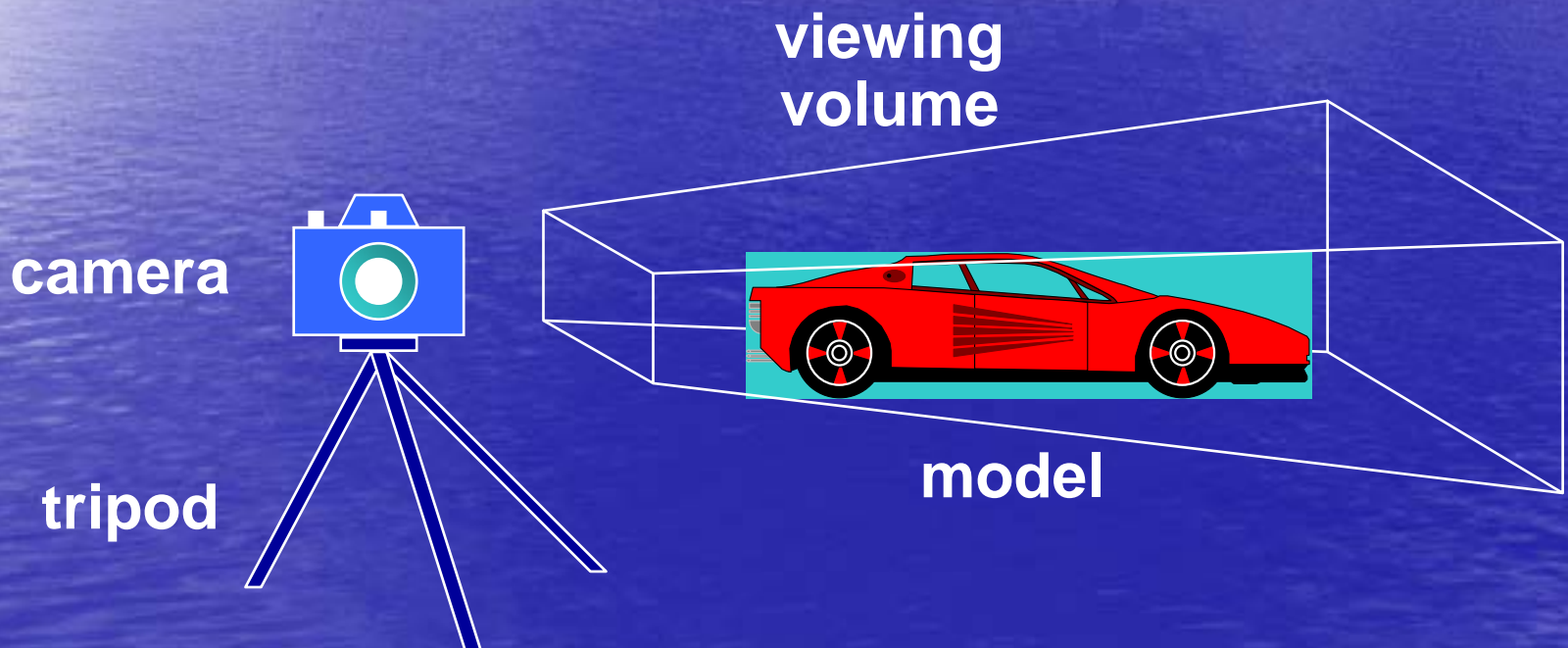
```
glDisable( GL_TEXTURE_2D );
```

Transformations in OpenGL

- Modeling
- Viewing
 - orient camera
 - projection
- Animation
- Map to screen

Camera Analogy

- 3D is just like taking a photograph (lots of photographs!)



Camera Analogy and Transformations

- Projection transformations
 - adjust the lens of the camera
- Viewing transformations
 - tripod—define position and orientation of the viewing volume in the world
- Modeling transformations
 - moving the model
- Viewport transformations
 - enlarge or reduce the physical photograph

Coordinate Systems and Transformations

- Steps in Forming an Image
 - specify geometry (world coordinates)
 - specify camera (camera coordinates)
 - project (window coordinates)
 - map to viewport (screen coordinates)
- Each step uses transformations
- Every transformation is equivalent to a change in coordinate systems (frames)

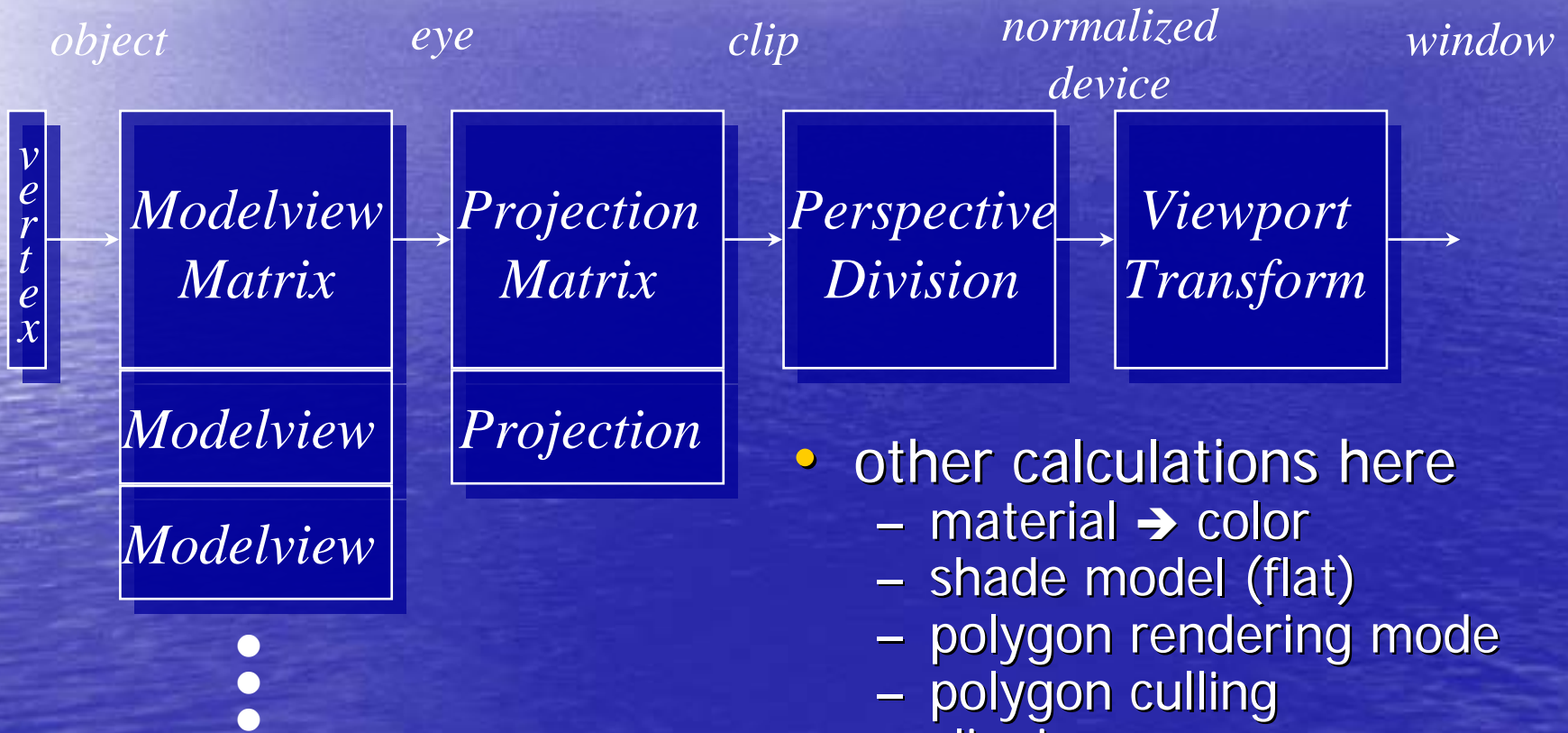
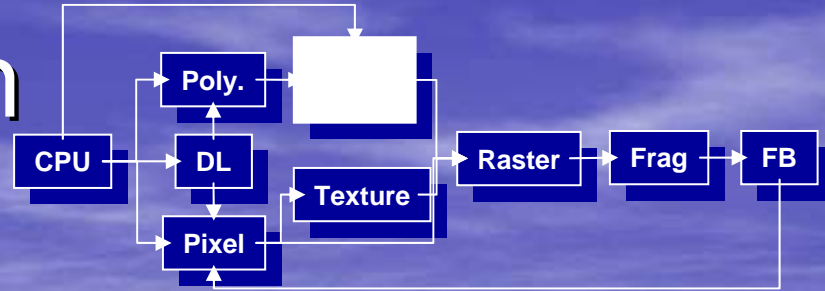
Specifying Transformations

- Programmer has two styles of specifying transformations
 - specify matrices (**glLoadMatrix**, **glMultMatrix**)
 - specify operation (**glRotate**, **glOrtho**)
- Programmer does not have to remember the exact matrices
 - check appendix of Red Book (Programming Guide)

Programming Transformations

- Prior to rendering, view, locate, and orient:
 - eye/camera position
 - 3D geometry
- Manage the matrices
 - including matrix stack
- Combine (composite) transformations

Transformation Pipeline



- other calculations here
 - material → color
 - shade model (flat)
 - polygon rendering mode
 - polygon culling
 - clipping

Matrix Operations

- Specify Current Matrix Stack

glMatrixMode(*GL_MODELVIEW* or *GL_PROJECTION*)

- Other Matrix or Stack Operations

**glLoadIdentity() glPushMatrix()
 glPopMatrix()**

- Viewport

– usually same as window size

– viewport aspect ratio should be same as projection transformation or resulting image may be distorted

glViewport(*x, y, width, height*)

Projection Transformation

- Shape of viewing frustum
- Perspective projection

```
gluPerspective( fovy, aspect, zNear, zFar )
```

