Graphics and Visualization

Holger Kenn

International University Bremen

Spring Semester 2006

First steps in OpenGL

Coordinates

Representing Curves

Recap

- Display Devices
- First “Lab Course”

OpenGL

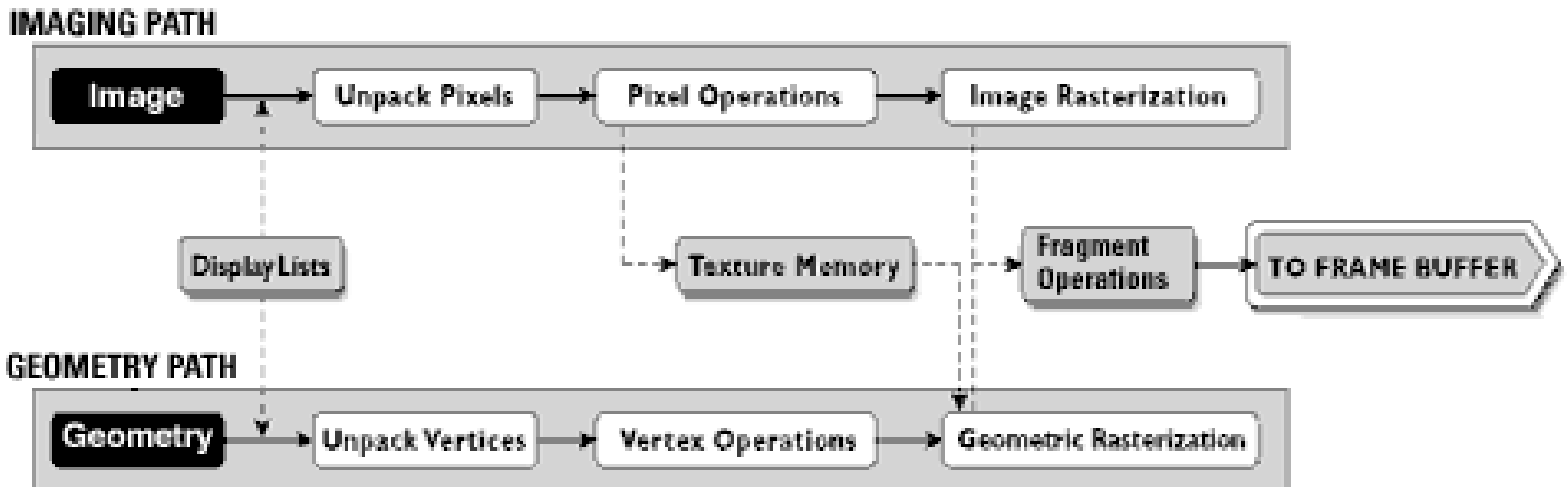
“OpenGL is the premier environment for developing portable, interactive 2D and 3D graphics applications. Since its introduction in 1992, OpenGL has become the industry’s most widely used and supported 2D and 3D graphics application programming interface (API), bringing thousands of applications to a wide variety of computer platforms. OpenGL fosters innovation and speeds application development by incorporating a broad set of rendering, texture mapping, special effects, and other powerful visualization functions. Developers can leverage the power of OpenGL across all popular desktop and workstation platforms, ensuring wide application deployment.”

`www.opengl.org`

OpenGL: What can it do?

- Imaging part: works on pixels, bitmaps
- Geometry part: works on vertices, polygons
- uses a rendering pipeline that starts from data and ends with a display device.

