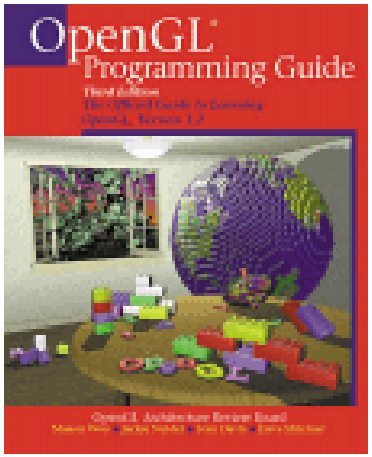
4. Introduction to OpenGL

- 4.1 Resources
- 4.2 Background
- 4.3 GL/GLU/GLUT etc.
- 4.4 A Simple OpenGL Program
  - What the Program Does
  - The Code
  - Aspects of the Code
- 4.5 Geometric Primitives in OpenGL

4.1 Resources

- "OpenGL Programming Guide: The Official Guide to Learning OpenGL", Woo, Neider, and Davis, Addison-Wesley (aka "The Red Book").
  - 1st edition online: http://www.glprogramming.com/red
- OpenGL/GLUT manuals
  - See *COMP 372 Resources* page



Resources (cont'd)

- OpenGL homepage: http://www.opengl.org/
  - Examples, Discussion forums, etc.
- OpenGL Examples (see *372 Resources* page)
  - Consists of one solution (.sol) with one project for each example
  - Contains all major examples from the OpenGL Programming Guide.
  - In order to run an example open the solution, choose an active project, compile and execute it (try `fog`, `teapots`, `material`, `dof`, `aapoly`).
    - The examples `fogindex` and `aaindex` require 256 colour mode.
  - Read comments at the beginning of each source files (you are not expected to understand them, yet).

4.2 Background

- SGI (Silicon Graphics Inc.) devised a proprietary graphics language *IrisGL*
- *IrisGL* is a software interface to polygon-rendering hardware and consists of a library of *C* functions to:
  - define geometric objects in 2D and 3D.
  - Control how these objects are rendered in the frame buffer (set up view transformations, perspective transformations, illumination).
  - Output textured Gouraud-shaded polygons (with z-buffering), plus lines and points.

Background (cont'd)

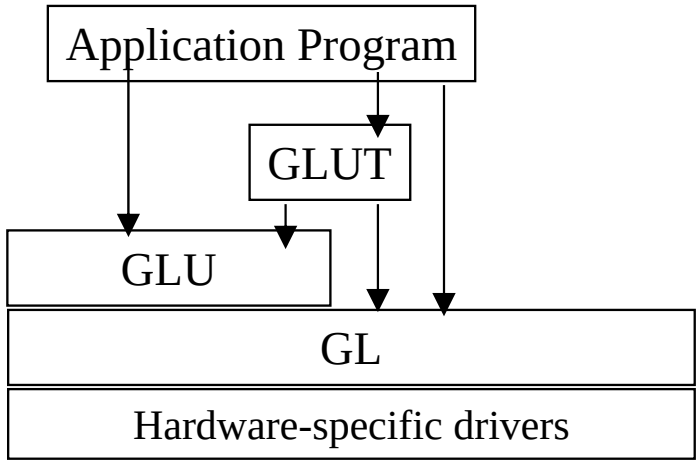
- In 1992, SGI made the spec. publicly available, calling it *OpenGL*
  - Widely adopted by other graphics companies
  - Specification maintained and expanded by the OpenGL Architectural Review Board (ARB)
  - Supported by most PC graphics card manufacturers
  - Standard for most Unix workstations, adopted by Apple for the Mac
  - Incorporated into Windows 9X/NT/2000
    - Competes with Direct-3D component of Direct-X
- In 2004, OpenGL 2.0 was released including a number of major additions to the standard. The most significant one is GLSL (the Open**GL** **S**hading **L**anguage) which enables the programmer to replace the OpenGL fixed-function vertex and fragment processing
- A public domain source-code implementation called *Mesa* is available for a wide range of platforms

OpenGL Applications

- Professional 3D Graphics & Effects
  - 3D graphics & special effects in the movie industry.
  - CAD/CAM/CAE, entertainment, medical imaging and virtual reality.
  - All leading 3D modelling, rendering & animation, and visualization software packages use OpenGL (3D StudioMax, Lightwave3D, AVS Express, Amira, …).
- Games
  - OpenGL allows hardware accelerated rendering, texture-mapping and special effects.
  - Many leading games support both OpenGL and DirectX (Halo 2, Half-Life 2, World of Warcraft etc.) for hardware acceleration.
- Mobile applications
  - Nokia has licensed Hybrid Graphics' OpenGL ES
  - http://www.khronos.org/opengles/

4.3 GL/GLU/GLUT etc

- In usual C implementations, *OpenGL* has three components
  - GL, GLU and GLUT
  - There may also be a package for directly interfacing to a particular windowing system e.g. GLX for OpenGL on X-window systems, but we'll ignore.
  - Graphical User Interfaces (GUIs) are often developed using a platform independent scripting language (Python, Tcl/Tk)



