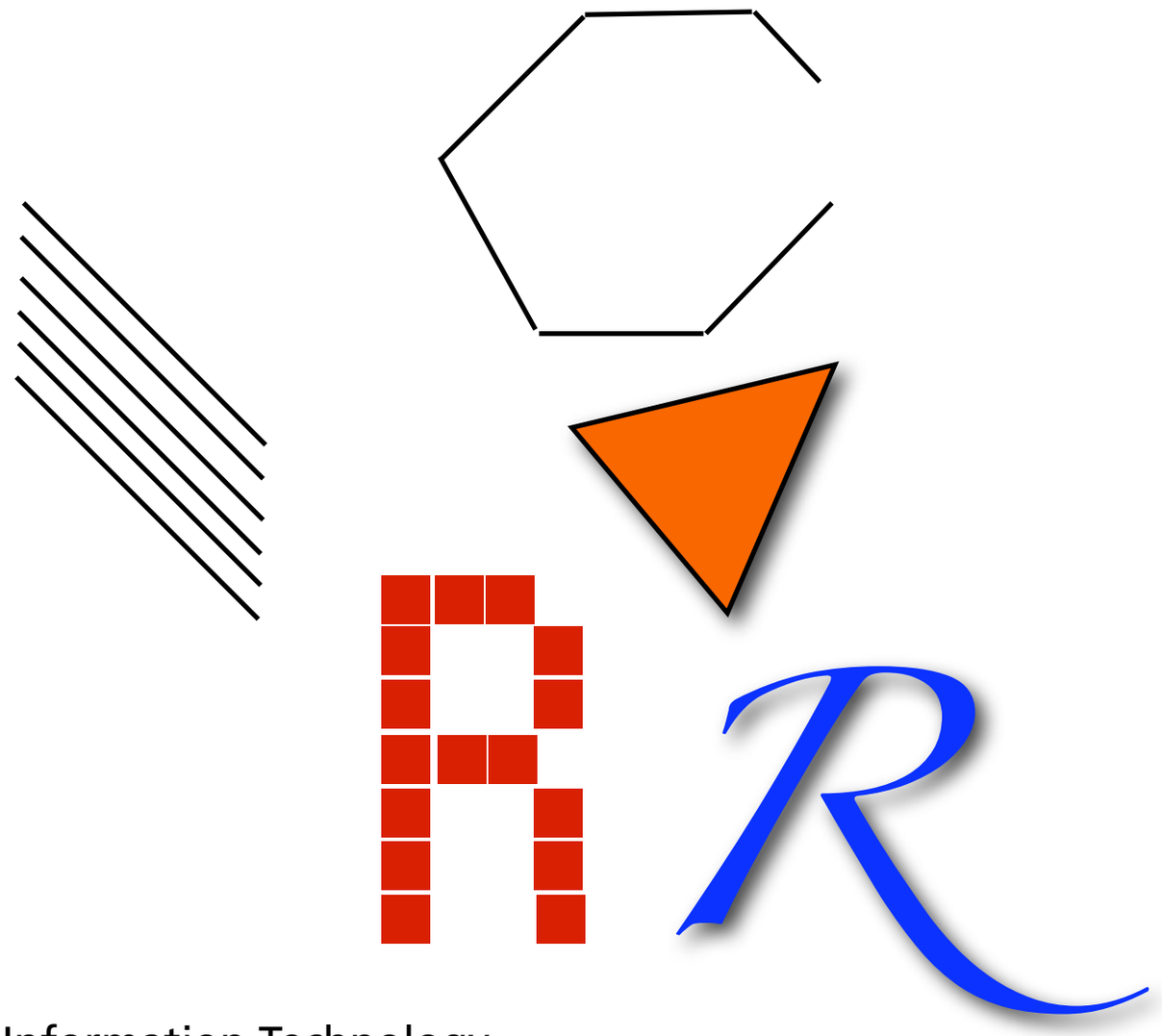


Monash University • Clayton's School of Information Technology

# CSE3313 Computer Graphics

Lecture 4: OpenGL Part 1

# What is OpenGL?

- Hardware and OS independent standard for real-time 2D and 3D graphics — API (Application Programmer's Interface).
- Standard for the majority of graphics cards for workstations and PCs.
- Other graphics APIs include:
  - GKS, PHIGS (mainly obsolete)
  - Java 3D
  - Direct X
  - RenderMan — for 'realistic' rendering.
- Interface between the program and the graphics input/output systems.

