OpenGL (Open Graphics Library) is a software interface (API) to graphics hardware with over 700 commands (library functions), including $\sim 50$ in the OpenGL Utility Library (GLU).

- OpenGL is hardware independent, and therefore requires an interface to a windowing system (Microsoft Windows or X Windows).
- OpenGL enables interactive 3-D applications with realistic rendering and animation, but only at a low-level using geometric primitives (points, lines, and polygons).
- OpenGL may be implemented for a networked environment (X Windows) with a client running on a remote machine, and a server running locally.
- OpenGL is a state machine with current state defined by state variables such as the current drawing color and modelview matrix, each with a default value. This is more efficient than including the required attributes with every drawing command.
- OpenGL has bindings to C/C++, Fortran 90, Ada, Java, etc.
OpenGL commands are prefixed with gl or glu, use capitalized words, and may include a suffix of one to three characters: number of arguments, type, and 'v' for vector (pointer). For example,

\[ \text{glColor3f}(1.0f, 0.0f, 0.0f); \]

is equivalent to

\[ \text{GLfloat color_array[]} = \{1.0f, 0.0f, 0.0f\}; \text{glColor3fv(color_array);} \]

OpenGL constants are prefixed by GL_, and consist of one or more words, all uppercase, separated by underscores; e.g., GL_COLOR_BUFFER_BIT.

OpenGL type names (defined on page 8 of the Redbook) have the form GLtype; e.g., GLdouble, GLfloat, GLint. The OpenGL types are recommended for portability but unfortunately are not used by GLUT.
Modern processors use hardware pipelines (*instruction level parallelism*) in which an instruction is executed as a sequence of stages, each with dedicated hardware, and all stages executed concurrently. At each clock cycle a new instruction enters the first stage, and all currently executing instructions advance to the next stage so that, as long as the pipe is full, a new instruction is completed at each cycle. The complexity of the stages must be consistent with the clock cycle time.

Similarly, GPU’s have pipelines, originally just for integer operations on fragments associated with pixels, but now also for floating-point operations on vertices (transform and lighting operations).

OpenGL uses an implementation-dependent software pipeline, originally fixed but now programmable through shading languages.
The OpenGL rendering pipeline operates on two types of data. *Vertex data* includes color intensities, texture coordinates, and geometric data (vertex coordinates, vertex normals, line segments, polygons, and control points). *Pixel data* includes pixels, images, and bitmaps. The stages of the fixed rendering pipeline are as follows.

- **Display Lists** can be used to store sequences of OpenGL commands for later execution, as opposed to executing the commands in immediate mode; e.g., render a tricycle by saving the wheel geometry in a display list and executing the list three times, each with a different translation and scaling (modelview matrix). Display lists are stored on the server, possibly in an optimized form such as a single matrix representing a sequence of rotations, and only need to be transmitted once. Not all commands can be stored in a display list, and lists cannot be modified. (Chapter 7)
• **Evaluators** convert control points to vertex data. Control points, along with basis functions, provide compact representations of B-spline and Bézier curves and surfaces. (Chapter 12)

• **Per-vertex Operations** consist of applying the modelview and projection matrices to vertices and normals, generation of texture coordinates, and lighting calculations (T & L).

• **Primitive Assembly** includes clipping, perspective division, backface culling, and viewport mapping. The result is geometric primitives with vertices in window coordinates with colors, depths, and texture coordinate values.
Pixel Operations: unpack from memory storage format into components, scale and bias, process by a pixel map, clamp, and write to texture memory or send to the rasterization stage. Data from the framebuffer is packed and returned to system memory after the pixel operations are applied, or written back to the framebuffer, or written to texture memory.

Texture Assembly of texture objects. (Chapter 9)

Rasterization is the conversion of geometric and pixel data into fragments, where a fragment is a set of color, depth, and texture-coordinate values associated with a pixel (position in the frame buffer). Conversion includes scan conversion of lines, and linear interpolation of vertex colors and depths.
**Fragment Operations** alter or eliminate fragments before they are written to the frame buffer and become pixels. Each can be enabled or disabled. (Chapters 6, 10). The sequence of operations is

- Apply a texel from texture memory to each fragment
- Fog
- Scissor test
- Alpha test
- Stencil test
- Depth buffer test
- Blending
- Dithering
- Logic
- Masking by a bitmask
A typical OpenGL program needs the following header files. The order of the first two files is important.

```
#include <stdlib.h>
#include <GL/glut.h>  (This includes gl.h and glu.h)
#include <stdio.h>    (if using C I/O)
#include <math.h>     (if using C math library)
```

For portability, replace the second statement by the following:

```
#ifdef __APPLE__
#include <GLUT/glut.h>
#else
#include <GL/glut.h>
#endif
```
The OpenGL Utility Toolkit (GLUT) by Mark Kilgard provides a simple interface to the window system.