GL/GLU/GLUT etc (cont'd)

- GL is the main function library for polygon rendering
  - All function names begin with "gl"
  - Increasingly, the GL functions are done in hardware on the graphics card.
- GLU is the OpenGL Utility library
  - Extra functions, such as tesselations of spheres, cones, curved surfaces …
  - All function names begin with "glu"
  - Functionality provided through calls to GL functions
- GLUT, the OpenGL Utility Toolkit
  - Not officially specified/supported by ARB
  - Provides support for a windowing environment in a window-system independent manner
    - Window creation/destruction, Pop-up menus, Mouse and keypress interactions
  - But rather limited (e.g. no pull down menus, toolbars, panes, splits, scrolling, …)

GL/GLU/GLUT etc (cont'd)

- OpenGL, GLU and GLUT come as include libraries (.lib) and as dynamic link libraries (.dll)
- Two versions: Microsoft (opengl32.lib, glu32.lib, glut32.lib) and Silicon Graphics (opengl.lib, glu.lib, glut.lib).
  - We are using the Microsoft version.
  - Microsoft libraries very old, but you get access to the latest OpenGL functionality by using OpenGL extensions (http://oss.sgi.com/projects/ogl-sample/registry/) and by downloading a suitable driver for your graphics card
- Both versions use the same header files (gl.h, glu.h, glut.h)
- Downloads, manuals and examples are available on the COMP 372 *Resources* page.

How to use OpenGL?

- A typical OpenGL program looks like this:

```
Create_a_Window     // open a window into the frame buffer into which the
                    // program will draw.
                    // A GL context is associated with the window and all
                    // subsequent OpenGL commands are with respect to the
                    // current context:
Define_view         // Specify how scene (2D or 3D) is mapped onto a window
                    // on the screen
Define_rendering_parameters     // For example, lighting
Draw_the_scene      // Set colour or material properties of objects.
                    // Draw object
                    // Note: A scene (a collection of geometric objects) must be
                    // converted into primitives (polygons) before calling the
                    // OpenGL drawing routines. GLUT defines a number of
                    // geometric objects (sphere, torus, …) using polygons.
```

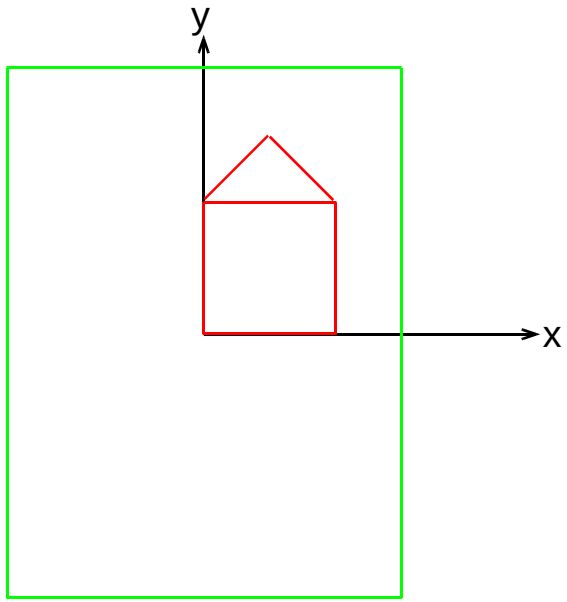
4.4 A Simple OpenGL Program

- What the program does
- The code
- Aspects of the code
  - Include the graphic libraries
  - Represent the Wireframe House
  - Create a Drawing window
  - Initialise window & view
  - Draw the Picture
- Summary
- Remarks
- Exercises

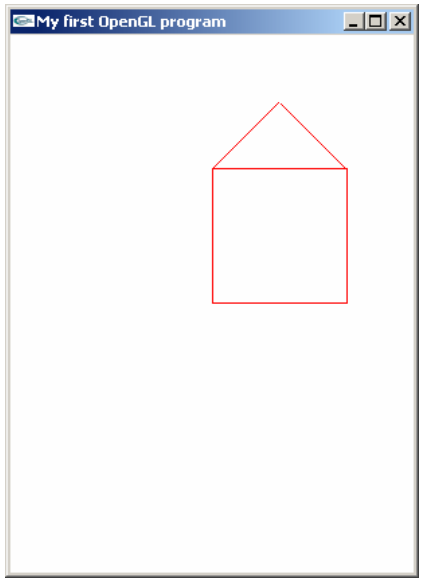
What the program does

- Defines a simple house shape in *wireframe* form (i.e. made up of straight lines representing the edges) in 2D-space.

- Displays a picture of the house using a 2D *orthographic projection*
  - Maps a section of the 2D coordinate system onto the output window.



