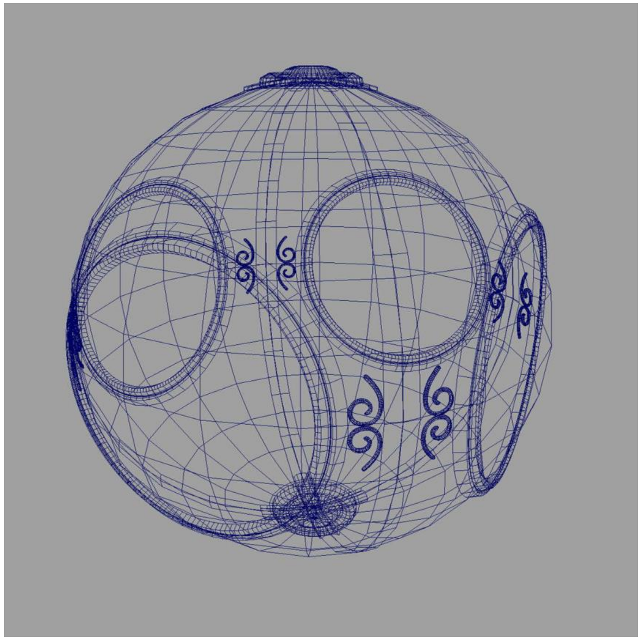
The University of New Mexico

Participation

- Participation my preference send a simple email subject includes you name, Class (GUI); date
- Please catch up on all history
- Missed one no panic send an email later
- Synchronous Classes on M, T, W,
- R is asynchronous study at home (mainly on Unity) “proof” of studying via a submission (screen shots, video clip, final product of a tutorial)

Graphics Primitives

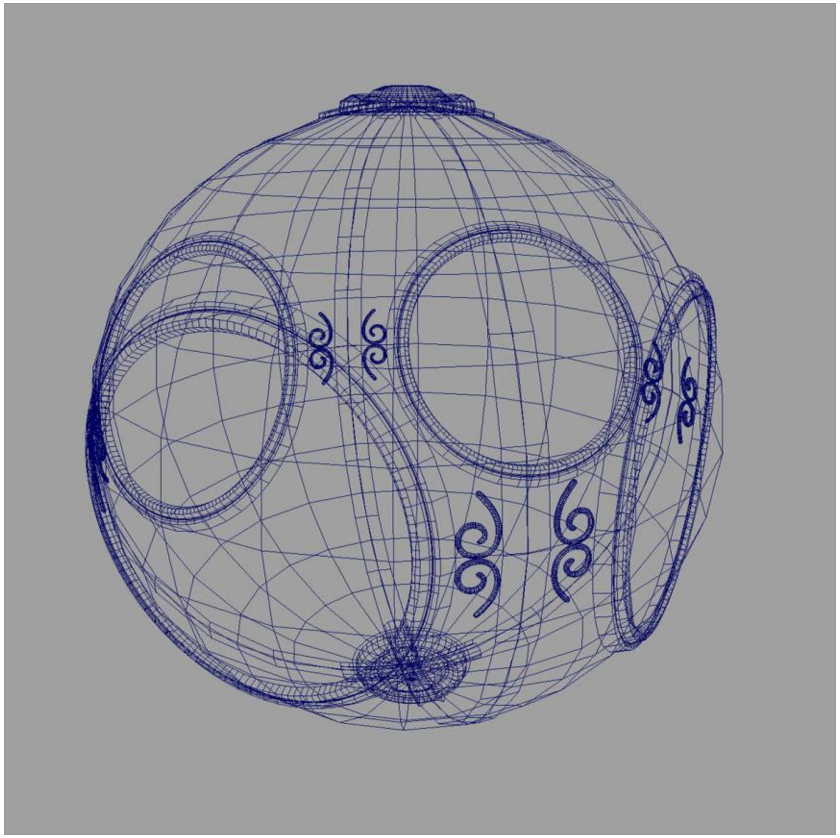
- *Points (point cloud)*
- *Points and lines*
  - *Used in Laser Graphics*
- *Geometry Primitives (OGL)*
  - *Vertex (point)*
  - *Line (edge)*
  - *Polygon*
- *Raster Graphics (OGL)*
  - *Digital Image*



Computer Graphics: 1960-1970

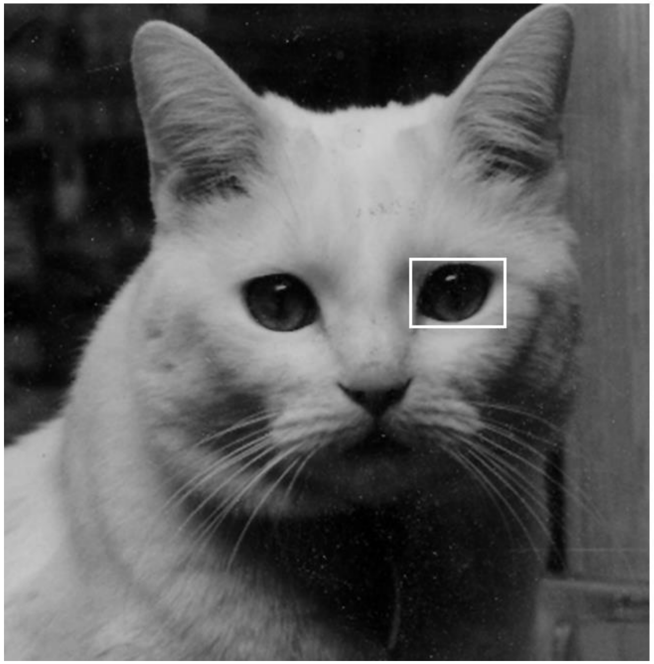
- *Wireframe* graphics
  - Draw only lines
- Sketchpad
- Display Processors
- Storage tube