## History of OpenGL

- 1980s: GL (Graphics Language) on SGI Workstations;
- 1992: OpenGL 1.0 Specification June 1992, first versions in 1993;
- 1997: OpenGL 1.1 – added vertex arrays and texture objects;
- 1998: OpenGL 1.2 – 3D textures and imaging functionality; optional features permitted in different implementations;
- 2001: OpenGL 1.3 – cube texture maps, compressed textures, multi-texturing;
- 2002: OpenGL 1.4 – automatic mipmap generation, additional blending functions, shadow maps, multiple vertex arrays in single command, stencil wrapping, extra texturing commands;
- 2003: OpenGL 1.5 –vertex buffer objects, shadow comparison functions, occlusion queries, non power-of-2 textures

## History of OpenGL (continued)

- Up to OpenGL 1.5 the rendering pipeline is more or less fixed, the application program doesn't have any control on how pixels are shaded, for example.
- 2004: OpenGL 2.0 – support for application specific rendering algorithms using a high-level shading language.
- (http://www.opengl.org/documentation/specs/version2.0/glspec20.pdf)
- OpenGL 2.0 is 'upwards compatible' with OpenGL 1.0
- OpenGL Shading Language (GLSL) is a high-level, C-like language used to write *vertex* and *fragment* shaders.

# The OpenGL Interface

- OpenGL function names all begin with **`gl`** and are stored in a library called *GL.*
- Related libraries:

  **Graphics Utility Library** (GLU)

  - contains only calls to GL functions
  - contains code for common objects, e.g. spheres
  - available on all OpenGL implementations.

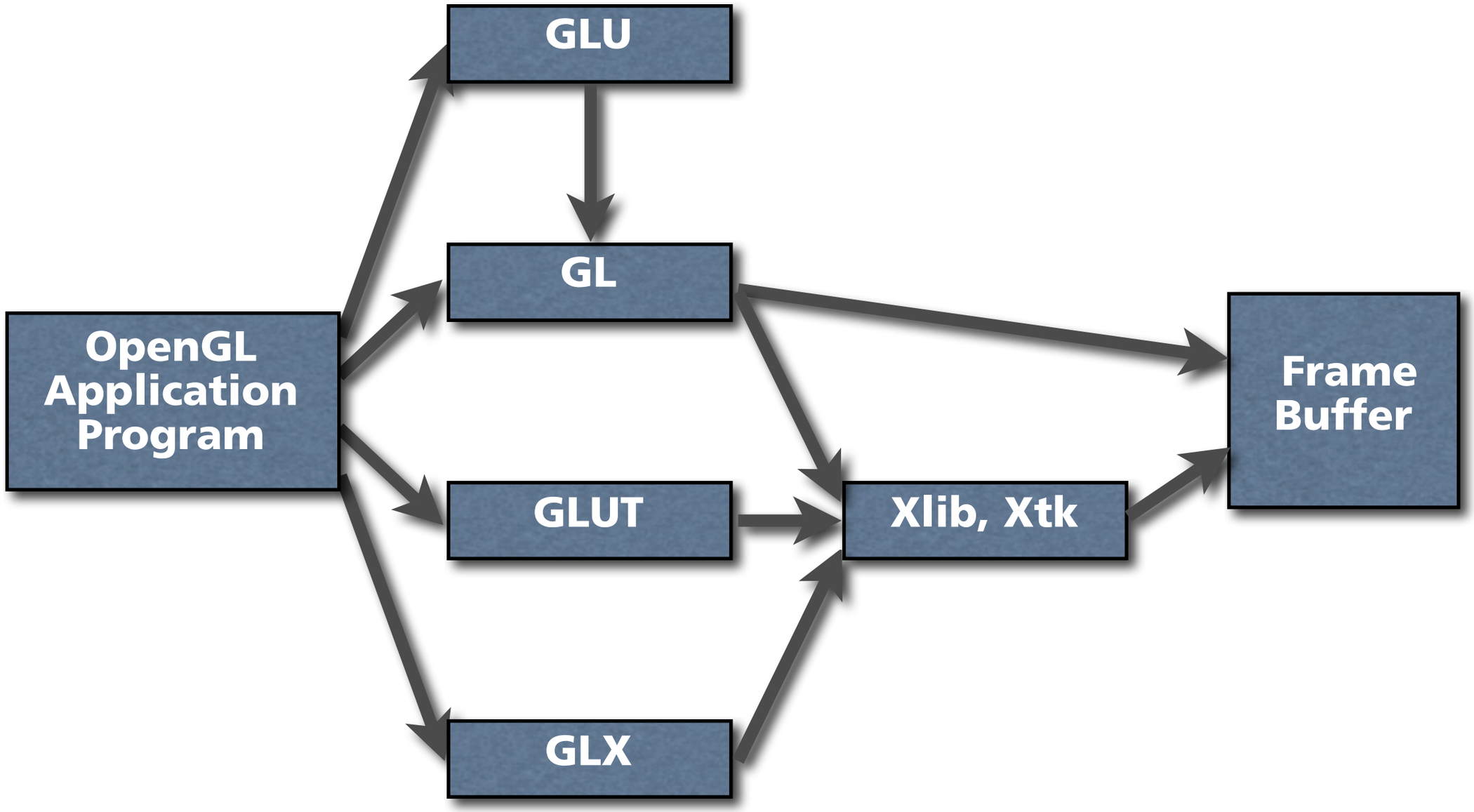
- **GL Utility Toolkit** (GLUT)
  - allows program to interface with the window system and input devices, independent of OS and window system.