The Situation



The Program Output

The code

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

const int windowWidth=300;  const int windowHeight=400;
// define vertices and edges of the house
const int numVertices=5;  const int numEdges=6;
const float vertices[numVertices][2] =  {{0.0, 0.0},{100.0, 0.0},{0.0, 100.0},{100.0, 100.0},{50.0,
   150.0}};
const int edges[numEdges][2] = {{0, 1},{1, 3},{3, 2},{2, 0},{2, 4},{3, 4}};

void display(void){
   glClear(GL_COLOR_BUFFER_BIT);   // clear all pixels in frame buffer

   glColor3f (1.0, 0.0, 0.0);                  // draw edges in red [given as RGB (red,green,blue) value]
   glBegin(GL_LINES);
   for(int i=0;i<numEdges;i++){
      glVertex2fv(vertices[edges[i][0]]);
      glVertex2fv(vertices[edges[i][1]]); }
   glEnd();
   glFlush ();                                 // start processing buffered OpenGL routines
}
```

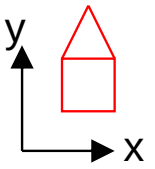
The code (cont'd)

```
void init(void) {
   // select clearing color (for glClear)
   glClearColor (1.0, 1.0, 1.0, 0.0);          // RGB-value for white
   // initialize view (simple orthographic projection)
   GLdouble halfWidth=(GLdouble) windowWidth/2.0;
   GLdouble halfHeight=(GLdouble) windowHeight/2.0;
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluOrtho2D(-halfWidth, halfWidth, -halfHeight, halfHeight);
}

// create a single buffered colour window
int main(int argc, char** argv){
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize(windowWidth, windowHeight);
   glutInitWindowPosition(100, 100);
   glutCreateWindow("My first OpenGL program");
   init ();                                    // initialise view
   glutDisplayFunc(display);                   // draw scene
   glutMainLoop();
   return 0;
}
```

OpenGL code ⇔ Graphics Pipeline

Define vertices (glVertex2fv)



No model transformations in example!

MODELVIEW Transformation

Illumination

No illumination in example!

PROJECTION Transformation

Project vertices onto OpenGL window (gluOrtho2D) – uses info from GLUT initialisation

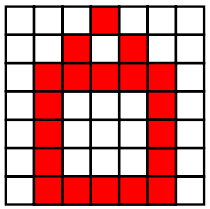
Clipping

No clipping!

Viewport Transformation

No viewport transformation!

Rasterization



Vertices get assembled to lines [glBegin(GL_LINES)] and drawn on the screen

Display

Aspects of the Code

- Include the graphic libraries
- Define the scene
- Create a drawing window
- Initialise window & view
- Draw the scene

Including the graphic libraries

- `#include <windows.h>`
  - includes constants and function prototypes of the *Graphic Device Interface* (GDI), which is at the core of all Windows graphics.
- `#include <gl/gl.h>`

  `#include <gl/glu.h>`

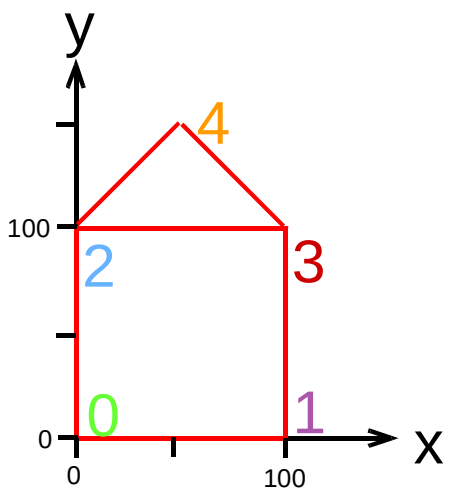
  `#include <gl/glut.h>`
  - include constants and function prototypes of the OpenGL, GLU, and GLUT libraries, respectively.
  - Header files are stored in a subdirectory ‘gl’ of the Visual C++ ‘include’ directory.
  - `glut.h` includes all other header files, but it is good style to include them explicitly as done above

Defining the Scene

- Have a vertex table and an edge table