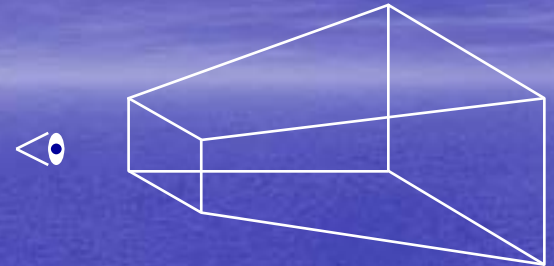
```
glFrustum( left, right, bottom, top, zNear, zFar )
```

- Orthographic parallel projection

```
glOrtho( left, right, bottom, top, zNear, zFar )
```

```
gluOrtho2D( left, right, bottom, top )
```

- calls **glOrtho** with z values near zero



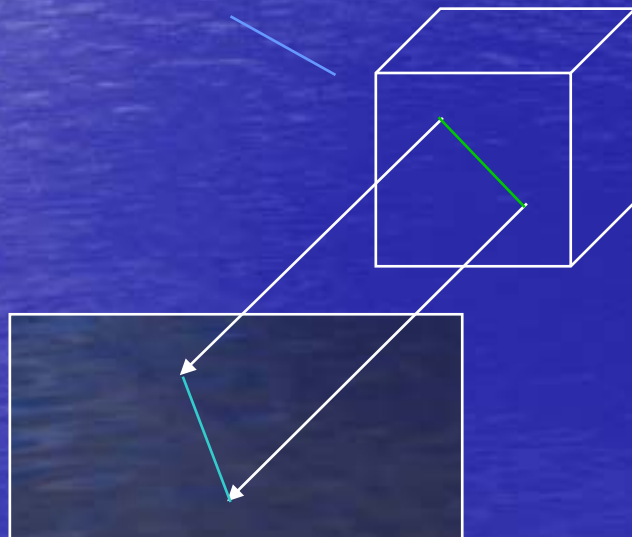
Applying Projection Transformations

- Typical use (orthographic projection)

```
glMatrixMode( GL_PROJECTION );
```

```
glLoadIdentity();
```

```
glOrtho( left, right, bottom, top, zNear,  
zFar );
```



Viewing Transformations

- Position the camera/eye in the scene
 - place the tripod down; aim camera
- To “fly through” a scene
 - change viewing transformation and redraw scene
- `gluLookAt(eyex, eyey, eyez,
aimx, aimy, aimz,
upx, upy, upz)`
 - up vector determines unique orientation
 - careful of degenerate positions



Modeling Transformations

- Move object

```
glTranslate{fd}( x, y, z )
```

- Rotate object around arbitrary axis $(x \ y \ z)$

```
glRotate{fd}( angle, x, y, z )
```

– angle is in degrees

- Dilate (stretch or shrink) or mirror object

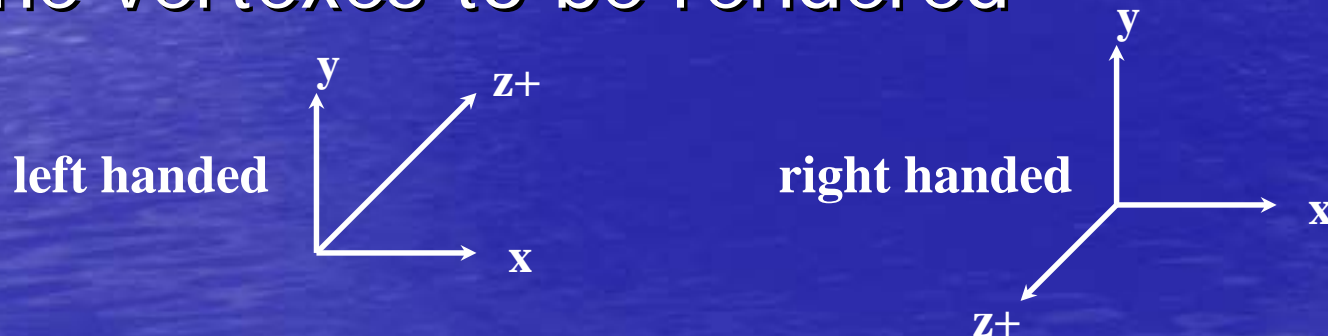
```
glScale{fd}( x, y, z )
```

Connection: Viewing and Modeling

- Moving camera is equivalent to moving every object in the world towards a stationary camera
- Viewing transformations are equivalent to several modeling transformations
`gluLookAt()` has its own command
can make your own *polar view* or *pilot view*

Projection is left handed

- Projection transformations (`gluPerspective`, `glOrtho`) are left handed
 - think of *zNear* and *zFar* as distance from view point
- Everything else is right handed, including the vertexes to be rendered



Common Transformation Usage

- 3 examples of `resize()` routine
 - restate projection & viewing transformations
- Usually called when window resized
- Registered as callback for `glutReshapeFunc()`

resize () : Perspective & LookAt

```
void resize( int w, int h )
{
    glViewport( 0, 0, (GLsizei) w, (GLsizei)
h );
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    gluPerspective( 65.0, (GLfloat) w / h,
1.0, 100.0 );
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    gluLookAt( 0.0, 0.0, 5.0,
0.0, 0.0, 0.0,
0.0, 1.0, 0.0 );
}
```

resize (): Perspective & Translate

- Same effect as previous LookAt

```
void resize( int w, int h )
{
    glViewport( 0, 0, (GLsizei) w, (GLsizei)
h );
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    gluPerspective( 65.0, (GLfloat) w/h,
1.0, 100.0 );
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    glTranslatef( 0.0, 0.0, -5.0 );
}
```


resize (): Ortho (part 1)

```
void resize( int width, int height )
{
    GLdouble aspect = (GLdouble) width /
height;
    GLdouble left = -2.5, right = 2.5;
    GLdouble bottom = -2.5, top = 2.5;
    glViewport( 0, 0, (GLsizei) w, (GLsizei) h
);
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    ... continued ...
}
```

resize(): Ortho (part 2)

```
if ( aspect < 1.0 ) {
    left /= aspect;
    right /= aspect;
} else {
    bottom *= aspect;
    top *= aspect;
}
glOrtho( left, right, bottom, top, near,
far );
glMatrixMode( GL_MODELVIEW );
glLoadIdentity();
}
```

Compositing Modeling Transformations

- Problem 1: hierarchical objects
 - one position depends upon a previous position
 - robot arm or hand; sub-assemblies
- Solution 1: moving local coordinate system
 - modeling transformations move coordinate system
 - post-multiply column-major matrices
 - OpenGL post-multiplies matrices

Compositing Modeling Transformations

- Problem 2: objects move relative to absolute world origin
 - my object rotates around the wrong origin
 - make it spin around its center or something else
- Solution 2: fixed coordinate system
 - modeling transformations move objects around fixed coordinate system
 - pre-multiply column-major matrices
 - OpenGL post-multiplies matrices
 - must reverse order of operations to achieve desired effect

Additional Clipping Planes

- At least 6 more clipping planes available
- Good for cross-sections
- Modelview matrix moves clipping plane
- $Ax + By + Cz + D < 0$ clipped
- `glEnable(GL_CLIP_PLANEi)`
- `glClipPlane(GL_CLIP_PLANEi, GLdouble*
coeff)`

Reversing Coordinate Projection

- Screen space back to world space

```
glGetIntegerv( GL_VIEWPORT, GLint viewport[4] )
glGetDoublev( GL_MODELVIEW_MATRIX, GLdouble mvmatrix[16] )
glGetDoublev( GL_PROJECTION_MATRIX,
              GLdouble projmatrix[16] )
gluUnProject( GLdouble winx, winy, winz,
              mvmatrix[16], projmatrix[16],
              GLint viewport[4],
              GLdouble *objx, *objy, *objz )
```

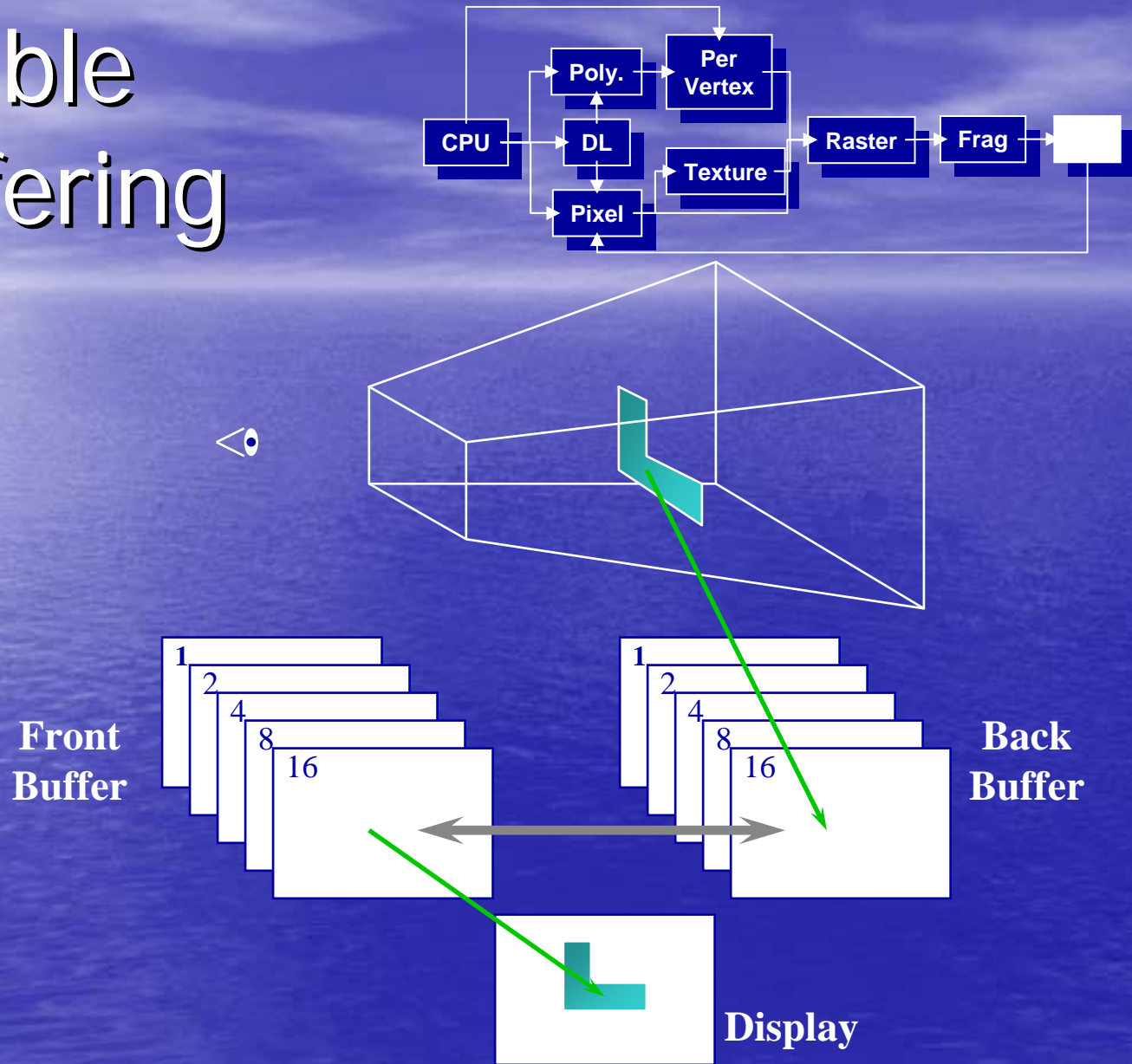
- `gluProject` goes from world to screen space