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

    or **`#include <glut.h>`**

    will read in glut.h, glu.h and gl.h

# OpenGL Library Organization



## Primitives and Attributes

- The basic OpenGL primitives are specified via points in space or *vertices.*

- By default, a point covers 1 pixel. Objects are defined by sequences of the form:

```
glBegin (type);
    glVertex*(...);
    .
    .
    glVertex*(...);
glEnd();
```

- Other code and OpenGL function calls can occur between `glBegin` and `glEnd`.

# glVertex

- The general form for a vertex is:

  `glVertex`

  where * can be interpreted as either `nt` or `ntv` where

  - n denotes the number of dimensions (2,3, or 4);

  - `t` denotes the data type:

    `i` – integer, `f` – float, `d` – double;

  - `v`, if present denotes that variables are specified through a pointer to an array and not through an argument list.

  - Examples:

```
glVertex2i( GLint Xi, GLint Yi );
glVertex3f( GLfloat x, GLfloat y, GLfloat z );
GLfloat vertex[3];
glVertex3fv( vertex );
```

## glVertex (cont.)

- The underlying representation is always the same and actually uses 4 coordinates.
- All other geometric types are specified either in terms of points or **line segments**.
- Line segments themselves are specified by pairs of points.
- Apart from points and line segments other basic geometric types have interiors that can be coloured in different ways.
- **Line segments** — `GL_LINES` — successive points define the end points of each line segment. The line segments themselves are disconnected.
- **Polylines** — `GL_LINE_STRIP` — successive vertices (and line segments) are connected. To force a polyline to be closed, we can use `GL LINE LOOP`.

# Point and Line Segment types

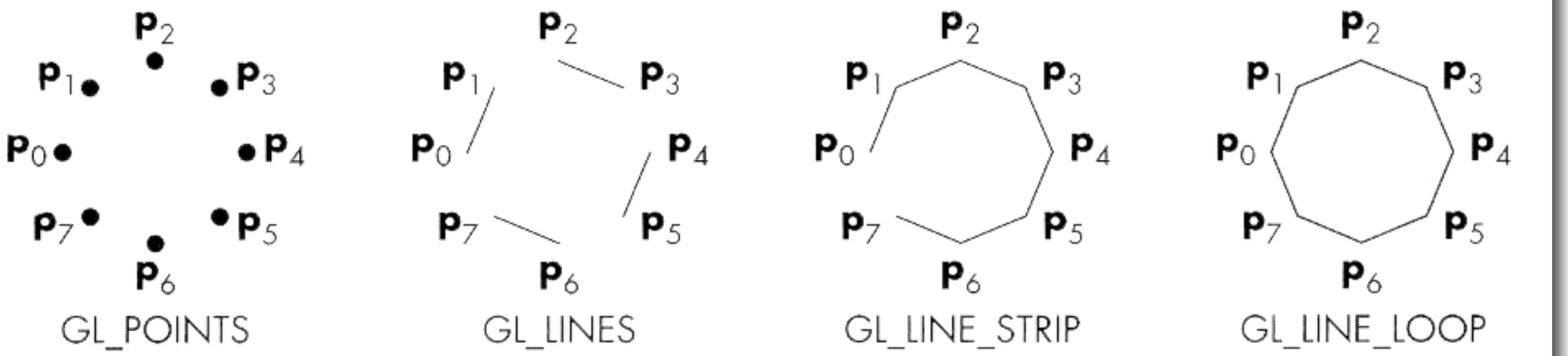



Figure 2.7 Point and line-segment types.