```
const int numVertices=5;
const int numEdges=6;
const float vertices[numVertices][2] = {{0.0, 0.0},
    {100.0, 0.0},{0.0, 100.0},{100.0, 100.0},{50.0, 150.0}};
const int edges[numEdges][2] = {{0, 1},{1, 3},{3, 2},
                                {2, 0},{2, 4},{3, 4}};
```

- Each vertex table array entry is an array of 2 floats, representing a point in R^2 [2D-space]
- Edge table values are *indices* into vertex table
  - e.g. edge {1,3} is the edge from V_1=(100.0, 0.0) to V_3=(100.0, 100.0).
  - Ordering of both vertex table and edge table is arbitrary
  - Ordering of vertices in an edge is also arbitrary, e.g. {1,3} is identical to {3,1}



Creating a Drawing Window

- `glutInit(int *argcp,char** argv);`
  - Initiates Window session with the underlying operating system (OS)
  - Uses command line arguments from `main()` [OS-specific].
- `glutInitDisplayMode(unsigned int mode);`
  - Specifies display mode for the to be created window .
  - Select single buffered (`GLUT_SINGLE`) colour **(**`GLUT_RGB`**)** window (i.e. the window is associated with one frame buffer of RGB values).
- `glutInitWindowSize(int width, int height);`
  - Width and height (in pixels) of the to be created window.

Creating a Drawing Window (cont'd)

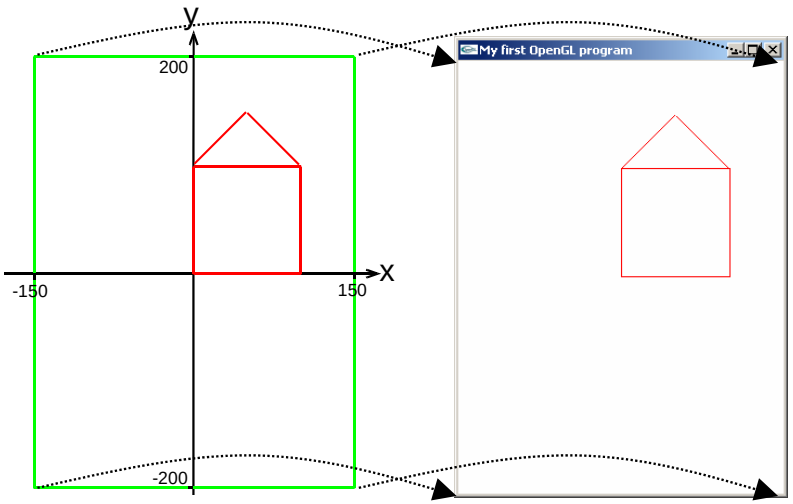
- `glutInitWindowPosition(int x, int y);`
  - x and y coordinate of the screen location of the to be created window.
- `glutCreateWindow(char* s);`
  - Create window with the title `s`.
  - Size, location, and mode of the window are specified by the current state of the OpenGL program (set by the previous three commands).
  - Returns a unique window identifier. Important when using multiple windows.

Initialisation window & view

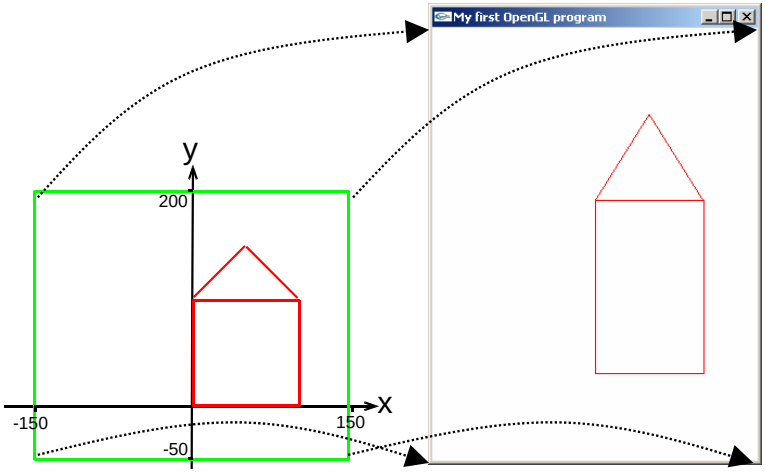
- `glClearColor(Glclampf red, Glclampf green, Glclampf blue, Glclampf alpha);`
  - Set colour used for clearing (`glClear()`) the drawing window.
  - Colour is an RGBA tuple (red, green, blue, alpha) where alpha is transparency.
  - Arguments are floats (values are clamped to the range [0,1])
- `glMatrixMode(Glenum mode);`
  - Applies subsequent matrix operations to the matrix stack specified by `mode`.
  - `GL_PROJECTION` selects the projection matrix stack.
  - The projection matrix specifies how the scene (2D or 3D) is projected onto the 2D drawing window.
- `glLoadIdentity();`
  - Initialise the matrix stack with an identity matrix.

Initialisation window & view (cont'd)

- `gluOrtho2D(Gldouble left, Gldouble right, Gldouble bottom, Gldouble top);`
  - Defines a 2D orthographic projection matrix.
  - Maps a section of the 2D world coordinates (the coordinates in which the scene is defined) onto the drawing window.
  - Ratio (right-left):(top-bottom) should be the same as width:height of the window →otherwise scene is distorted



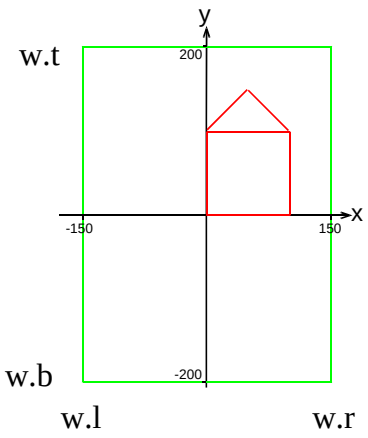
(a) Using '`gluOrtho2D(-150, 150, -200, 200)`'

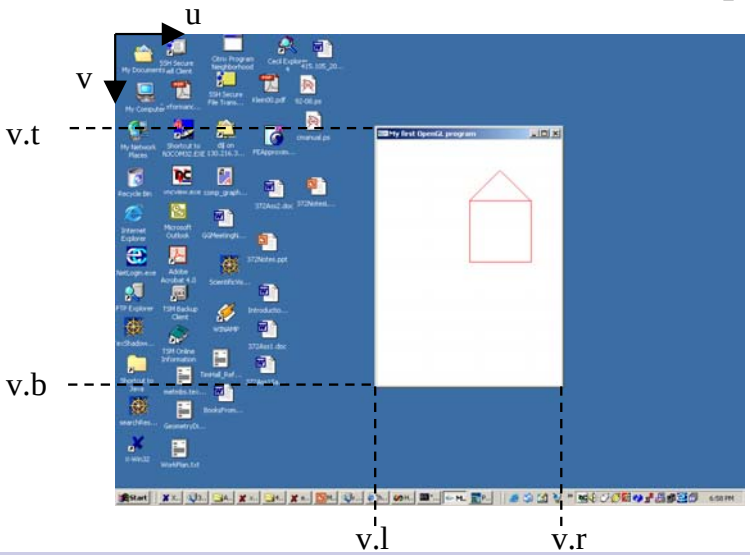


(b) Using '`gluOrtho2D(-150, 150, -50, 200)`'

gluOrtho2D

- How does `gluOrtho2D` do the mapping?
  - `gluOrtho2D` implements a *(world-)window-to-viewport mapping*.
  - The *world-window* is the section of the world coordinates we want to display (specified by the coordinates w.l, w.r, w.b, w.t).