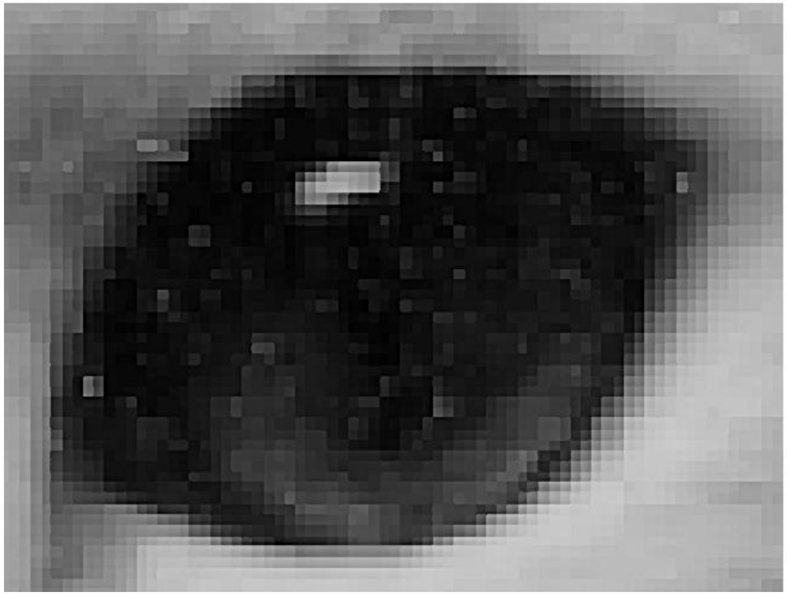
wireframe representation of sun object



Raster Graphics

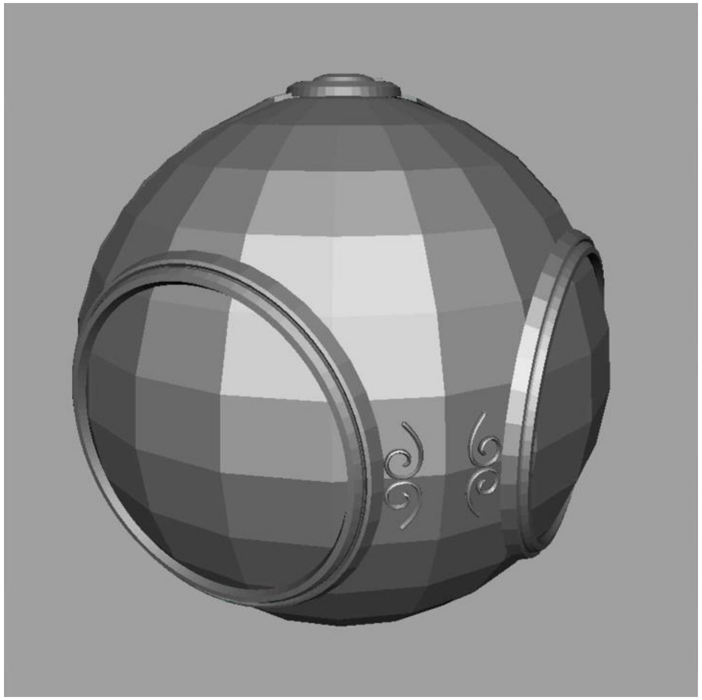
- Image produced as an array (the *raster*) of picture elements (*pixels*) in the *frame buffer*





Raster Graphics

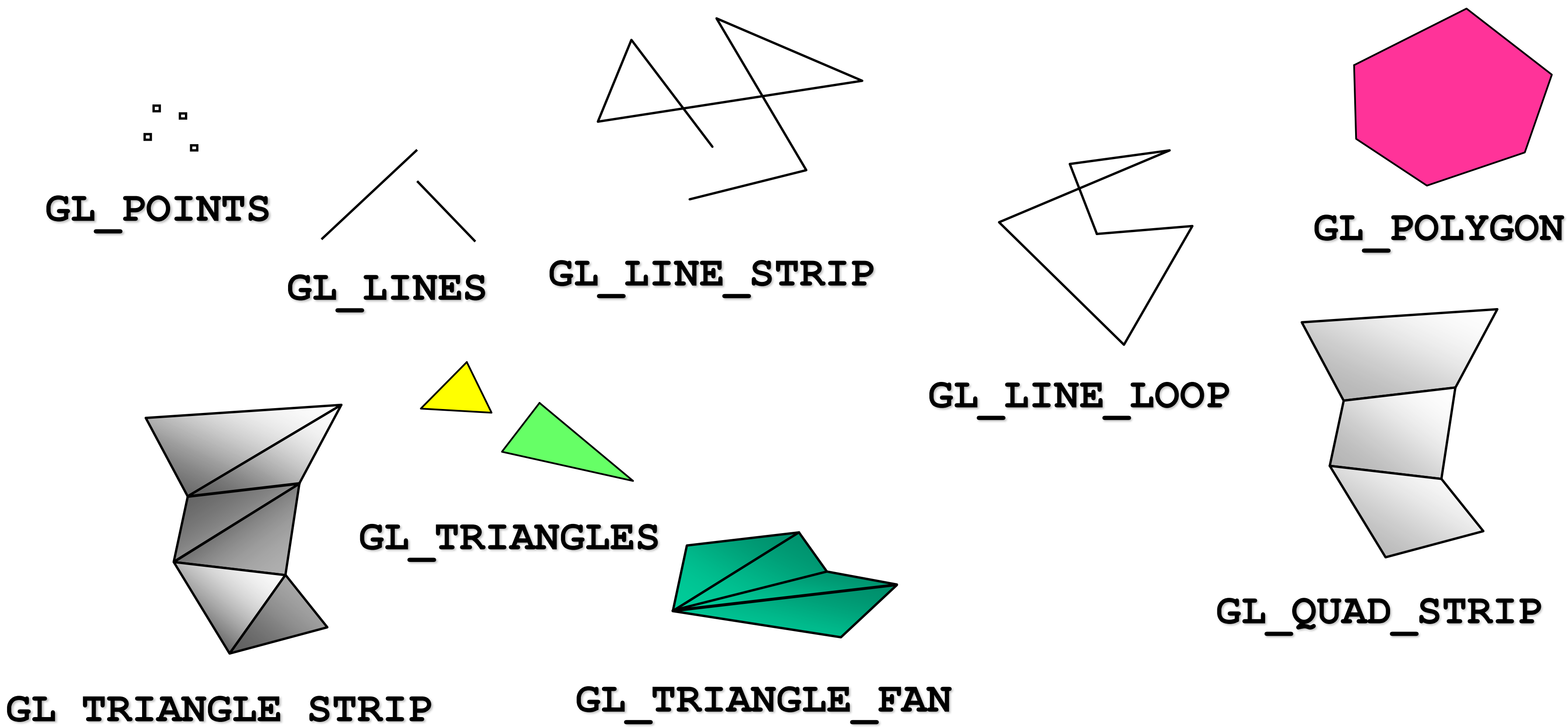
- Allows us to go from lines and wire frame images to filled polygons