## Polygon Basics

- Line segments and polylines can model the edges of objects. An object that has a border that can be described by a line loop and which has an interior is called a **polygon**.
- Polygons can be displayed rapidly by graphics hardware and can be used to approximate curved surfaces.
- In two dimensions, as long as no two edges of a polygon cross each other the polygon is **simple**. If an API requires that polygons be simple, it will generally not check. It will be up to the applications program to ensure polygons are simple.
- Some implementations require polygons to be *convex*. A polygon is convex if a line segment connecting any two points in the polygon or its boundary does not go outside the polygon.

# Polygons

- In 3 dimensions a polygon is not necessarily planar (flat). A triangle will be planar if its three points are not collinear.
- Hardware and software often support a triangle polygon type that is rendered much faster than a polygon with three edges.

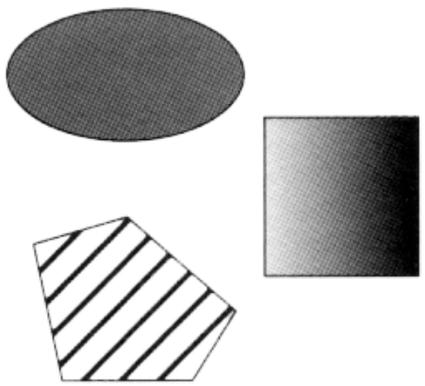


Figure 2.8 Filled objects.

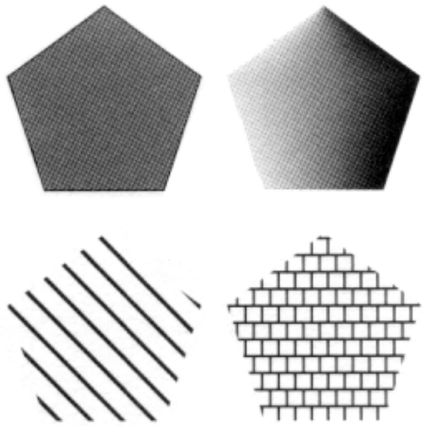


Figure 2.9 Methods of displaying a polygon.

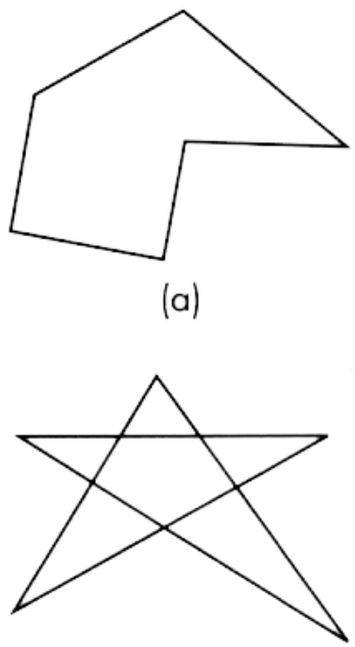



Figure 2.10 Polygons. (a) Simple. (b) Nonsimple.

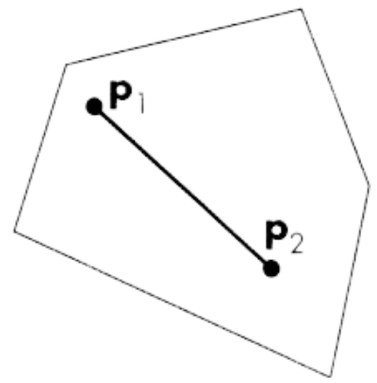



Figure 2.11 Convexity.

# Polygon Types in OpenGL

- **Polygons —** `GL_POLYGON`

  The edges are the same as if a line loop were used. Edges are assumed to have no width.

- Most graphics systems allow you to draw the edges or to fill the interior of a polygon, but not both at once. You might need to draw the edges and fill the interior in two operations.