Animation and Depth Buffering

Animation and Depth Buffering

- Discuss double buffering and animation
- Discuss hidden surface removal using the depth buffer

Double Buffering



Animation Using Double Buffering

- ① Request a double buffered color buffer

```
glutInitDisplayMode( GLUT_RGB /  
GLUT_DOUBLE );
```

- ② Clear color buffer

```
glClear( GL_COLOR_BUFFER_BIT );
```

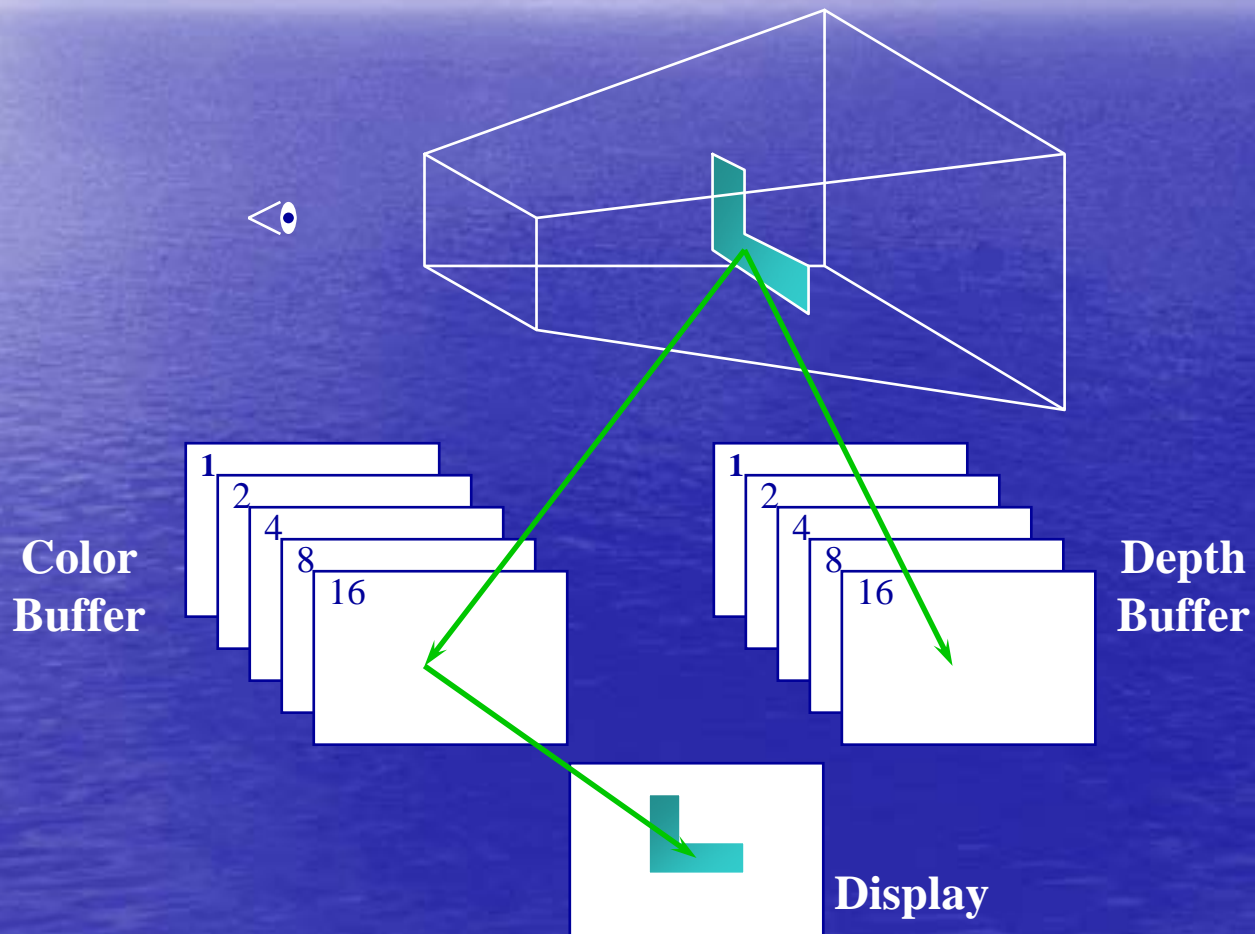
- ③ Render scene

- ④ Request swap of front and back buffers

```
glutSwapBuffers();
```

- Repeat steps 2 - 4 for animation

Depth Buffering and Hidden Surface Removal



Depth Buffering Using OpenGL

① Request a depth buffer

```
glutInitDisplayMode( GLUT_RGB /  
GLUT_DOUBLE / GLUT_DEPTH );
```

② Enable depth buffering

```
glEnable( GL_DEPTH_TEST );
```

③ Clear color and depth buffers

```
glClear( GL_COLOR_BUFFER_BIT /  
GL_DEPTH_BUFFER_BIT );
```

④ Render scene

⑤ Swap color buffers

An Updated Program Template

```
void main( int argc, char** argv )
{
    glutInit( &argc, argv );
    glutInitDisplayMode( GLUT_RGB |
        GLUT_DOUBLE | GLUT_DEPTH );
    glutCreateWindow( "Tetrahedron"
    );
    init();
    glutIdleFunc( idle );
    glutDisplayFunc( display );
    glutMainLoop();
}
```

An Updated Program Template (cont.)

```
void init( void )  
{  
    glClearColor( 0.0, 0.0, 1.0, 1.0  
);  
}
```

```
void idle( void )  
{  
    glutPostRedisplay();  
}
```

An Updated Program Template (cont.)

```
void drawScene( void )
{
    GLfloat vertices[] = { ... };
    GLfloat colors[] = { ... };
    glClear( GL_COLOR_BUFFER_BIT |
            GL_DEPTH_BUFFER_BIT );
    glBegin( GL_TRIANGLE_STRIP );
    /* calls to glColor*() and glVertex*() */
    glEnd();
    glutSwapBuffers();
}
```

Lighting

Lighting Principles

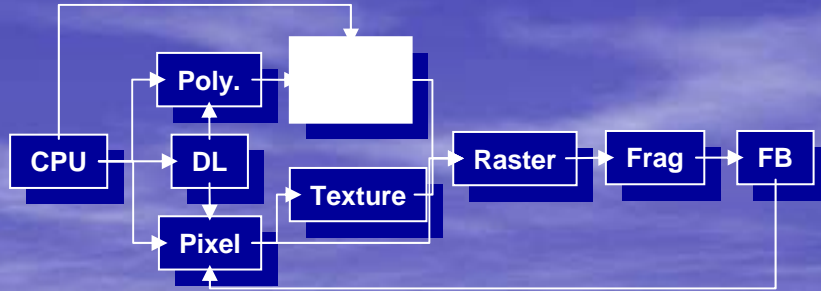
- Lighting simulates how objects reflect light
 - material composition of object
 - light's color and position
 - global lighting parameters
 - ambient light
 - two sided lighting
 - available in both color index and RGBA mode



How OpenGL Simulates Lights

- Phong lighting model
 - Computed at vertices
- Lighting contributors
 - Surface material properties
 - Light properties
 - Lighting model properties

Surface Normals



- Normals define how a surface reflects light

`glNormal3f(x, y, z)`

- Current normal is used to compute vertex's color

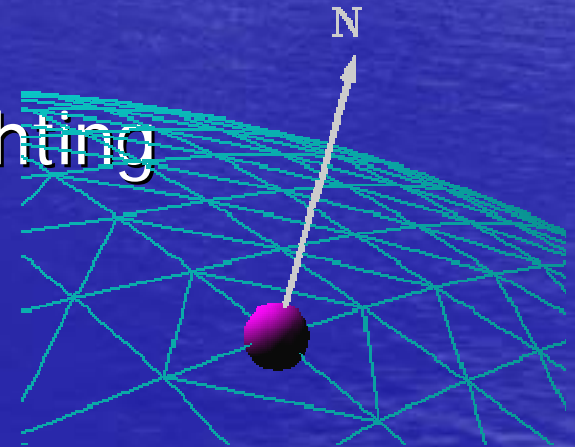
- Use *unit* normals for proper lighting

- scaling affects a normal's length

`glEnable(GL_NORMALIZE)`

or

`glEnable(GL_RESCALE_NORMAL)`



Material Properties

- Define the surface properties of a primitive

`glMaterialfv(face, property, value`

<code>GL_DIFFUSE</code>	Base color
<code>GL_SPECULAR</code>	Highlight Color
<code>GL_AMBIENT</code>	Low-light Color
<code>GL_EMISSION</code>	Glow Color
<code>GL_SHININESS</code>	Surface Smoothness

– separate materials for front and back

Light Properties

```
glLightfv( light, property, value );
```

– *light* specifies which light

- multiple lights, starting with `GL_LIGHT0`

```
glGetIntegerv( GL_MAX_LIGHTS, &n );
```

– *properties*

- colors
- position and type
- attenuation

Light Sources (cont.)