The University of New Mexico

OpenGL (2.x) Geometry Primitives

Used in QT4; DT; OGL 4+ Unity; WebGL; QT5



The University of New Mexico

OpenGL function format

function name

dimensions

`glVertex3f(x,y,z)`

`x,y,z` are `floats`

gl - belongs to GL library
gl – gl function (in QT too)
glu – GL Utility not in QT (may need to include for placing camera QT)
glut – Rudimentary OGL GUI (replaced by QT GUI)

`glVertex3fv(p)`

`p` is a pointer to an array

OpenGL #defines

- Most constants are defined in the include files **`gl.h`**, **`glu.h`** and **`glut.h`**
  - Note **`#include <GL/glut.h>`** should automatically include the others
  - Examples
  - **`glBegin(GL_POLYGON)`**
  - **`glClear(GL_COLOR_BUFFER_BIT)`**
- include files also define OpenGL data types: **`GLfloat`**, **`GLdouble`**,….

Object Definition Example

The University of New Mexico

type of object

location of vertex

```
glBegin(GL_POLYGON could USE GL_TRIANGLE)
 glVertex3f(0.0, 0.0, 0.0);
 glVertex3f(0.0, 1.0, 0.0);
 glVertex3f(0.0, 0.0, 1.0);
glEnd( );
```

end of object definition

OGL 3+ only has pints, lines, and triangles (tiling)
This example is with static vertices

Static Time vs. Dynamic time

C program

Compile

Assembly

Machine Code

Link (takes modules from different sources program library)

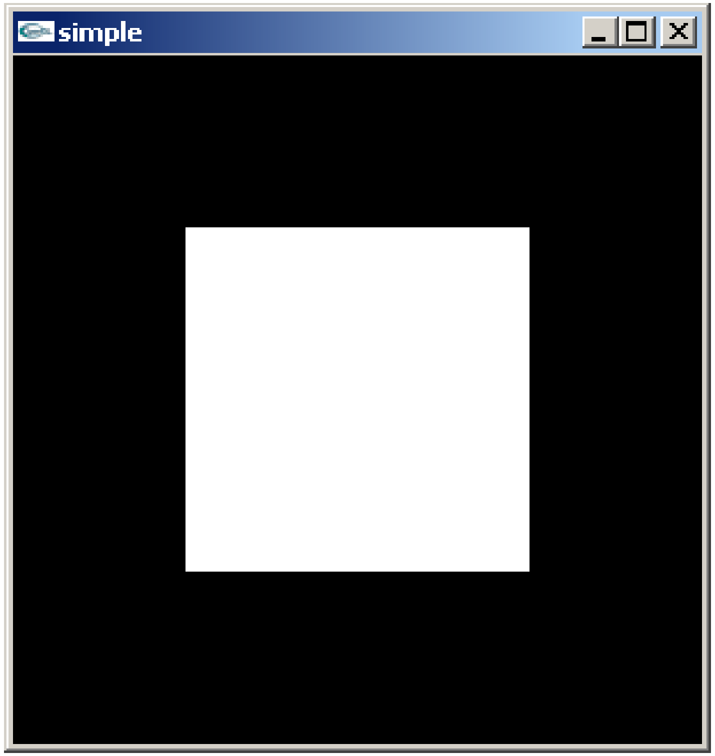
Loader Loads program to memory

Static

Dynamic time program is executed (run time)

A Simple Program

Generate a square on a solid background



simple.c

```
#include <GL/glut.h>
void mydisplay(){
     glClear(GL_COLOR_BUFFER_BIT);
      glBegin(GL_POLYGON);
            glVertex2f(-0.5, -0.5);
            glVertex2f(-0.5, 0.5);
            glVertex2f(0.5, 0.5);
            glVertex2f(0.5, -0.5);
      glEnd();
      glFlush(); // Send the frame buffer (raster storage of the graphic
to the defined screen/window/port)
}
int main(int argc, char** argv){
      glutCreateWindow("simple");
      glutDisplayFunc(mydisplay);  // call back
      glutMainLoop(); // Waiting for events
}
```