  - The *viewport* is the drawing window on the screen.
  - The viewport is described with respect to the screen coordinates (eg. top-left pixel of the screen is (0,0) and the bottom-right pixel of screen (1023,767)).
  - The viewport is then specified by the coordinates v.l, v.r, v.t and v.t with respect to those screen coordinates.





gluOrtho2D (cont'd)

Denote the world coordinates by $\mathbf{x} = \begin{pmatrix} x \\ y \end{pmatrix}$

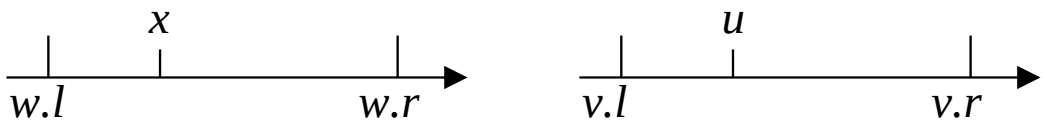
and the screen coordinates by $\mathbf{u} = \begin{pmatrix} u \\ v \end{pmatrix}$

then the world-to-viewport mapping is described by the equation

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} A & 0 \\ 0 & B \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} C \\ D \end{pmatrix} = \begin{pmatrix} Ax + C \\ By + D \end{pmatrix}$$

This equation can be described by a single matrix multiplication (see chapter 5). The matrix is pushed onto the projection matrix stack!

gluOrtho2D (cont'd)



The ratio of the distances of x to the left and right window boundary must be equal to the ratio of u to the left and right viewport boundaries (proportionality of x)

$$\frac{u - v.l}{v.r - v.l} = \frac{x - w.l}{w.r\text{-}w.l} \quad \text{or} \quad u = \frac{v.r - v.l}{w.r - w.l}x + \left(v.l - \frac{v.r - v.l}{w.r - w.l} w.l \right)$$

Comparing coefficents with the linear equation from the previous slide gives

$$\mathrm{A} = \frac{v.r - v.l}{w.r - w.l} \qquad \text{and} \qquad \mathrm{C} = v.l - A * w.l$$

Similarly proportionality for y gives

$$\mathrm{B} = \frac{v.t - v.b}{w.t - w.b} \qquad \text{and} \qquad \mathrm{D} = v.b - B * w.b$$

Drawing the Scene

- `glClear(GLbitfield mask );`
  - Clears all pixels in the buffer specified by mask.
  - In order to clear colour buffer use `GL_COLOR_BUFFER_BIT` as mask.

    → Sets all pixels of the drawing window to the previously defined ‘clear-colour’.

- `glColor3f(GLfloat red, GLfloat green, GLfloat blue );`
  - Sets colour for subsequent drawing commands.
  - Colour is an RGB tuple (red, green, blue).
  - Many different versions of this command are available:
    - `void` **`glColor3b`**`( GLbyte red, GLbyte green, GLbyte blue )`
    - `void` **`glColor3d`**`( GLdouble red, GLdouble green, GLdouble blue)`
    - `void` **`glColor4f`**`( GLfloat red, GLfloat green, GLfloat blue, GLfloat alpha )`
    - see manual for a complete listing

Drawing the Scene (cont'd)

- `glBegin(Glenum mode);`
  …
  `glEnd();`
  - `glBegin` and `glEnd` delimit the vertices that define a primitive or a group of like primitives.
  - The type of primitive is specified by the argument `mode`.
  - `GL_LINES` treats each pair of vertices as an independent line segment. Vertices *2n-1* and *2n* define line *n*. *N/2* lines are drawn.

- `glVertex2fv(const GLfloat *v )`
  - *v* specifies a pointer to an array of two float numbers representing a vertex.
  - the `glVertex` command is used within glBegin/glEnd pairs to specify point, line, and polygon vertices. The current color, normal, and texture coordinates are associated with the vertex when glVertex is called.

- `glFlush()`
  - empties all buffers, causing all issued commands to be executed as quickly as they are accepted by the actual rendering engine.

Summary