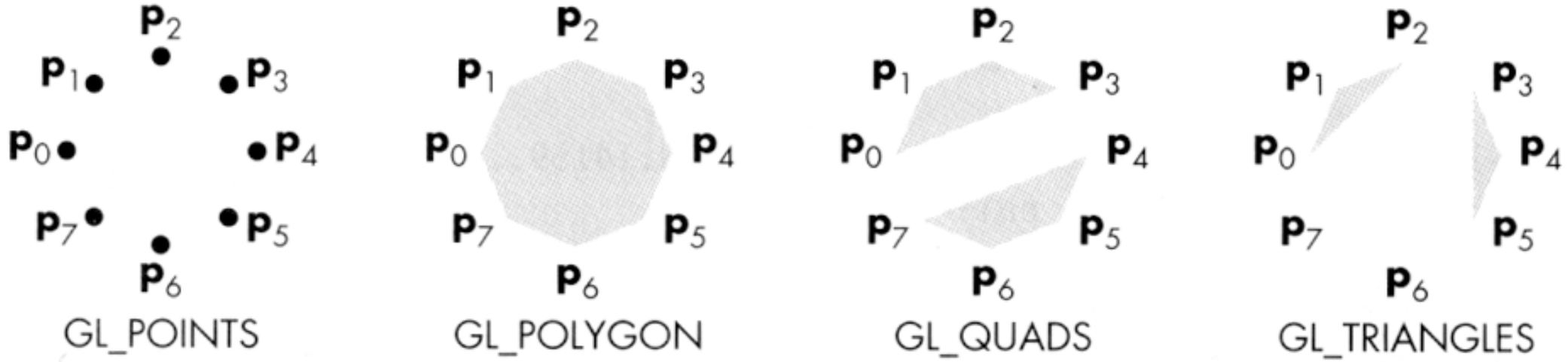



Figure 2.13 Polygon types.

## Other Geometric Types

- **Triangles** and **Quadrilaterals** — `GL_TRIANGLES` and `GL_QUADS` — are special cases of polygons. Their use might lead to increased efficiency in rendering.

- **Strips** and **Fans** — `GL_TRIANGLE_STRIP`, `GL_QUAD_STRIP` and `GL_TRIANGLE_FAN` are groups of triangles or quadrilaterals that share edges.

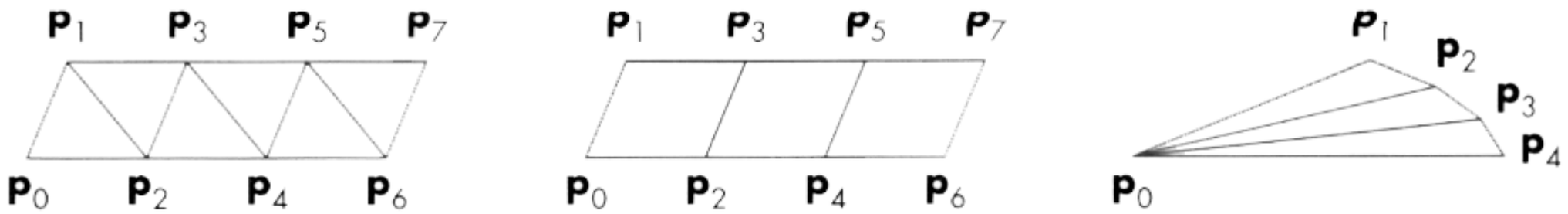



Figure 2.14 Triangle strip, quadrilateral strip, and triangle fan.

# Text in OpenGL

- There are two types of text: **stroke** and **raster**.
- Stroke text is created using vertices and line segments. It can be *transformed* (translation, rotation, scaling) and viewed (perspective and parallel projections) like other graphical objects.
- A character need only be defined once.
- Defining every character for 128 or 256 characters in a particular font can require a lot of detail. Manipulating stroke characters can require considerable computing resources.

Computer Graphics Rocks!

Stroke Text (postscript font)

# Raster Text

- Raster text is simple and fast. Each character is described as a rectangular block of bits which are either set of clear.

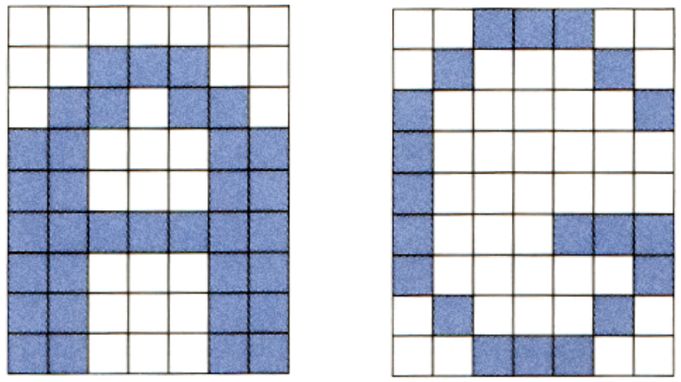


Figure 2.17 Raster text.