- **void glutInit(int *argc, char **argv)** initializes GLUT and processes command line arguments such as -display and -geometry for X Windows. This function must be called first.

- **void glutInitDisplayMode(unsigned int mode)** specifies RGBA or indexed color, a single or double buffered window, and the buffers to be used; e.g.,
  
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH).

- **void glutInitWindowPosition(int x, int y)** specifies screen coordinates of the upper left corner of the window, where (0,0) is the upper left corner of the screen.
void glutInitWindowSize(int width, int height) specifies the window size in screen coordinates.

int glutCreateWindow(char *string) creates a window with an OpenGL context, and returns a unique positive integer identifier. The string appears in the title bar.

void glutDisplayFunc(void (*func)(void)) registers display callback function (named display by convention if there is only one window).

void glutPostRedisplay(void) marks the current window as needing to be redrawn.

void glutMainLoop(void) starts the GLUT event processing loop (from which there is no return).
void glutReshapeFunc(void (*func)(int w, int h))
registers function called when the window is resized. The
default reshape function calls glViewport(0,0,w,h).

void glutKeyboardFunc(void (*func)(unsigned char
key, int x, int y)) registers a callback function for
keypresses associated with ASCII characters.

void glutSpecialFunc(void (*func)(int key, int x,
int y)) registers a callback function for keypresses associated
with function keys and directional keys. Parameter key is
GLUT_KEY_*, where * is F1, F2, ..., F12, LEFT, UP, RIGHT,
DOWN, PAGE_UP, PAGE_DOWN, HOME, END, or INSERT.

int glutGetModifiers(void) returns the modifier key
state (GLUT_ACTIVE_SHIFT, GLUT_ACTIVE_CTRL, or
GLUT_ACTIVE_ALT) when a keyboard, special, or mouse
callback is generated.
void glutMouseFunc(void (*func)(int button, int state, int x, int y)) registers a callback associated with a mouse button press or release. The first parameter is GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON, or GLUT_RIGHT_BUTTON, state is GLUT_UP or GLUT_DOWN, and (x,y) is the mouse position in window coordinates.

void glutMotionFunc(void (*func)(int x, int y)) registers a function called when the mouse pointer moves within the window with a mouse button down.

void glutPassiveMotionFunc(void (*func)(int x, int y)) registers a function called when the mouse pointer moves within the window, and no mouse button is pressed.

void glutIdleFunc(void (*func)(void)) registers a function called continuously when events are not being received. Disable the idle callback by passing NULL to glutIdleFunc.
A GLUT menu may be attached to the right mouse button by code similar to the following.

```c
void makeMenu(void)
{
    glutCreateMenu(menu);
    glutAddMenuEntry("x:  Rotate about the x axis", 'x');
    glutAddMenuEntry("y:  Rotate about the y axis", 'y');
    glutAttachMenu(GLUT_RIGHT_BUTTON);
    return;
}
```

Note that the characters whose ASCII values identify the menu entries appear in the strings. This alerts the user to the keypress alternatives.
An example menu callback function follows. The keyboard callback function should call this function rather than duplicating the code.

```c
void menu(int item)
{
    switch (item) {
    case 'x':
        x_rotation_angle += x_increment;
        break;
    case 'y':
        y_rotation_angle += y_increment;
        break;
    }
    glutPostRedisplay();
    return;
}
```
Animation

Analog movies display 24 frames per second (16 fps for old movies) by mechanically moving frames into position behind the lens in sync with opening and closing the shutter. The brain sees smooth motion.

The refresh rate of a CRT is typically 60 to 120 cycles per second, while an LCD monitor can have a refresh rate as high as 600 Hz. In the case of a CRT, a low refresh rate results in flicker as the light intensity fades and brightens in each cycle.

A fundamental difference between movies and animated graphics, such as games, is that the graphics system is creating the images on the fly. What if we did this?

```c
for (i=0; i<n; i++) {
    clear_window();
    draw_frame(i);
    wait_for_end_of_24th_of_a_second_time_interval();
}
```
Problem The portion of the picture drawn early is visible for most of the $(1/24)$-second time interval, while the portion drawn last may disappear immediately.

Solution Double-buffering — display one buffer while the image is being drawn in the other. This is like a two-frame movie.

Problem If we swap buffers whenever we are done drawing the frame, the frame rate is erratic. If we swap every 24-th of a second, the swap may not coincide with the end of the refresh cycle, so that the image changes while it is visible; i.e., during the remainder of the current refresh cycle, part of the screen image reflects the contents of one buffer, and part of it came from the other buffer.