- Light color properties
 - **GL_AMBIENT**
 - **GL_DIFFUSE**
 - **GL_SPECULAR**

Types of Lights

- OpenGL supports two types of Lights
 - Local (Point) light sources
 - Infinite (Directional) light sources
- Type of light controlled by w coordinate

$w = 0$ ***Infinite Light directed along*** $(x \quad y \quad z)$

$w \neq 0$ ***Local Light positioned at*** $(\frac{x}{w} \quad \frac{y}{w} \quad \frac{z}{w})$

Turning on the Lights

- Flip each light's switch

```
glEnable( GL_LIGHTn );
```

- Turn on the power

```
glEnable( GL_LIGHTING );
```


Controlling a Light's Position

- Modelview matrix affects a light's position
 - Different effects based on when position is specified
 - eye coordinates
 - world coordinates
 - model coordinates
 - Push and pop matrices to uniquely control a light's position

Advanced Lighting Features

- Spotlights
 - localize lighting affects
 - *GL_SPOT_DIRECTION*
 - *GL_SPOT_CUTOFF*
 - *GL_SPOT_EXPONENT*

Advanced Lighting Features

- Light attenuation
 - decrease light intensity with distance
 - *GL_CONSTANT_ATTENUATION*
 - *GL_LINEAR_ATTENUATION*
 - *GL_QUADRATIC_ATTENUATION*

$$f_i = \frac{1}{k_c + k_l d + k_q d^2}$$

Light Model Properties

```
glLightModelfv( property, value );
```

- Enabling two sided lighting

```
GL_LIGHT_MODEL_TWO_SIDE
```

- Global ambient color

```
GL_LIGHT_MODEL_AMBIENT
```

- Local viewer mode

```
GL_LIGHT_MODEL_LOCAL_VIEWER
```

- Separate specular color

```
GL_LIGHT_MODEL_COLOR_CONTROL
```

Tips for Better Lighting

- Recall lighting computed only at vertices
 - model tessellation heavily affects lighting results
 - better results but more geometry to process
- Use a single infinite light for fastest lighting
 - minimal computation per vertex

Imaging and Raster Primitives

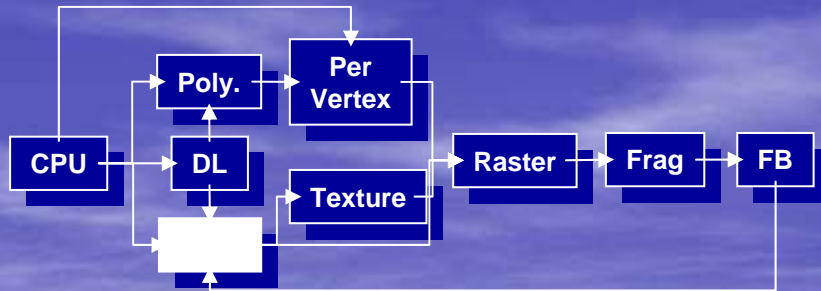
Imaging and Raster Primitives

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

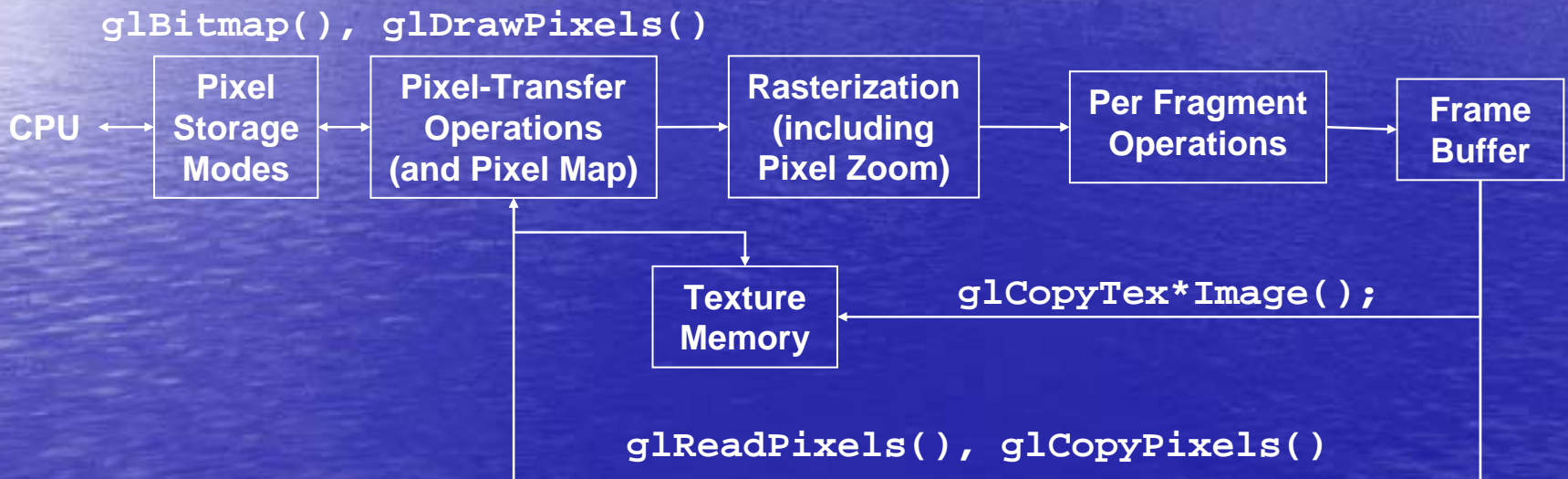
Pixel-based primitives

- 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

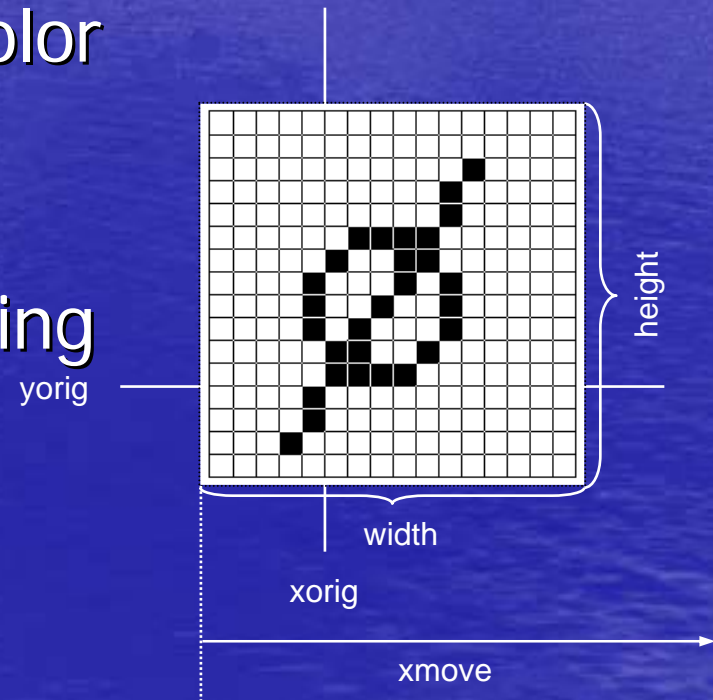


Raster Position

Rendering Bitmaps

```
glBitmap( width, height, xorig, yorig,  
         xmove, ymove, bitmap )
```

- render bitmap in current color at $(x - xorig, y - yorig)$
- advance raster position by $(xmove, ymove)$ 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 from 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