```
void display(void){
   // draw scene objects
}
```

```
void init(void) {
   // set up background colour, window etc.
   // set up view projection (e.g. glOrtho2D)
   // set up scene
}
```

```
int main(int argc, char** argv){
   // create a single buffered colour window
   // initialise view and scene
   // set up event handling
   glutDisplayFunc(display);     // draw scene
   glutMainLoop();
   return 0;
}
```

- An OpenGL program has 3 important parts:
  - Like all C programs it has a `main` function, which is used to initialise the drawing window and set up the scene and event handling.
  - Before drawing a scene the scene and the virtual camera (view of the scene) must be defined
    - Can be done inside the `display` function but more efficient to define a separate method `init`.
    - Only called once, except if the view is changed (e.g. zoom in).
  - Scene is draw by the `display` function.
    - Window must be redrawn if it is created, moved or uncovered.
    - Performed automatically by GLUT by calling `glutDisplayFunc(display)`.
- Two parts missing in our example:
  - could define `reshape` function in case the drawing window is resized.
  - event handling (mouse and keyboard input).

Remarks

- Note that many methods (e.g. `glColor2f`, `glVertex2fv`) are of the form: `glXXX<n><t>[v]`, where
  - *n* is dimension of quantity being defined (3 ⇒ 3-space)
  - *t* is type of parameter (*f* = float)
  - optional *v* denotes "vector" parameter, e.g. *glVertex3fv* takes a 3-element array of floats as a parameter, whereas *glVertex3f* takes three separate floats, (*x*,*y*,*z*)
- OpenGL often defines several different forms of each call, e.g. `glVertex{234}{sifd}[v]`
  - 24 forms in total!
  - In 372 we are using the *f* form (float) everywhere
    - Simplest to stick with a single type when learning
    - Although *double* is often a little more convenient, it costs too much in memory and performance.

Remarks (cont'd)

- The *GL* data types (`Glfloat`, `Gldouble`, …) are not *C* types.
  - For example, `GLint` is not necessarily equivalent with the *C* type `int`.
  - The reason for this is that OpenGL needs for each data type a certain minimum number of bits in order to get the necessary precision for graphics operation.
  - The corresponding C data types are specified in the file `gl.h`, e.g.

```
typedef float GLfloat;
typedef float GLclampf;
typedef double GLdouble;
…
```

  - Hence if you use `float` instead of `Glfloat` you won't get a warning message. However, it's a good style to use the *GL* data types in case you port you program to another machine.

Exercises [Try doing all these]

- Change the program to draw the picture in white on a black background
- Change the program to use *glVertex2f* everywhere instead of *glVertex2fv*.
- Modify the program such that the left-bottom corner of the house is at the left-bottom corner of the window
  - Do this by modifying the vertex table only
  - Do this by modifying the arguments of `gluOrtho2D` only
- Add a variable *f* to the program and modify the program such that it increases the size of the picture in the output window by a factor *f*.
- Draw only the vertices of the house (use `GL_POINTS`)
  - In order to better see the points call `glPointSize(3.0)` before `glBegin`.

4.5 Geometric Primitives in OpenGL

- As mentioned before the command sequence

