Lecture 2:
Basic Graphics Programming

Topics:

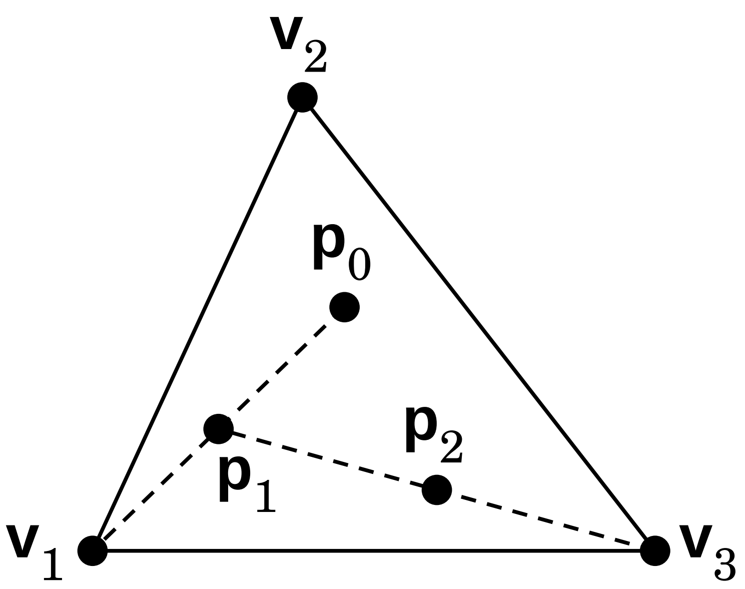
1. 2D example: Sierpinski gasket
2. The OpenGL API
3. Primitives
4. Attributes
5. Viewing
6. Transformation
7. Control

Chapter 2 of Angel.

Graphics Programming with OpenGL

A simple example: the Sierpinski Gasket

1. Pick a random point in a triangle.
2. Pick one of the vertices at random.
3. Find the point halfway between the initial point and the vertex.
4. Display the point.
5. Replace the inital point with this new point.
6. Return to step 2.



How do we display points in OpenGL?

We call

```
glVertex*
```

where `*` is 2 or 3 characters of the form `nt` or `ntv`, and

`n` = number of dimensions (2 or 3),
`t` = data type, e.g. integer (`i`), float (`f`), double (`d`),
`v` if present $\Rightarrow$ 'use an array'.

2D Points (x, y) are treated as 3D points (x, y, z) with $z = 0$. We use OpenGL types:

```
#define GLfloat float
```

Examples:

```
glVertex2i(GLint xi, GLint yi);
```

```
glVertex3f(GLfloat x, GLfloat y, GLfloat z);
```

```
Glfloat vertex[3];
glVertex3fv(vertex);
```

Vertices are grouped together to form objects, using

`glBegin` and `glEnd`.

Line segments:

```
glBegin(GL_LINES);
  glVertex2f(x1,y1);
  glVertex2f(x2,y2);
glEnd();
```