Raster
Position

```
glPixelZoom(1.0, -1.0);
```

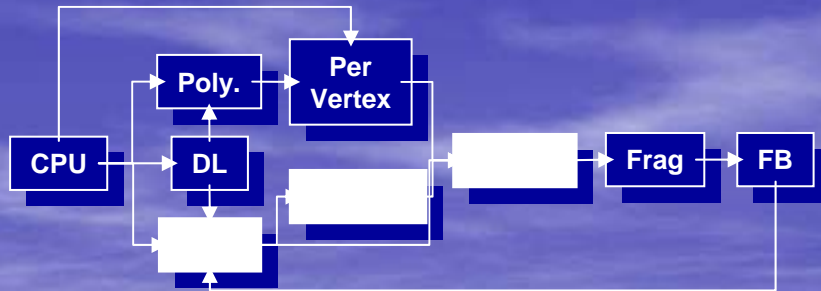


Storage and Transfer Modes

- 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

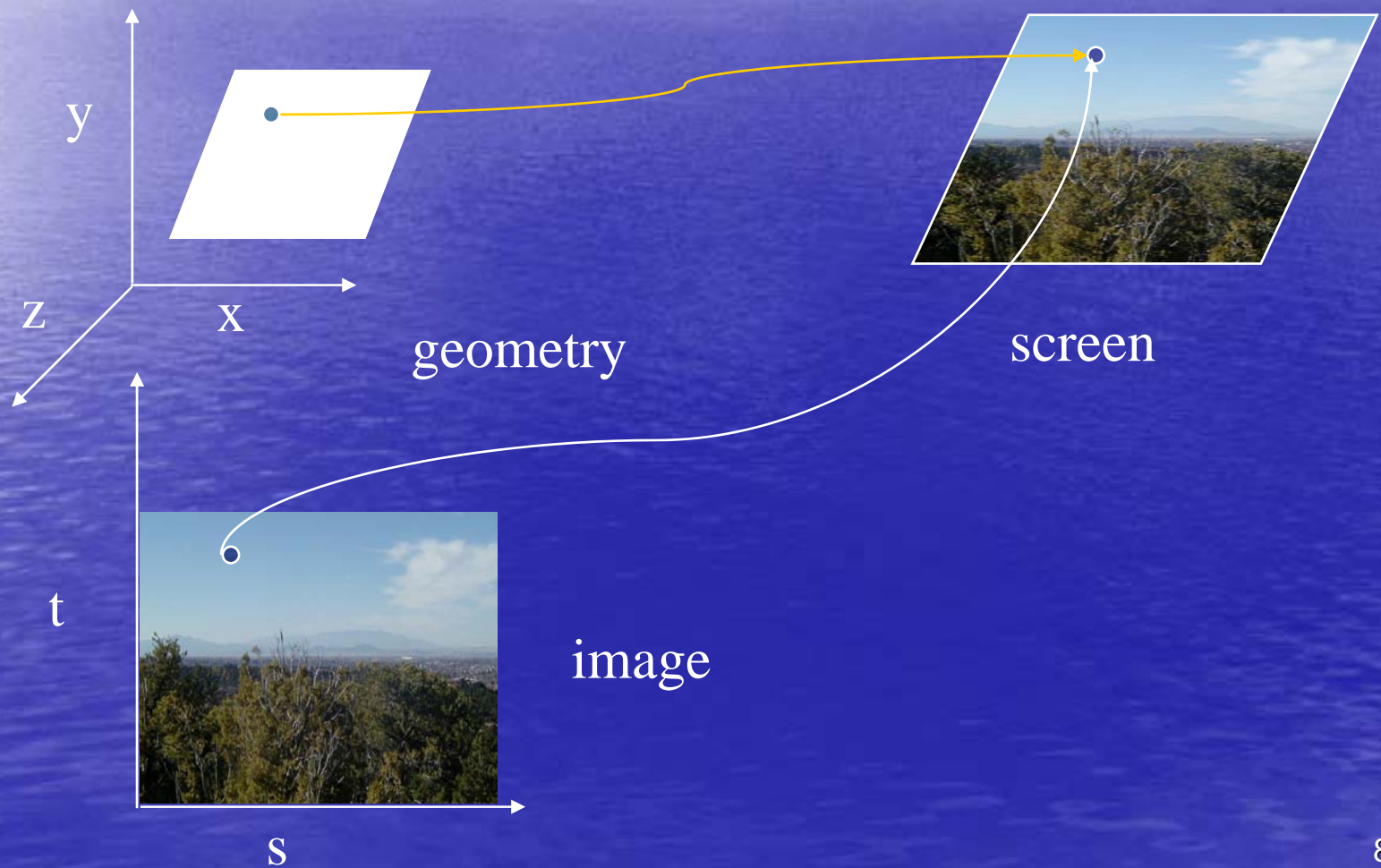
Texture Mapping

Texture Mapping



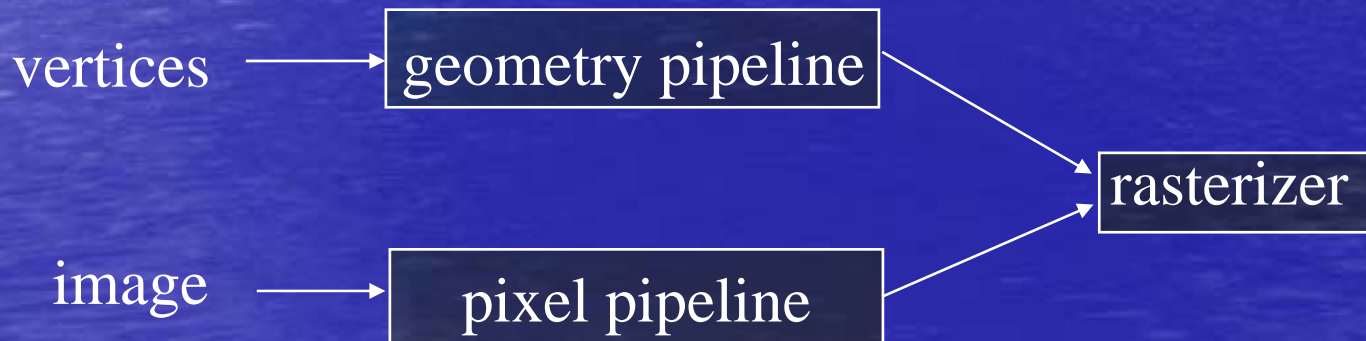
- Apply a 1D, 2D, or 3D image to geometric primitives
- Uses of Texturing
 - simulating materials
 - reducing geometric complexity
 - image warping
 - reflections

Texture Mapping



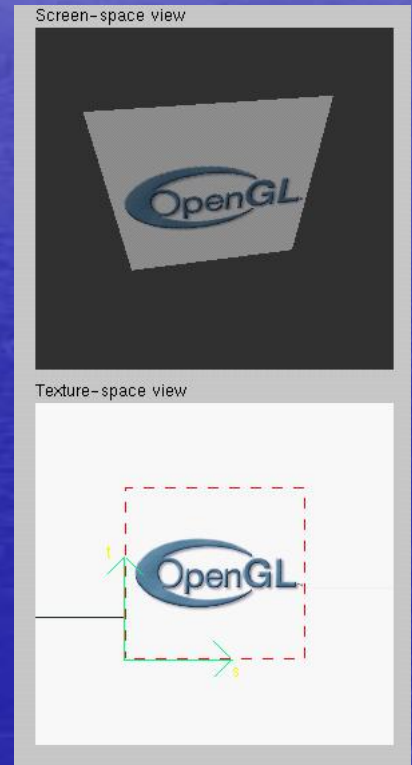
Texture Mapping and the OpenGL Pipeline

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



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.)

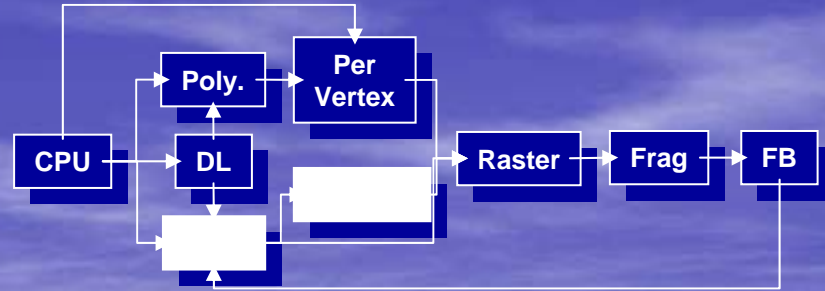
- Create texture objects with texture data and state