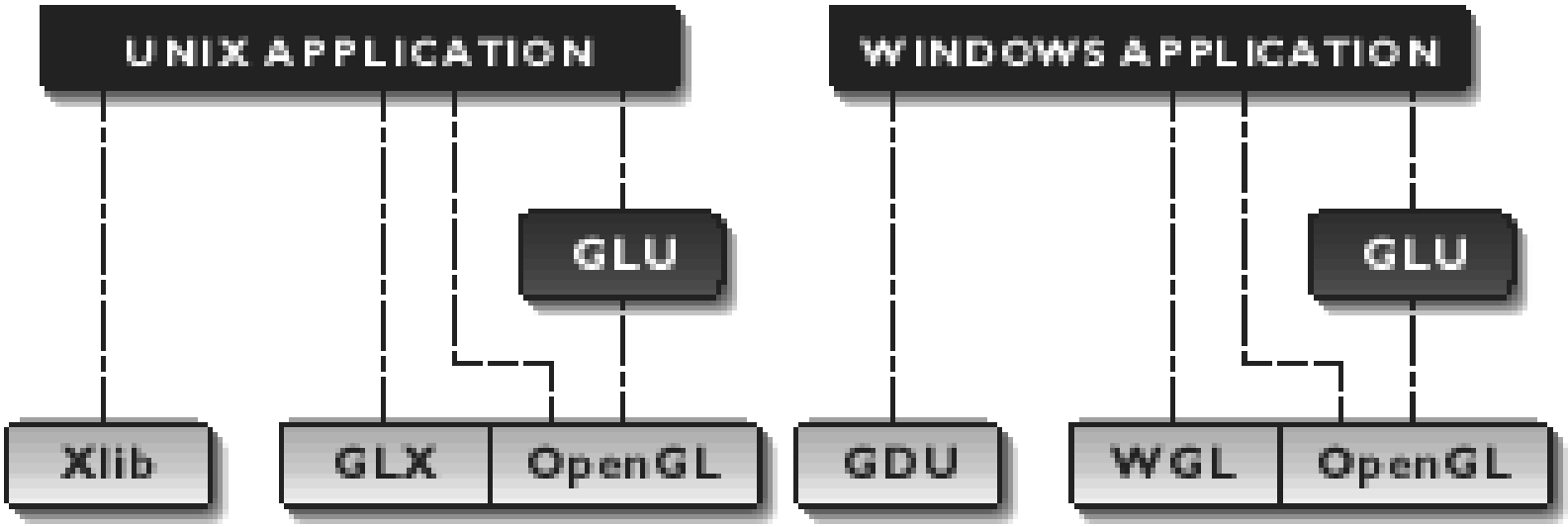
OpenGL rendering pipeline



OpenGL: More info

- Application Program Interface based on C-style function calls
- industry standard: one of several (Java3D, DirectX are others)
- stable, reliable and portable
- scalable: PDA to supercomputer
- well documented and easy to use

OpenGL on Windows and Unix



- GLU: OpenGL-Extension for complex polygons, curves etc.

OpenGL compiling and linking

```
#include <GL/glut.h>
#include <GL/glu.h>
#include <GL/gl.h>
```

- `gcc -lgl -lglu -lglut -o program program.c`
- In your OS, capitalization of GL, GLU and GLUT might be different!

The structure of an OpenGL application

```
int main(int argc, char** argv)
{
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)
        glutInitWindowSize(640,480);
        glutInitWindowPosition(100, 150);
        glutCreateWindow("my first attempt");
        glutDisplayFunc(myDisplay);
        myInit();
        glutMainLoop();
        return 0;
}
```

Other Callback Functions

```
...
    glutDisplayFunc(myDisplay);
    glutReshapeFunc(myReshape);
    glutMouseFunc(myMouse);
    glutKeyboardFunc(myKeyboard);
...
```

Draw three points

```
void myDisplay(void)
{
        glClear(GL_COLOR_BUFFER_BIT);
        glBegin(GL_POINTS);
                glVertex2i(100, 50);
                glVertex2i(100, 130);
                glVertex2i(150, 130);
        glEnd();
        glFlush();
}
```

OpenGL Functions

`glVertex2i()`

- `gl` is the prefix of all OpenGL function names
- `Vertex` is a function name
- `2i` describes the arguments: two integers

OpenGL Datatypes

GLenum,GLboolean,GLbitfield	unsigned datatypes
GLvoid	pseudo datatype for pointers and return values
GLbyte,GLshort,GLint	1,2,4-byte signed
GLubyte,GLushort,GLuint	1,2,4-byte unsigned
GLsizei	4-byte signed size datatype

OpenGL Datatypes

GLfloat	single precision float
GLclampf	single precision float in [0,1]
GLdouble	double precision float
GLclampd	double precision float in [0,1]