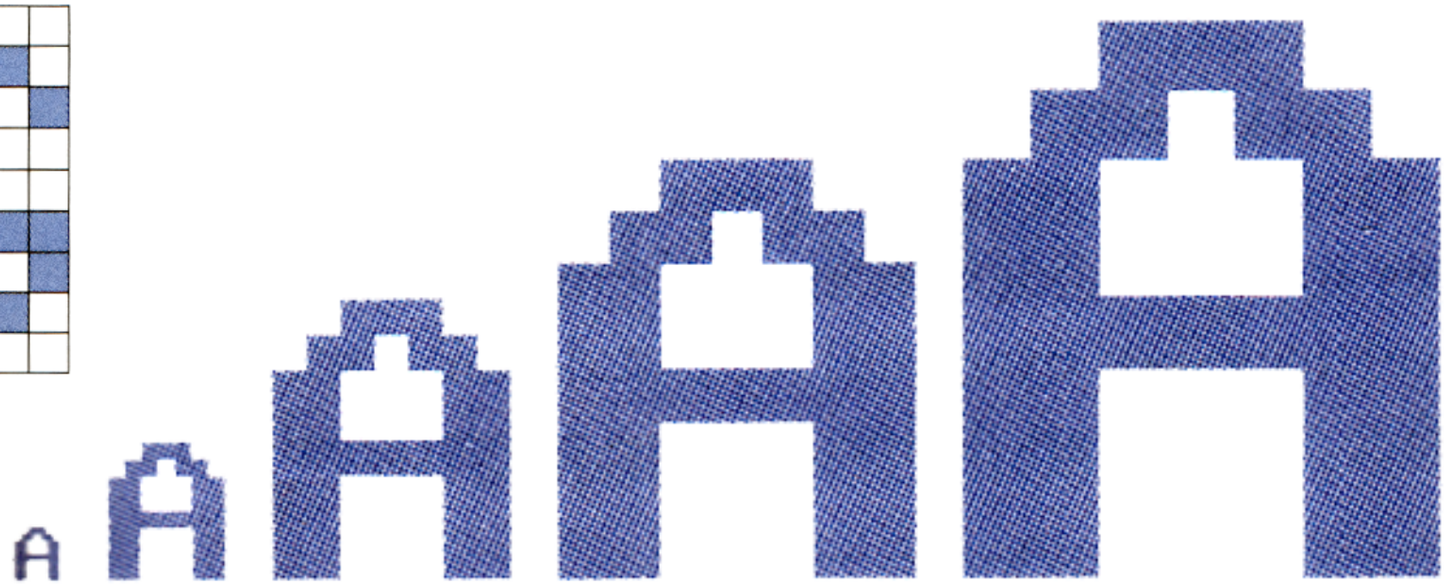


Figure 2.18 Raster-character replication.

Hardware can move raster characters quickly into the frame buffer via *bit-block-transfer* (**bitblt**) operations. OpenGL allows applications to manipulate the frame buffer directly.

Raster characters can only be increased in size by **replicating**.

## Raster Text (cont.)

- Because raster characters are rectangular arrays of bits they cannot be transformed.
- Fonts might reside in hardware on the graphics display device and may not be portable.
- OpenGL does not have text primitives. GLUT can create characters from other primitives. It provides a few bitmap and stroke character sets in software. These are portable. E.g.:
- `glutBitmapCharacter( GLUT_BITMAP_8_13, c );`
  c is the number of characters.
- Characters are placed in the current **raster position** in the display.
- This position is measured in pixels and can be altered by functions of the form

  `glRasterPos*` **e.g.** `glRasterPos2i(GLint x, GLint y)`

# Curved Objects

- OpenGL primitives are defined in terms of vertices or line segments.
- To create curved objects we can approximate curves by line segments and meshes of convex polygons — **tesselation**.
- We can also define curves and curve surfaces mathematically and build our own graphics functions to render these objects on a pixel by pixel basis.
- The GLU contains function calls to perform tesselation and to render **NURBS** curves and surfaces.