```
glBegin(Glenum mode);
…
glEnd();
```

defines a primitive or a group of like primitives from the *N* vertices defined in between.

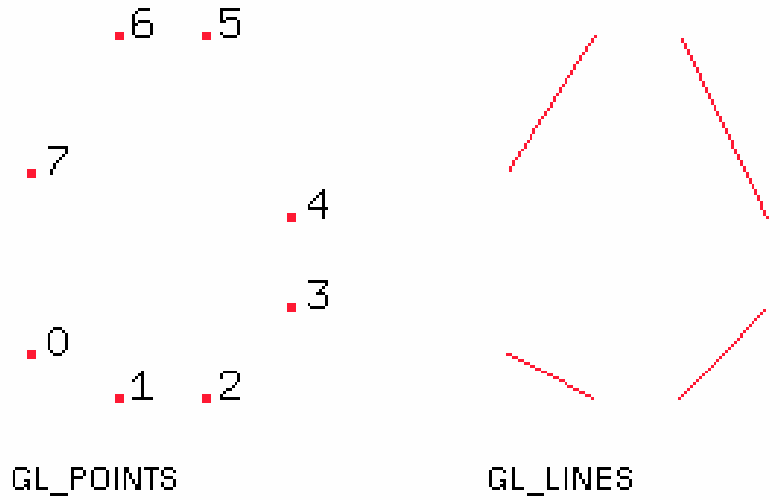
The argument `mode` specifies how the vertices are interpreted and can be any of the following:

- `GL_POINTS`
- `GL_LINES`
- `GL_LINE_STRIP`
- `GL_LINE_LOOP`
- `GL_TRIANGLES`
- `GL_TRIANGLE_STRIP`
- `GL_TRIANGLE_FAN`
- `GL_QUADS`
- `GL_QUAD_STRIP`
- `GL_POLYGON`

Geometric Primitives (cont'd)

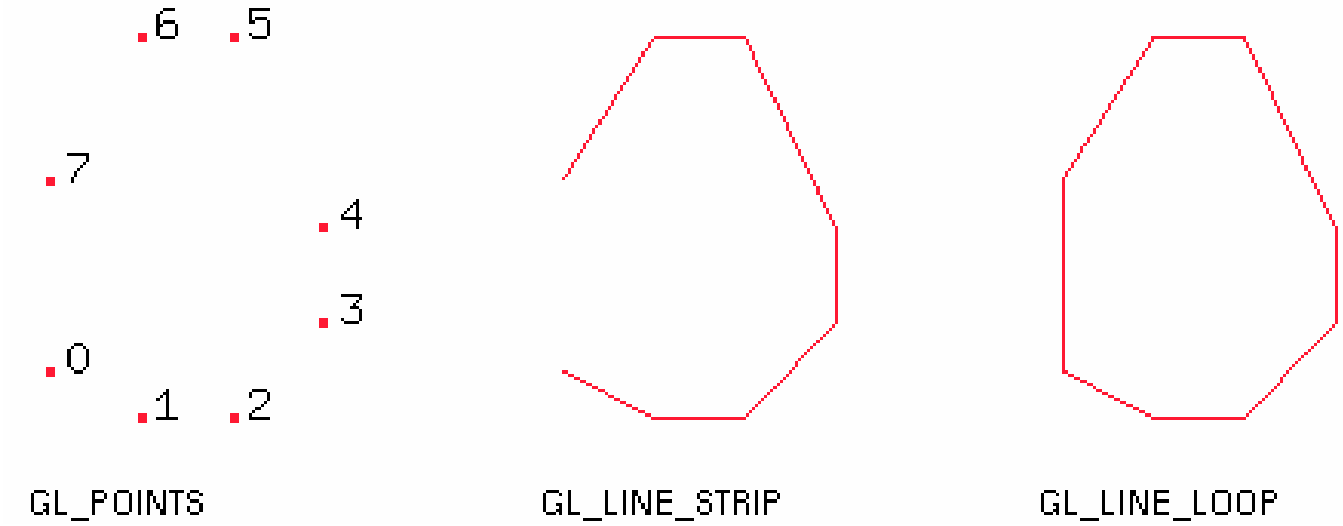
- `GL_POINTS`
  - treats each vertex as a single point.
  - vertex *n* defines point *n* (*n*=0,...,*N*-1).

- `GL_LINES`
  - treats each pair of vertices as an independent line segment.
  - Vertices *2n* and *2n+1* defines line *n* (*n*=0,...,*N/2*-1).



Geometric Primitives (cont'd)

- `GL_LINE_STRIP`
  - draws a connected group of line segments from the first vertex to the last.
  - vertices *n* and *n+1* define line *n* (*n*=0,...,*N*-2).
- `GL_LINE_LOOP`
  - as above, but additionally a line segment is drawn from the last vertex to the first.

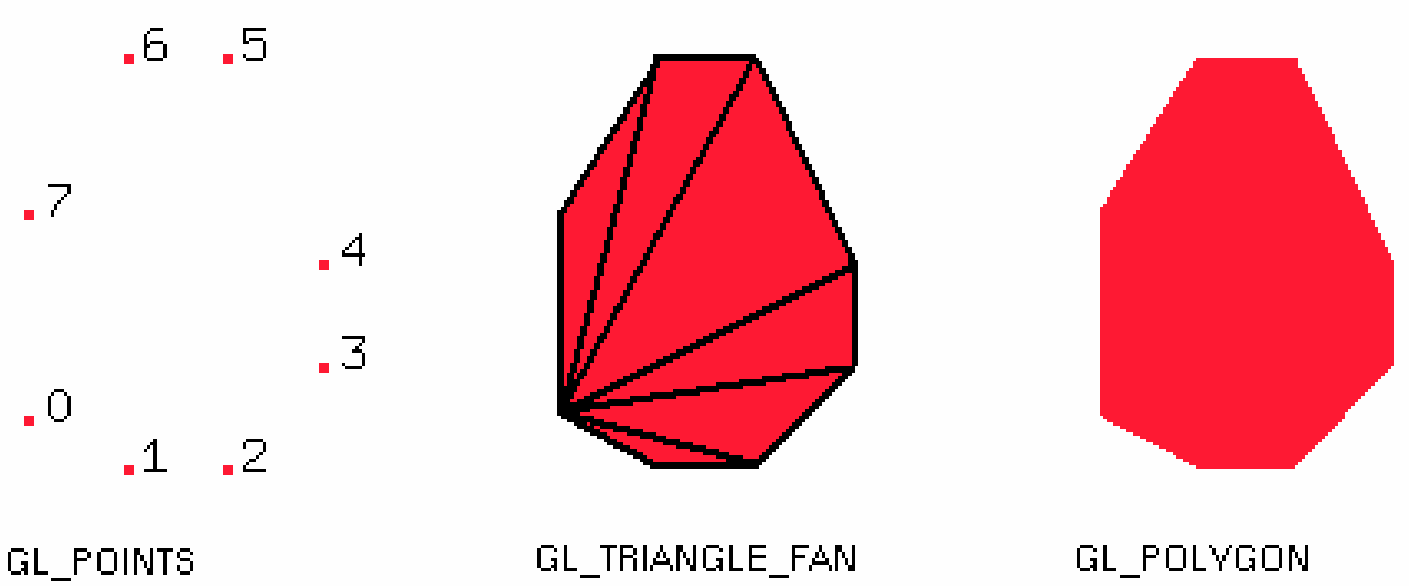


Geometric Primitives (cont'd)

- `GL_TRIANGLES`
  - treats each triplet of vertices as one independent triangle.
  - vertices *3n*, *3n+1* and *3n+2* define triangle *n* (*n*=0,…,*N/3*-1).
- `GL_TRIANGLE_STRIP`
  - draws a connected group of triangles with the first three vertices defining a triangle and each subsequent vertex forming a triangle with the last two vertices of the previous triangle.