Event Loop

- Note that the program defines a *display callback* function named **`mydisplay`**
  - Every glut program must have a display callback
  - The display callback is executed whenever OpenGL decides the display must be refreshed, for example when the window is opened
  - The **`main`** function ends with the program entering an event loop

Defaults and Parameters

(are like states in a state machine)

- `simple.c` is too simple
- Makes heavy use of state variable default values for
  - Viewing
  - Colors (parameter state)
  - Window parameters
- Next version will make the defaults more explicit

Notes on compilation

- No need we use QT
- See website and ftp for examples
- Unix/linux You can use the Class
    - Include files usually in …/include/GL
    - Compile with –lglut –lglu –lgl loader flags
    - May have to add –L flag for X libraries
    - Mesa implementation included with most linux distributions
    - Check web for latest versions of Mesa and glut

Objectives

- Refine the first program
  - Alter the default values
  - Introduce a standard program structure
- Simple viewing
  - Two-dimensional viewing as a special case of three-dimensional viewing
- Fundamental OpenGL primitives
- Attributes

Program Structure

- Most OpenGL programs have a similar structure that consists of the following functions
  - **`main()`**:
    - defines the callback functions
    - opens one or more windows with the required properties
    - enters event loop (last executable statement)
  - **`init()`**: sets the state variables
    - Viewing
    - Attributes; colors
  - Callbacks We use QT
    - Display function
    - Input and window functions

simple.c revisited

- In this version, we shall see the same output but we have defined all the relevant state values through function calls using the default values
- In particular, we set
  - Colors
  - Viewing conditions
  - Window properties

The University of New Mexico

main.c

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

int main(int argc, char** argv)
{
 glutInit(&argc,argv);
 glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB); double
 for animation
 glutInitWindowSize(500,500);
 glutInitWindowPosition(0,0);
 glutCreateWindow("simple");
 glutDisplayFunc(mydisplay);

 init();

 glutMainLoop();
}
```

includes `gl.h`

define window properties

display callback

set OpenGL state

enter event loop

GLUT functions

- **`glutInit`** allows application to get command line arguments and initializes system
- **`gluInitDisplayMode`** requests properties for the window (the *rendering context*)
  - RGB color
  - Single buffering
  - Properties logically ORed together
- **`glutWindowSize`** in pixels
- **`glutWindowPosition`** from top-left corner of display
- **`glutCreateWindow`** create window with title “simple”
- **`glutDisplayFunc`** display callback
- **`glutMainLoop`** enter infinite event loop

init.c (one time)

```
void init()
{
 glClearColor (0.0, 0.0, 0.0, 1.0);
(R,G,B,O)  O - Pecity Color is between  [0..1]
 [0..255] (least intensity.. most intensity)

 glColor3f(1.0, 1.0, 1.0);

 glMatrixMode (GL_PROJECTION);
# Define a parallel projection camera
 glLoadIdentity ();  // initiate
 glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
glortho (L, R, B, T, N, F)
}
```

Transformations and Viewing

- In OpenGL, projection is carried out by a projection matrix (transformation)
- There is only one set of transformation functions so we must set the matrix mode first

```
glMatrixMode (GL_PROJECTION)
```

- Transformation functions are incremental so we start with an identity matrix and alter it with a projection matrix that gives the view volume

```
glLoadIdentity();
glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
```

Two- and three-dimensional viewing

- In **`glOrtho(left, right, bottom, top, near, far)`** the near and far distances are measured <u>from</u> the camera
- Two-dimensional vertex commands place all vertices in the plane z=0
- If the application is in two dimensions, we can use the function

  `gluOrtho2D(left, right,bottom,top)`
- In two dimensions, the view or clipping volume becomes a *clipping window*

mydisplay.c

```
void mydisplay()
{
 glClear(GL_COLOR_BUFFER_BIT);
 glBegin(GL_POLYGON);
     glVertex2f(-0.5, -0.5);
     glVertex2f(-0.5, 0.5);
     glVertex2f(0.5, 0.5);
     glVertex2f(0.5, -0.5);
 glEnd();
 glFlush();
}
```

Dynamic Example

```
Use variables
1) Obtained from UI (command line; mouse, KBD, widgets,
   menu)
2) From data structures including files
3) Generated by a program
    1) Loop
    2) Recursion
```

```
glBegin(GL_POLYGON could USE GL_TRIANGLE)
 glVertex3f(x1, y1, z1);
 glVertex3f(x2, y2, z2);
 glVertex3f(x3, y3, z3);
glEnd( );
```

Plot a Line

End points x0,y0)(x1,y1) are given. Plot the line between the end points. Given the end points, we can represent the line y = mx + b

```
float x, y, x1, y1, x0, y0, dx, m, b;
dx = 0.001;
 glBegin(GL_POINTS)
      for (x=x0,x<= x1, x = x+ dx){
      y= m*x + b
      glVertex2f(x, y);
      }
glEnd( );
```

Plot a Circle

The University of New Mexico

Circle of radius 1 with center at (0, 0)

y = mx + b is referred to as the explicit function of line

$x^2 + y^2 - r^2 = 0$ implicit circle

$x = +\mathrm{sqrt}(r^2 - y^2)$ explicit circle

```
float x, y, x1, y1, x0, y0, dx, r;
dx = 0.0001;
 glBegin(GL_POINTS)
      for (x=-r,x<= r, x=x+dx){
      use C for "x= +sqrt(r2 - y2)"
      glVertex2f(x, y); }
glEnd( );
```

Plot a 1 Variable function

Obtain an explicit representation y=f(x)

Will be posted on TRACS Resources class notes