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

- 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 );
```

 - **_in is for source image*
 - **_out is for destination image*
- Image interpolated and filtered during scaling

Specifying a Texture: Other Methods

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

`glCopyTexImage2D(...)`

`glCopyTexImage1D(...)`

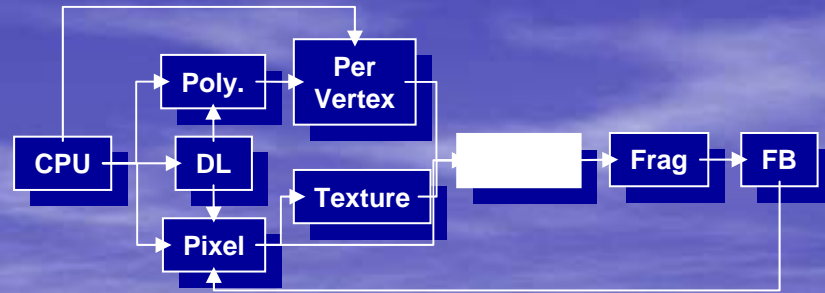
- Modify part of a defined texture

`glTexSubImage2D(...)`

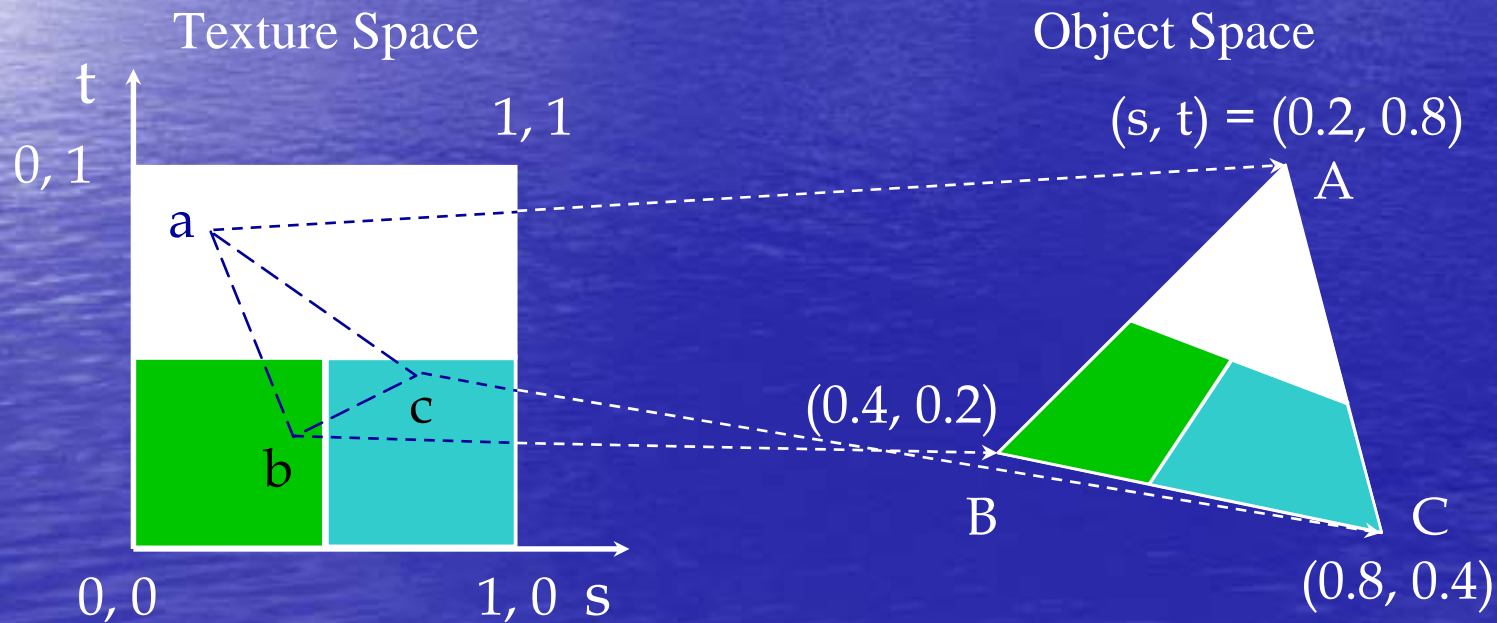
`glTexSubImage1D(...)`

- 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**

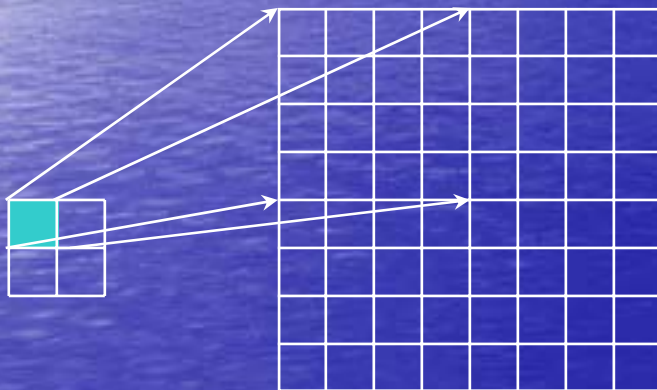
Texture Application Methods

- 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:

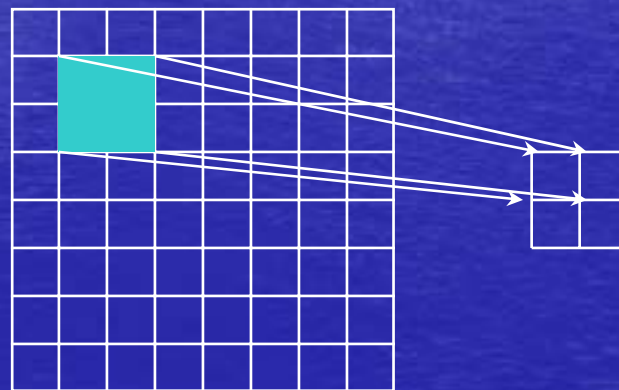
```
glTexParameteri( target, type, mode );
```



Texture

Polygon

Magnification



Texture

Polygon

Minification

Mipmapped Textures

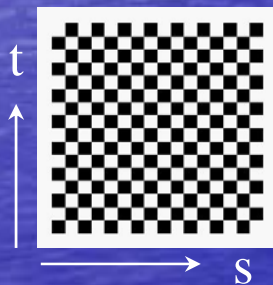
- Mipmap allows for prefiltered texture maps of decreasing resolutions
- Lessens interpolation errors for smaller textured objects
- Declare mipmap level during texture definition
`glTexImage*D(GL_TEXTURE_*D, level, ...)`
- GLU mipmap builder routines
`gluBuild*DMipmaps(...)`
- OpenGL 1.2 introduces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

Wrapping Mode

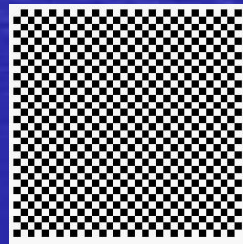
- Example:

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

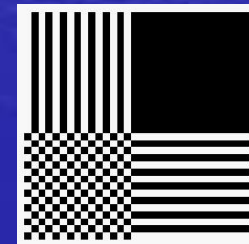
```
glTexParameteri( GL_TEXTURE_2D,  
                 GL_TEXTURE_WRAP_T, GL_REPEAT )
```



texture



GL_REPEAT
wrapping



GL_CLAMP
wrapping

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

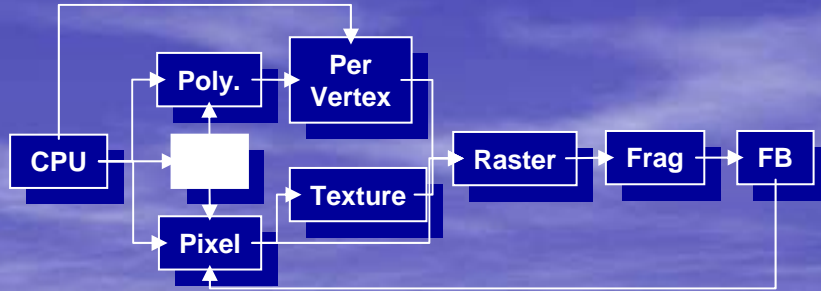
Advanced OpenGL Topics

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

Immediate Mode versus Display Listed Rendering

- Immediate Mode Graphics
 - Primitives are sent to pipeline and display right away
 - No memory of graphical entities
- Display Listed Graphics
 - Primitives placed in display lists
 - Display lists kept on graphics server
 - Can be redisplayed with different state
 - Can be shared among OpenGL graphics contexts

Display Lists



- Creating a display list

```
GLuint id;  
void init( void )  
{  
    id = glGenLists( 1 );  
    glNewList( id, GL_COMPILE );  
    /* other OpenGL routines */  
    glEndList();  
}
```

- Call a created list

```
void display( void )  
{  
    glCallList( id );  
}
```

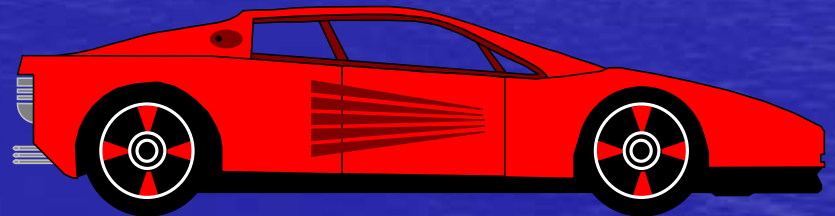
Display Lists

- Not all OpenGL routines can be stored in display lists
- State changes persist, even after a display list is finished
- Display lists can call other display lists
- Display lists are not editable, but you can fake it
 - make a list (A) which calls other lists (B, C, and D)
 - delete and replace B, C, and D, as needed

Display Lists and Hierarchy

- Consider model of a car
 - Create display list for chassis
 - Create display list for wheel

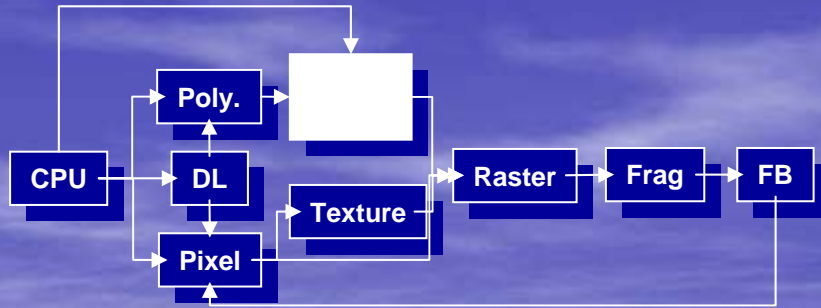
```
glNewList( CAR, GL_COMPILE );  
  glCallList( CHASSIS );  
  glTranslatef( ... );  
  glCallList( WHEEL );  
  glTranslatef( ... );  
  glCallList( WHEEL );  
  ...  
glEndList();
```



Advanced Primitives

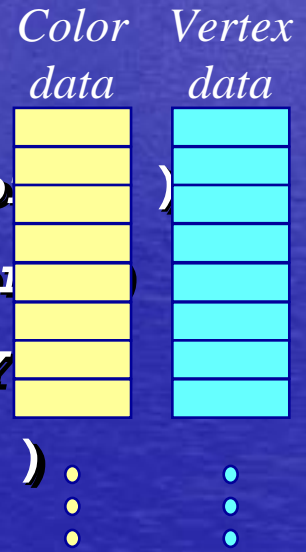
- **Vertex Arrays**
- **Bernstein Polynomial Evaluators**
 - basis for GLU NURBS
 - NURBS (Non-Uniform Rational B-Splines)
- **GLU Quadric Objects**
 - sphere
 - cylinder (or cone)
 - disk (circle)

Vertex Arrays



- Pass arrays of vertices, colors, etc. to OpenGL in a large chunk