  - for even *n* vertices *n*, *n+1* and *n+2* define triangle *n*, for odd *n* vertices *n+1*, *n* and *n+2* define triangle *n* (*n*=0,…,*N-3*).

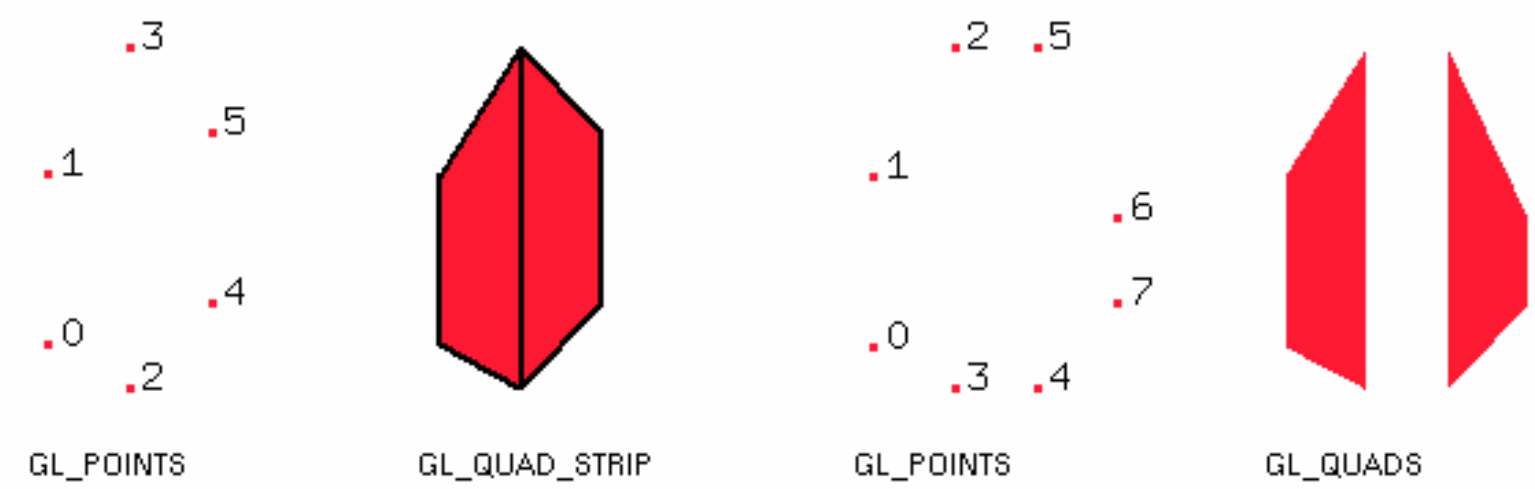
Geometric Primitives (cont'd)

- `GL_TRIANGLE_FAN`
  - draws a connected group of triangles. Each pair of vertices after the first vertex defines one triangle with the first vertex.
  - vertices *0*, *n+1* and *n+2* defines triangle *n* (*n*=0,…,*N*-3).
- `GL_POLYGON`
  - draws a single *convex* polygon defined by the vertices *0* to *N*-1.



Geometric Primitives (cont'd)

- `GL_QUADS`
  - treats each group of four vertices as an independent quadrilateral.
  - vertices *4n* , *4n+1* , *4n+2* and *4n+3* define quadrilateral *n* (*n*=0,…,*N*/4-1).
- `GL_QUAD_STRIP`
  - draws a connected group of quadrilaterals with the first four vertices defining one quadrilateral and each subsequent pair of vertices defining a quadrilateral with the last two vertices of the previous one.
  - vertices *2n* , *2n+1* , *2n+3* and *2n+2* define quadrilateral *n* (*n*=0,…,*N*/2-2).



Remarks

- Geometric primitives are defined identical in 2D and 3D with the dimension of the vertices being the only difference.
- The vertices of a triangle strip or quad strip should be all either in clockwise or in anticlockwise order (necessary for a consistent surface orientation in 3D).
- Point size is modified with `glPointSize(Glfloat size)`.
- Line width is modified with `glLineWidth(Glfloat width)`.
- Can apply a colour either to all vertices or to each vertex individually. In the latter case the vertex colours are *interpolated*.