Solution Synchronize the frame swap with the refresh cycle.

void glutSwapBuffers(void) waits for the end of the refresh cycle and then swaps buffers.
Suppose the refresh rate is 60 Hz (cps). Then the maximum frame rate is 60 fps. If the time required to create a frame is more than 1/60 seconds but less than 1/30 seconds, then the frame rate is 30 fps (two refresh cycles per frame). If 1/30 seconds is not enough time, the frame rate drops to $60/3 = 20$ fps.

If the frame rate is close to $(1/k)$ times the refresh rate for some integer $k$, it may be irregular due to random variations or different frame complexities. It might be necessary to add a small delay to consistently get the slower rate. The complexities often do not differ much because the frames are associated with moving the eye position or camera position and redrawing the same scene.

Refer to double.c in the Redbook.
1. What are the differences among the OpenGL, GLU, and GLUT libraries?
Review Questions

1. What are the differences among the OpenGL, GLU, and GLUT libraries?
2. What is a state machine?
1. What are the differences among the OpenGL, GLU, and GLUT libraries?
2. What is a state machine?
3. What are the OpenGL naming conventions for functions, enumerated constants, and types?
Review Questions

1. What are the differences among the OpenGL, GLU, and GLUT libraries?
2. What is a state machine?
3. What are the OpenGL naming conventions for functions, enumerated constants, and types?
4. How do you set the current drawing color?
Review Questions

1. What are the differences among the OpenGL, GLU, and GLUT libraries?
2. What is a state machine?
3. What are the OpenGL naming conventions for functions, enumerated constants, and types?
4. How do you set the current drawing color?
5. What is the difference between a fixed and programmable pipeline?
1. What are the differences among the OpenGL, GLU, and GLUT libraries?
2. What is a state machine?
3. What are the OpenGL naming conventions for functions, enumerated constants, and types?
4. How do you set the current drawing color?
5. What is the difference between a fixed and programmable pipeline?
6. What is vertex data, geometric data, and pixel data?
Review Questions

1. What are the differences among the OpenGL, GLU, and GLUT libraries?
2. What is a state machine?
3. What are the OpenGL naming conventions for functions, enumerated constants, and types?
4. How do you set the current drawing color?
5. What is the difference between a fixed and programmable pipeline?
6. What is vertex data, geometric data, and pixel data?
7. How do display lists improve performance? Where are they stored? What are their limitations?
Review Questions

1. What are the differences among the OpenGL, GLU, and GLUT libraries?
2. What is a state machine?
3. What are the OpenGL naming conventions for functions, enumerated constants, and types?
4. How do you set the current drawing color?
5. What is the difference between a fixed and programmable pipeline?
6. What is vertex data, geometric data, and pixel data?
7. How do display lists improve performance? Where are they stored? What are their limitations?
8. What are control points? What are evaluators?
Review Questions

1. What are the differences among the OpenGL, GLU, and GLUT libraries?
2. What is a state machine?
3. What are the OpenGL naming conventions for functions, enumerated constants, and types?
4. How do you set the current drawing color?
5. What is the difference between a fixed and programmable pipeline?
6. What is vertex data, geometric data, and pixel data?
7. How do display lists improve performance? Where are they stored? What are their limitations?
8. What are control points? What are evaluators?
9. What are the geometric primitives?
10. What is rasterization?
10. What is rasterization?

11. What is a fragment?
10. What is rasterization?
11. What is a fragment?
12. What are vertex normals? How are they related to lighting?
10. What is rasterization?
11. What is a fragment?
12. What are vertex normals? How are they related to lighting?
13. What are vertex texture coordinates?
Review Questions continued

10. What is rasterization?
11. What is a fragment?
12. What are vertex normals? How are they related to lighting?
13. What are vertex texture coordinates?
14. What is the modelview matrix?
10. What is rasterization?
11. What is a fragment?
12. What are vertex normals? How are they related to lighting?
13. What are vertex texture coordinates?
14. What is the modelview matrix?
15. What is T & L?
Review Questions continued

10. What is rasterization?
11. What is a fragment?
12. What are vertex normals? How are they related to lighting?
13. What are vertex texture coordinates?
14. What is the modelview matrix?
15. What is T & L?
16. What is clipping? Backface culling? Depth?
10 What is rasterization?
11 What is a fragment?
12 What are vertex normals? How are they related to lighting?
13 What are vertex texture coordinates?
14 What is the modelview matrix?
15 What is T & L?
16 What is clipping? Backface culling? Depth?
17 What is a viewport?
10. What is rasterization?
11. What is a fragment?
12. What are vertex normals? How are they related to lighting?
13. What are vertex texture coordinates?
14. What is the modelview matrix?
15. What is T & L?
16. What is clipping? Backface culling? Depth?
17. What is a viewport?
18. What is the purpose of double buffering?