```
glVertexPointer( 3, GL_FLOAT, 0, coord );
glColorPointer( 4, GL_FLOAT, 0, color );
glEnableClientState( GL_VERTEX_ARRAY );
glEnableClientState( GL_COLOR_ARRAY );
glDrawArrays( GL_TRIANGLE_STRIP, 0, numVerts );
```



- All active arrays are used in rendering

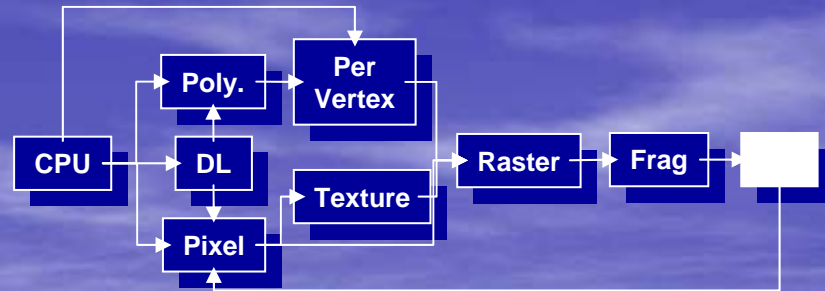
Why use Display Lists or Vertex Arrays?

- May provide better performance than immediate mode rendering
- Display lists can be shared between multiple OpenGL context
 - reduce memory usage for multi-context applications
- Vertex arrays may format data for better memory access

Alpha: the 4th Color Component

- Measure of Opacity
 - simulate translucent objects
 - glass, water, etc.
 - composite images
 - antialiasing
 - ignored if blending is not enabled
- glEnable(GL_BLEND)**

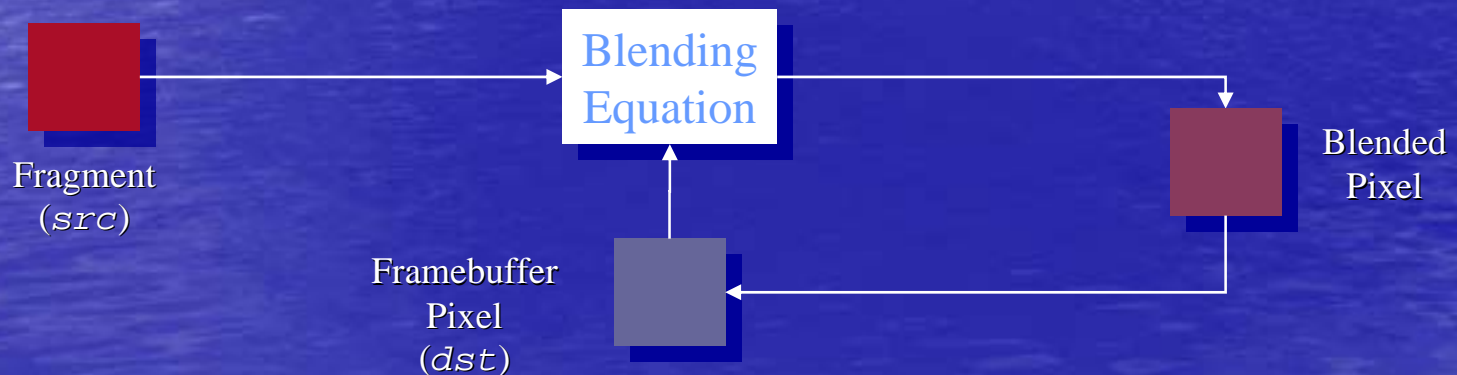
Blending



- Combine pixels with what's in already in the framebuffer

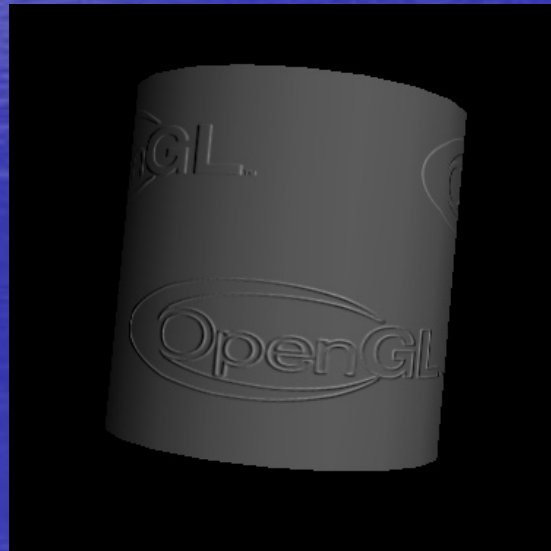
`glBlendFunc(src, dst)`

$$\vec{C}_r = src \vec{C}_f + dst \vec{C}_p$$



Multi-pass Rendering

- Blending allows results from multiple drawing passes to be combined together
 - enables more complex rendering algorithms



Example of bump-mapping
done with a multi-pass
OpenGL algorithm

Antialiasing

- Removing the Jaggies

```
glEnable( mode )
```

- **GL_POINT_SMOOTH**
- **GL_LINE_SMOOTH**
- **GL_POLYGON_SMOOTH**

- alpha value computed by computing sub-pixel coverage
- available in both RGBA and colormap modes



Accumulation Buffer

- Problems of compositing into color buffers
 - limited color resolution
 - clamping
 - loss of accuracy
 - Accumulation buffer acts as a “floating point” color buffer
 - accumulate into accumulation buffer
 - transfer results to frame buffer

Accessing Accumulation Buffer

`glAccum(op, value)`

– operations

- within the accumulation buffer: *GL_ADD*, *GL_MULT*
- from read buffer: *GL_ACCUM*, *GL_LOAD*
- transfer back to write buffer: *GL_RETURN*

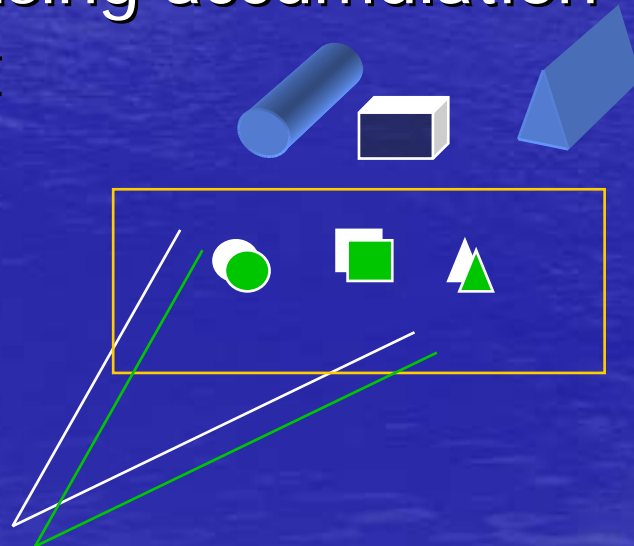
– `glAccum(GL_ACCUM, 0.5)` multiplies each value in write buffer by 0.5 and adds to accumulation buffer

Accumulation Buffer Applications

- Compositing
- Full Scene Antialiasing
- Depth of Field
- Filtering
- Motion Blur

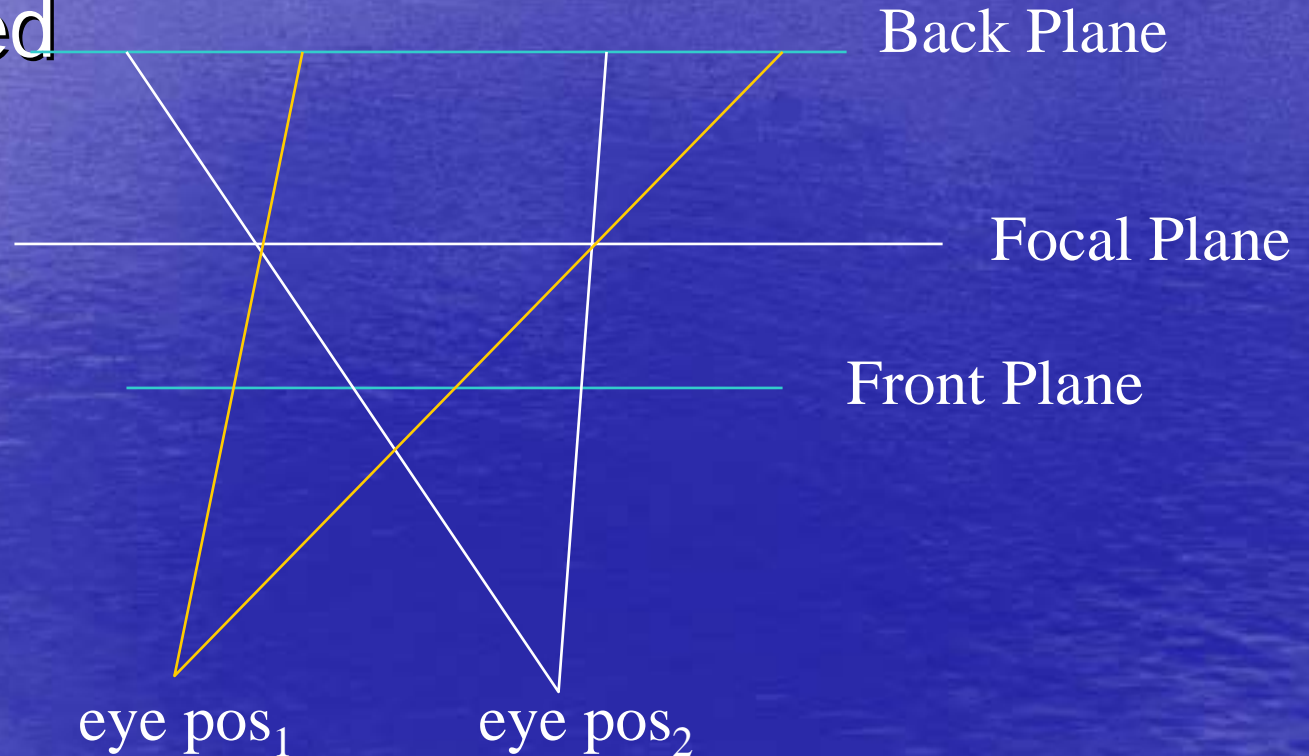
Full Scene Antialiasing : *Jittering the view*

- Each time we move the viewer, the image shifts
 - Different aliasing artifacts in each image
 - Averaging images using accumulation buffer averages out these artifacts



Depth of Focus : *Keeping a Plane in Focus*

- Jitter the viewer to keep one plane unchanged



Fog

`glFog(property, value)`

- Depth Cueing
 - Specify a range for a linear fog ramp
 - **GL_FOG_LINEAR**
- Environmental effects
 - Simulate more realistic fog
 - **GL_FOG_EXP**
 - **GL_FOG_EXP2**

Feedback Mode

- Transformed vertex data is returned to the application, not rendered
 - useful to determine which primitives will make it to the screen
- Need to specify a feedback buffer
`glFeedbackBuffer(size, type, buffer)`
- Select feedback mode for rendering
`glRenderMode(GL_FEEDBACK)`

Selection Mode

- Method to determine which primitives are inside the viewing volume
- Need to set up a buffer to have results returned to you

`glSelectBuffer(size, buffer)`

- Select selection mode for rendering

`glRenderMode(GL_SELECT)`

Selection Mode (cont.)

- To identify a primitive, give it a name
 - “names” are just integer values, not strings
- Names are stack based
 - allows for hierarchies of primitives
- Selection Name Routines

```
glLoadName( name ) glPushName( name )  
glInitNames( )
```

Picking

- Picking is a special case of selection
- Programming steps
 - restrict “drawing” to small region near pointer
USE `gluPickMatrix()` on projection matrix
 - enter selection mode; re-render scene
 - primitives drawn near cursor cause hits
 - exit selection; analyze hit records

Picking Template

```
glutMouseFunc( pickMe );
```

```
void pickMe( int button, int state, int x, int  
y )  
{  
    GLuint nameBuffer[256];  
    GLint hits;  
    GLint myViewport[4];  
    if (button != GLUT_LEFT_BUTTON ||  
        state != GLUT_DOWN) return;  
    glGetIntegerv( GL_VIEWPORT, myViewport );  
    glSelectBuffer( 256, nameBuffer );  
    (void) glRenderMode( GL_SELECT );  
    glInitNames();
```

Picking Template (cont.)