```
float x, y, x_min, x_max, dx;
dx = 0.0001;
 glBegin(GL_LINESTRIP)
     for (x=x_min,x<= x_max,x = x + dx){
     use C for y = f(x);
     glVertex2f(x, y);
     }
glEnd( );
```

Functions Forms

1. Explicit
2. Implicit
3. Parametric
4. Approximating, Interpolating Curves

Functions Forms

1. Explicit
    1. $y = f(x)$
    2. $z = f(x, y)$
2. Implicit
    1. $f(x, y) = 0 \quad x^2 + y^2 - r^2 = 0$
    2. $F(x, y, z) = 0$
3. Parametric
4. Approximating, Interpolating Curves Surfaces

Functions Forms

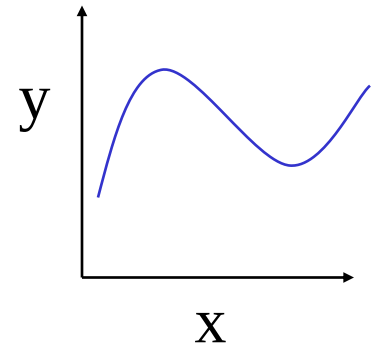
3. Parametric
    1. <p(u)> = <x(u), y(u)>
    2. <p(u, v)> = <x(u, v), y(u, v), z(u, v)>

4. Approximating, Interpolating Curves Surfaces

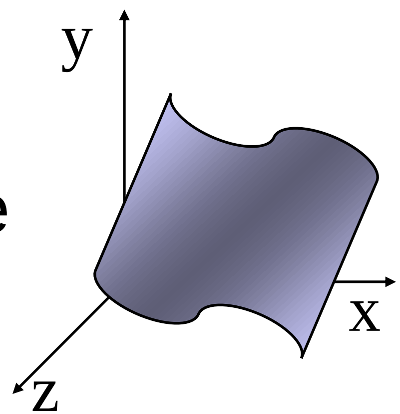
Explicit Representation

- Most familiar form of curve in 2D

$$y=f(x)$$



- Cannot represent all curves
  - Vertical lines
  - Circles
- Extension to 3D
  - $y=f(x)$, $z=g(x)$
  - The form $z = f(x, y)$ defines a surface

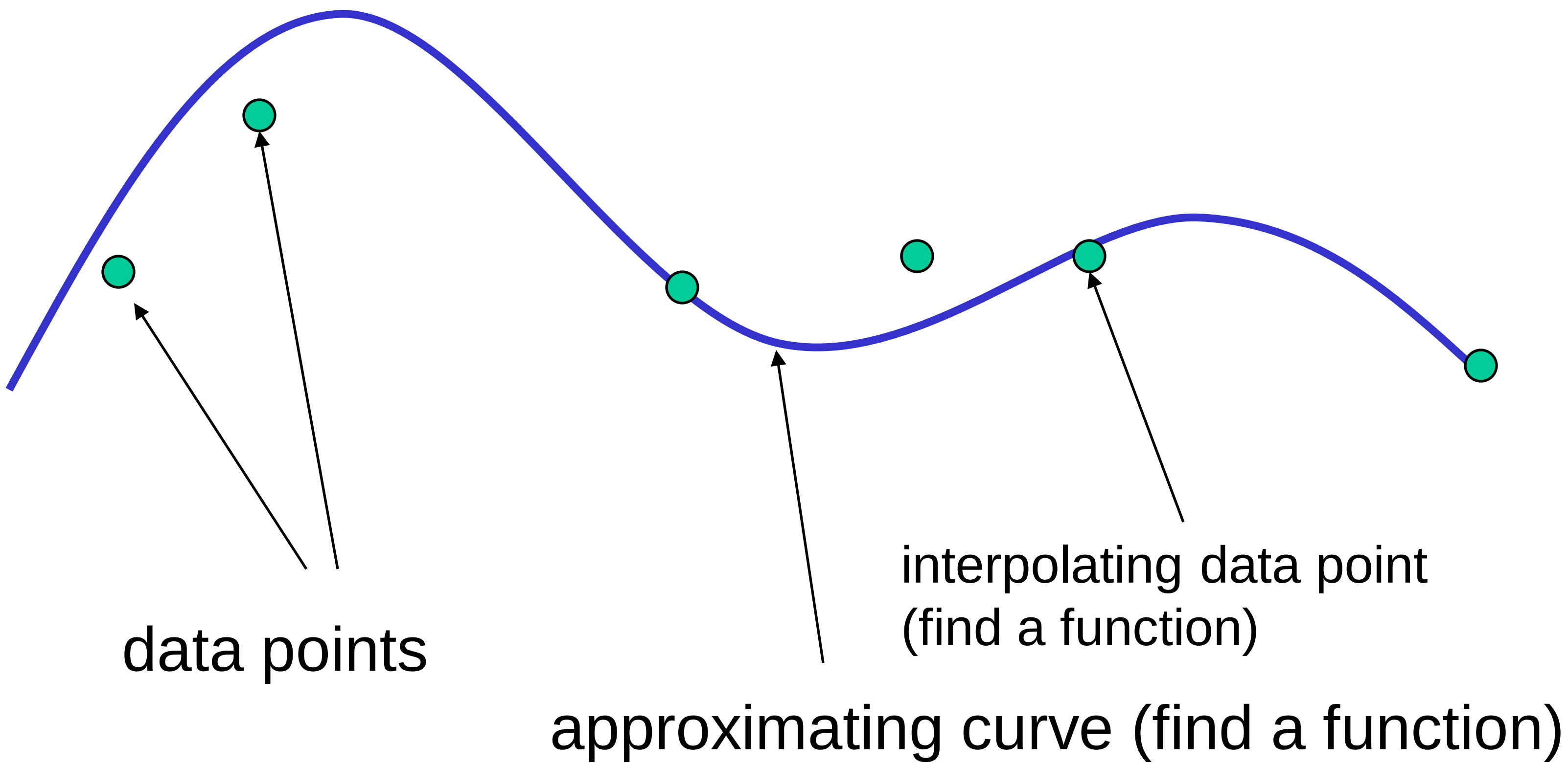


Parametric Form

- This form is known as the parametric form of the line
  - More robust and general than other forms
  - Extends to curves and surfaces
- Two-dimensional forms
  - Explicit: $y = mx + h$
  - Implicit: $ax + by + c = 0$
  - Parametric:

$$x(\alpha) = \alpha x_0 + (1-\alpha)x_1$$
$$y(\alpha) = \alpha y_0 + (1-\alpha)y_1$$

Modeling with Curves



Implicit Representation

- Twodimensional curve(s)

$$g(x, y)=0$$

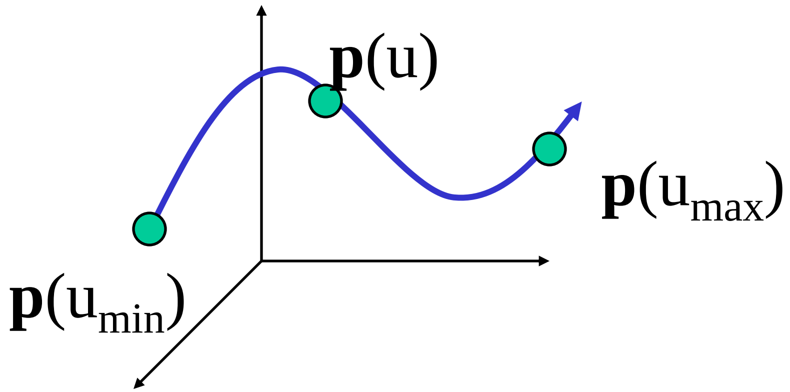
- Much more robust
  - All lines $ax+by+c=0$
  - Circles $x^2+y^2-r^2=0$
- Three dimensions $g(x,y,z)=0$ defines a surface
  - Intersect two surface to get a curve
- In general, we cannot solve for points that satisfy

Parametric Curves

- Separate equation for each spatial variable

$$x=x(u)$$
$$y=y(u) \qquad\qquad \mathbf{p}(u)=[x(u), y(u), z(u)]^T$$
$$z=z(u)$$

- For $u_{max} \geq u \geq u_{min}$ we trace out a curve in two or three dimensions



Plotting z = f(x, y)