Drawing Dots

```
glBegin(GL_POINTS);
        glVertex2i(100, 50);
        glVertex2i(100, 130);
        glVertex2i(150, 130);
glEnd();
```

Drawing a line

```
glBegin(GL_LINES);
        glVertex2i(100, 50);
        glVertex2i(100, 130);
glEnd();
```

Drawing two lines

```
glBegin(GL_LINES);
        glVertex2i(10, 20);
        glVertex2i(40, 20);
        glVertex2i(20, 10);
        glVertex2i(20, 40);
glEnd();
```

Drawing a polyline

```
glBegin(GL_LINE_STRIP);
        glVertex2i(10, 20);
        glVertex2i(40, 20);
        glVertex2i(20, 10);
        glVertex2i(20, 40);
glEnd();
```

Drawing a polygon

```
glBegin(GL_LINE_LOOP);
        glVertex2i(10, 20);
        glVertex2i(40, 20);
        glVertex2i(20, 10);
        glVertex2i(20, 40);
glEnd();
```

Drawing an aligned rectangle

```
glRecti(x1,y1,x2,y2);
```

What are those numbers?

- There is no predefined way of interpreting the coordinates
- OpenGL can work with different coordinate systems
- For OpenGL, we have to define a coordinate system to be used

Colors and a Coordinate System

```
void myInit(void)
{
    glClearColor(1.0,1.0,1.0,0.0);
    glColor3f(0.0f, 0.0f, 0.0f);
    glPointSize(4.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, 640.0, 0.0, 480.0);
}
```

Algorithmic Drawing

```
void Sierpinski(void){
    GLintPoint T[3]= {{10,10},{300,30},{200, 300}};
    int index = random(3);
    GLintPoint point = T[index];
    drawDot(point.x, point.y);
    for(int i = 0; i < 4000; i++) {
        index = random(3);
        point.x = (point.x + T[index].x) / 2;
        point.y = (point.y + T[index].y) / 2;
        drawDot(point.x,point.y);
    }
    glFlush();
}
```

Coordinate System

- For now, we have used a simple coordinate system:
  $x : 0 \ldots \text{ScreenWidth} - 1, y = 0 \ldots \text{ScreenHeight} - 1$
- In case ScreenWidth or ScreenHeight change, glut can inform us via the `glutReshapeFunc(myReshape);`
- We can manually apply a *coordinate transformation* in order to display arbitrary coordinate systems.
- Or we can have OpenGL do this for us

Some terms

- The space in which objects are described uses *world coordinates*.
- The part of this space that we want to display is called *world window*.
- The window that we see on the screen is our *viewport*.
- In order to know where to draw something, we need the *world-to-viewport transformation*
- Note that these terms can be used both for 2D and for 3D.

A simple example

$$
\begin{aligned}
sx &= Ax + C \\
sy &= By + D \\
A &= \frac{V.r - V.l}{W.r - W.l} \\
C &= V.l - AW.l \\
B &= \frac{V.t - V.b}{W.t - W.b} \\
D &= V.b - bW.b
\end{aligned}
$$

In OpenGL

```
void setWindow(float left, float right,
               float bottom, float top)
{
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(left, right, bottom,top);
}
void setViewport(int left, int right,
                 int bottom, int top)
{
  glViewport(left,bottom,right-left,top-bottom);
}
```

Clipping

- What happens to parts of the “world” that are outside of the world window?
  Answer: They are not drawn.
- How to identify the parts of the world that are to be drawn?
- Clipping Lines: identifying the segment of a line to be drawn
- Input: the endpoints of a line and a world window
- Output: the new endpoints of the line (if anything is to be drawn)

Clipping

- First step: Testing for trivial accept or reject
- Cohen Sutherland Clipping Algorithm
- For each point do four tests, compute 4 bit word:
  1. Is P to the left of the world window?
  2. Is P above the top of the world window?
  3. Is P to the right of the world window?
  4. Is P below the bottom of the world window?