```
glMatrixMode( GL_PROJECTION );
glPushMatrix();
glLoadIdentity();
gluPickMatrix( (GLdouble) x, (GLdouble)
    (myViewport[3]-y), 5.0, 5.0, myViewport
);
/*    gluPerspective or glOrtho or other
projection    */
glPushName( 1 );
/*    draw something    */
glLoadName( 2 );
/*    draw something else ... continue ...    */
```

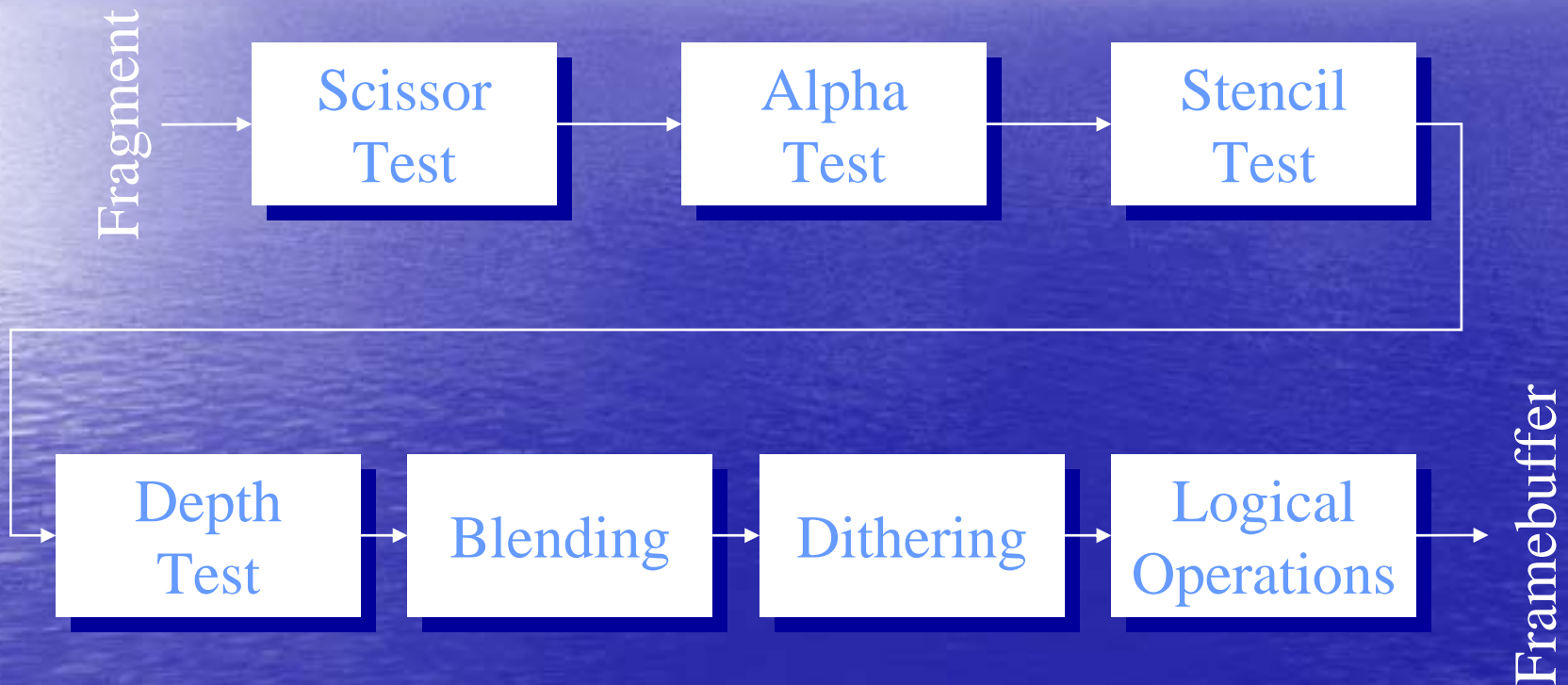

Picking Template (cont.)

```
glMatrixMode( GL_PROJECTION );  
glPopMatrix();  
hits = glRenderMode( GL_RENDER );  
/*   process nameBuffer   */  
}
```

Picking Ideas

- For OpenGL Picking Mechanism
 - only render what is pickable (e.g., don't clear screen!)
 - use an “invisible” filled rectangle, instead of text
 - if several primitives drawn in picking region, hard to use z values to distinguish which primitive is “on top”
- Alternatives to Standard Mechanism
 - color or stencil tricks (for example, use `glReadPixels()` to obtain pixel value from back buffer)

Getting to the Framebuffer



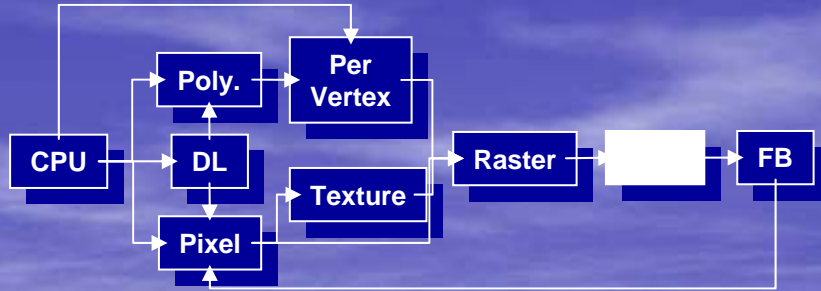
Scissor Box

- Additional Clipping Test

`glScissor(x, y, w, h)`

- any fragments outside of box are clipped
- useful for updating a small section of a viewport
 - affects `glClear()` operations

Alpha Test



- Reject pixels based on their alpha value

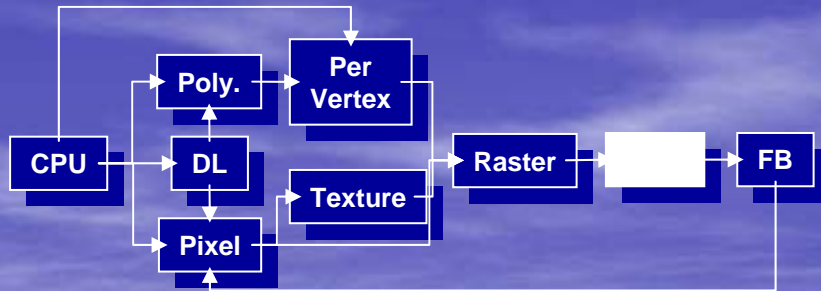
`glAlphaFunc(func, value)`

`glEnable(GL_ALPHA_TEST)`

- use alpha as a mask in textures



Stencil Buffer



- Used to control drawing based on values in the stencil buffer
 - Fragments that fail the stencil test are not drawn
 - Example: create a mask in stencil buffer and draw only objects not in mask area



Controlling Stencil Buffer

`glStencilFunc(func, ref, mask)`

- compare value in buffer with **ref** using **func**
- only applied for bits in **mask** which are 1
- **func** is one of standard comparison functions

`glStencilOp(fail, zfail, zpass)`

- Allows changes in stencil buffer based on passing or failing stencil and depth tests:
GL_KEEP, **GL_INCR**

Creating a Mask

```
glInitDisplayMode( ...|GLUT_STENCIL|... );  
glEnable( GL_STENCIL_TEST );  
glClearStencil( 0x1 );
```

```
glStencilFunc( GL_ALWAYS, 0x1, 0x1 );  
glStencilOp( GL_REPLACE, GL_REPLACE,  
            GL_REPLACE );
```

- *draw mask*

Using Stencil Mask

```
glStencilFunc( GL_EQUAL, 0x1, 0x1 )
```

- draw objects where stencil = 1

```
glStencilFunc( GL_NOT_EQUAL, 0x1, 0x1  
              );
```

```
glStencilOp( GL_KEEP, GL_KEEP, GL_KEEP  
            );
```

- draw objects where stencil != 1

Dithering

```
glEnable( GL_DITHER )
```

- Dither colors for better looking results
 - Used to simulate more available colors

Logical Operations on Pixels

- Combine pixels using bitwise logical operations

`glLogicOp(mode)`

– Common modes

- `GL_XOR`
- `GL_AND`

Advanced Imaging

- Imaging Subset
 - Only available if `GL_ARB_imaging` defined
 - Color matrix
 - Convolutions
 - Color tables
 - Histogram
 - MinMax
 - Advanced Blending

On-Line Resources

- <http://www.opengl.org>
 - start here; up to date specification and lots of sample code
- <news:comp.graphics.api.opengl>
- <http://www.sgi.com/software/opengl>
- <http://www.mesa3d.org/>
 - Brian Paul's Mesa 3D
- <http://www.cs.utah.edu/~narobins/opengl.html>
 - very special thanks to Nate Robins for the OpenGL Tutors
 - source code for tutors available here!

Books

- OpenGL Programming Guide, 3rd Edition
- OpenGL Reference Manual, 3rd Edition
- OpenGL Programming for the X Window System
 - includes many GLUT examples
- Interactive Computer Graphics: A top-down approach with OpenGL, 2nd Edition

Thanks for Coming

- Questions and Answers

Dave Shreiner

shreiner@sgi.com

Ed Angel

angel@cs.unm.edu

Vicki Shreiner

vshreiner@sgi.com