```
Plot z = f(x, y) = ax + by
x_min<=x<=x_max  y_min<=y<=y_max  a and b are given
 constants.
```

```
for (double x=<x_min; x<=<x_max; x+=dx)
   {
      glBegin(GL_QUADS);
      for (double y=<y_min; y<=<y_max ; y+=dy)
      {
          Use C++ (math.h) to calculate z = ax +by
            glVertex3f(x, y, z);
      }
      glEnd();
   }
```

Plotting z = f(x, y)

Plot z = f(x, y) = sin(x)/x * sin(y)/y x, y are in [-8*pi, 8*pi] line-strip (or points) with dx, dy 0.01
x_min<=x<=x_max y_min<=y<=y_max constants.

```
Example
for (double x=<x_min; x<=<x_max; x+=dx)
    {
       glBegin(GL_QUADS);
       for (double y=<y_min; y<=<y_max ; y+=dy)
       {
            Use C++ (math.h) to calculate z
              glVertex3f(x, y, z);
       }
       glEnd();
    }
```

Plotting z = f(x, y) alternative

```
Plot z = f(x, y) = sin(x)/x * sin(y)/y  x, y are in
[-8*pi, 8*pi] line-strip (or points) with dx, dy 0.01
x_min<=x<=x_max  y_min<=y<=y_max  constants.
```

Plotting <p(u)>=<x(u), y(u)>

Plot a circle using the parametric

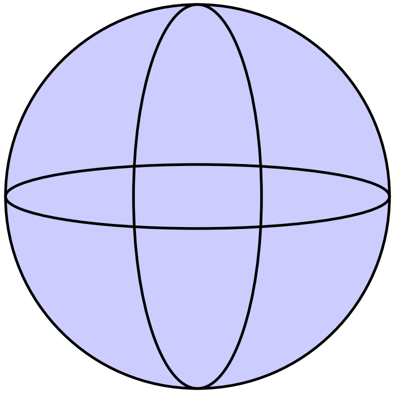
$$x(u) = r \cos u \qquad 360 \geq u \geq 0$$
$$y(u) = r \sin u$$