Cohen Sutherland

- Compute tests for both points of the line
- Trivial Accept: all tests false, all bits 0
- Trivial Reject: the words for both points have 1s in the same position
- Deal with the rest: neither trivial accept nor reject

The rest

- Identify which point is outside and to which side of the window
- Find the point where the line touches the world window border
- Move the outer point to the border of the window
- repeat all until trivial accept or reject

ClipSegment($p1, p2, W$)

```
1:  while (TRUE) do
2:      if (trivial accept) then
3:          RETURN 1
4:      end if
5:      if (trivial reject) then
6:          RETURN 0
7:      end if
8:      if (p1 is outside) then
9:          if (p1 is to the left) then
10:             chop against the left edge of W
11:         else
12:             if (p1 is to the right) then
13:                 chop against the right edge of W
14:             else
15:                 if (. . . ) then
16:                     . . .
17:                 end if
18:             end if
19:         end if
20:     end if
21: end while
```

Relative drawing

- It is often convenient to draw figures relative to a current pen position
- Idea: maintain the current position (CP) a static global variable
- use two functions MOVEREL and LINEREL to move/draw relative to CP
- implementation is obvious. (or can be found in the book on page 105)

Application of relative drawing

- Turtle graphics: originally from the logo programming language
  - logo has been invented at MIT to teach children how to program. try google for more info
- Simple primitives: TURNTO (absolute angle) TURN (relative angle) FORWARD (distance, isVisible)
- Implementation obvious: maintain additional current direction (CD) in a static global variable, use simple (sin, cos) trigonometry functions for FORWARD.

Application of relative drawing: n-gons

- The vertices of an n-gon lie on a circle
- divide the circle into n equal parts
- connect the endpoints of the parts on the circle with lines
- using relative drawing, this is very easy to implement
- by connecting every endpoint to every other endpoint, a rosette can be drawn

relative hexagon

```
for (i=0;i<6;i++)
{
  forward(L,1);
  turn(60);
}
```

Circles and Arcs

- Circles can be approximated with n-gons (with a high n)
- Arcs are partially drawn circles, instead of dividing the circle, divide the arc

Representing curves

- Two principle ways of describing a curve: implicitly and parametrically
- Implicitly: Give a function F so that $F(x, y) = 0$ for all points of the curve
- Example: $F(x, y) = (y - A_y)(B_x - A_x) - (x - A_x)(B_y - A_y)$ (a line)
- Example: $F(x, y) = x^2 + y^2 - R^2$ (a circle)

Implicit form of curves

- The implicit form is good for testing if a point is on a curve.
- For some cases, we can use the implicit form to define an "inside" and an "outside" of a curve: $F(x,y) < 0 \rightarrow$ inside, $F(x,y) > 0 \rightarrow$ outside
- some curves are *single valued* in x: $F(x,y) = y - g(x)$ or in y:$F(x,y) = x - h(y)$
- some curves are neiter, e.g. the circle needs two functions $y = \sqrt{R^2 - x^2}$ and $y = -\sqrt{R^2 - x^2}$

Parametric form of curves

- The parametric form of a curve suggests the movement of a point through time.
- Example: $x(t) = A_x + (B_x - A_x)t, y(t) = A_y + (B_y - A_y)t, t \in [0, 1]$
- Example: $x(t) = W\cos(t)$, $y(t) = H\sin(t), t \in [0, 2\pi]$
- In order to find an implicit form from a parametric form, we can use the two $x(t)$ and $y(t)$ equations to eliminate t and find a relationship that holds true for all t.
- For the Ellipse: $\left(\frac{x}{W}\right)^2 + \left(\frac{y}{H}\right)^2 = 1$

Drawing parametric curves

- In order to draw a parametric curve, we have to approximate it.
- In order to do that, we chose some values of t and sample the functions x and y at t_i.
- One option is to approximate the function in between with line segments.

```
glBegin(GL_LINES);
for  (i=0;i<n;i++)
     glVertex2f(x(t[i]),y(t[i]));
glEnd();
```

Superellipses

- A *superellipse* is defined by the implicit form $\left(\frac{x}{W}\right)^n + \left(\frac{y}{H}\right)^n = 1$
- A *supercircle* is a superellipse with $W = H$.
- $x(t) = W \cos(t) |\cos(t)^{2/n-1}|$
- $y(t) = H \sin(t) |\sin(t)^{2/n-1}|$