```
float x, y, u_min, u_max, du;
du = 0.01;
      glBegin(GL_LINE_STRIP)
            for (u=u_min,u<= u_max,u += du){
   use C++ functions (math.h)
            for x(u)=r* cos(u),y(u)=sin(u);
            glVertex2f(x, y);
            }
glEnd( );
```

Parametric Sphere

$$x(\theta, \phi) = r \cos\theta \sin\phi$$
$$y(\theta, \phi) = r \sin\theta \sin\phi$$
$$z(\theta, \phi) = r \cos\phi$$

$$360 \geq \theta \geq 0$$
$$180 \geq \phi \geq 0$$



θ constant: circles of constant longitude

ϕ constant: circles of constant latitude

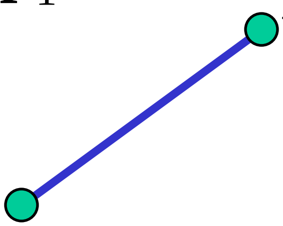
Plotting $\langle p(u, v)\rangle = \langle x(u, v), y(u, v), z(u, v)\rangle$

Parametric Lines

We can normalize u to be over the interval (0,1)

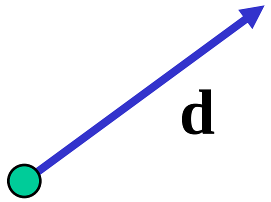
Line connecting two points $\mathbf{p}_0$ and $\mathbf{p}_1$

$$\mathbf{p}(u)=(1-u)\mathbf{p}_0+u\mathbf{p}_1$$



Ray from $\mathbf{p}_0$ in the direction $\mathbf{d}$

$$\mathbf{p}(u)=\mathbf{p}_0+u\mathbf{d}$$



The University of New Mexico

init.c

```
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
```

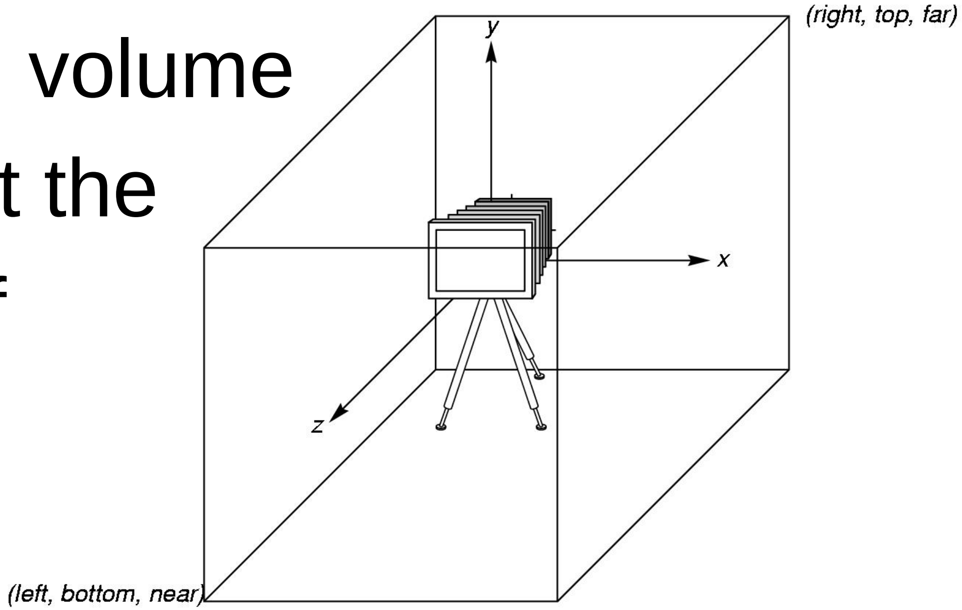
Standard view volume and default view volume

Coordinate Systems

- The units in **`glVertex`** are determined by the application and are called *object* or *problem coordinates* *Must be in the 3-D View volume to be visible*
- The viewing specifications are also in object coordinates and it is the size of the viewing volume that determines what will appear in the image
- Internally, OpenGL will convert to *camera (eye) coordinates* and later to *screen coordinates*
- OpenGL also uses some internal representations that usually are not visible to the application

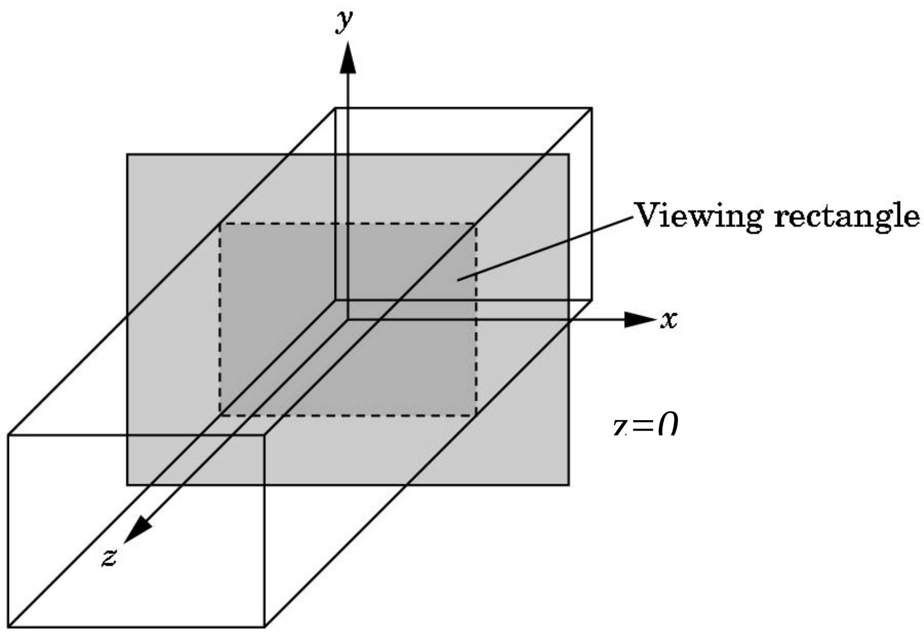
OpenGL Camera

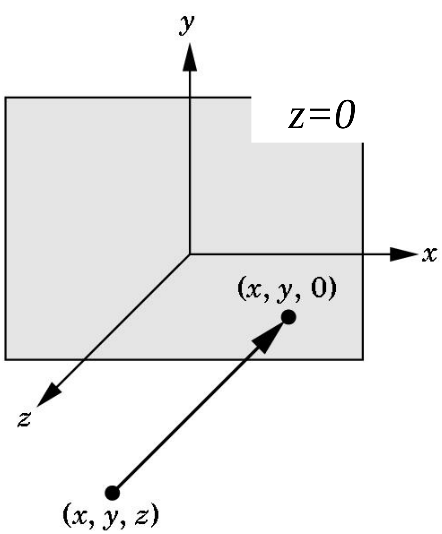
- OpenGL places a camera at the origin in object space pointing in the negative z direction
- The default viewing volume is a box centered at the origin with a side of length 2



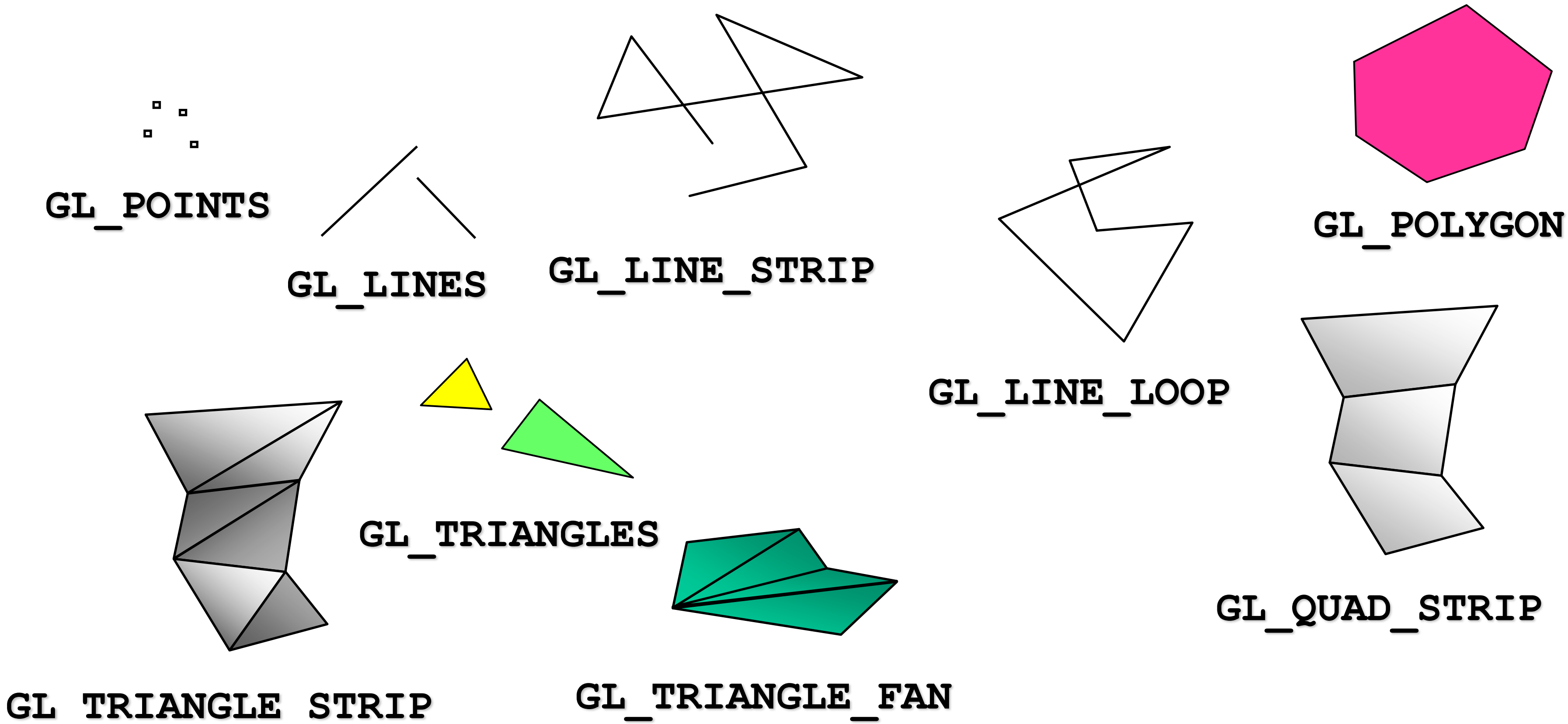
Orthographic Viewing

In the default orthographic view, points are projected forward along the *z* axis onto the plane *z=0*



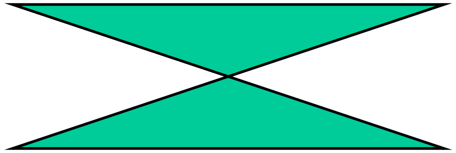


OpenGL Primitives

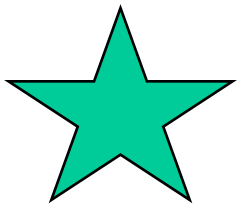


Polygon Issues

- OpenGL will only display polygons correctly that are
  - Simple: edges cannot cross
  - Convex: All points on line segment between two points in a polygon are also in the polygon
  - Flat: all vertices are in the same plane
- User program can check if above true
  - OpenGL will produce output if these conditions are violated but it may not be what is desired
- Triangles satisfy all conditions



nonsimple polygon



nonconvex polygon

The University of New Mexico

Attributes

- Attributes are part of the OpenGL state and determine the appearance of objects
  - Color (points, lines, polygons)
  - Size and width (points, lines)
  - Stipple pattern (lines, polygons)
  - Polygon mode
    - Display as filled: solid color or stipple pattern
    - Display edges
    - Display vertices