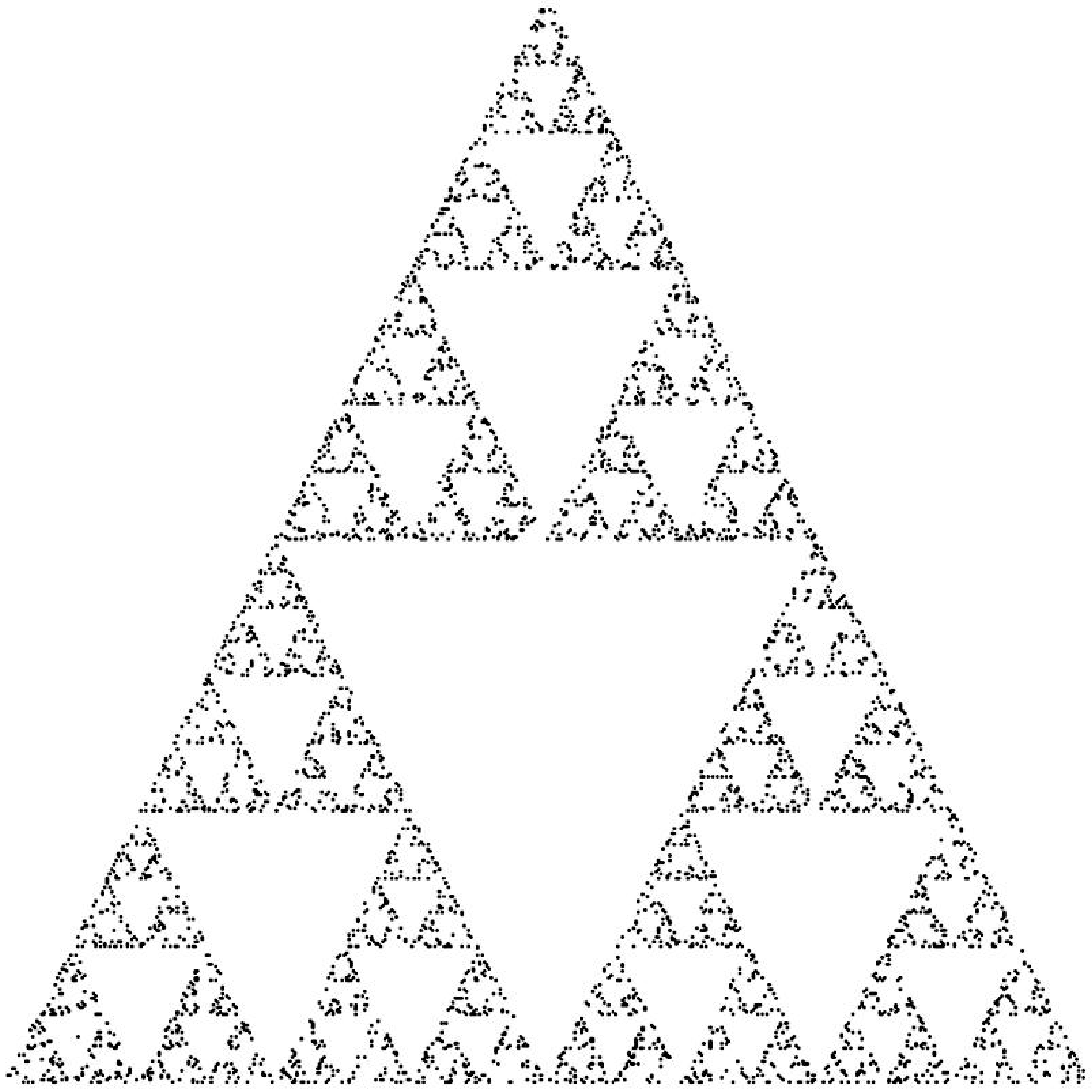
Sequences of points:

```
glBegin(GL_POINTS);
  glVertex2f(x1,y1);
  glVertex2f(x2,y2);
glEnd();
```

The `display` function for the Sierpinski Gasket:

```
void display(void)
{
  typedef GLfloat point2[2];
  point2 vertices[3]={{0.0,0.0},{250.0,500.0},{500.0,0.0}};
  int i,j,k;
  srand(0); // different to book
  point2 p = {75.0, 50.0};
  for(k=0; k<5000; k++)
  {
    j=rand() % 3;
    p[0] = (p[0]+vertices[j][0]) * 0.5;
    p[1] = (p[1]+vertices[j][1]) * 0.5;
    glBegin(GL_POINTS);
    glVertex2fv(p);
    glEnd();
  }
  glFlush();
}
```

The output of the Sierpinski Gasket:

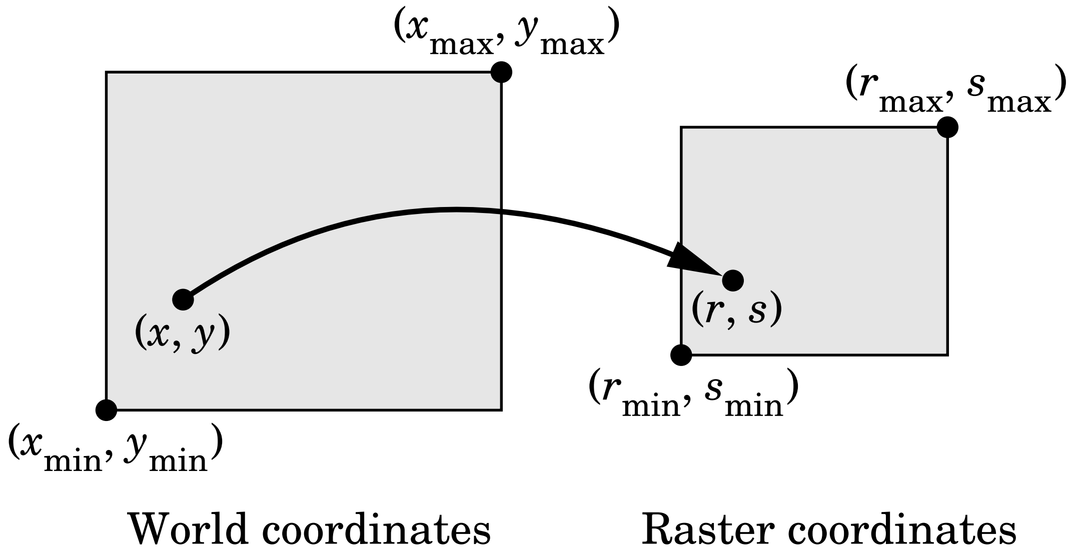


Coordinate systems

The coordinates for the objects in OpenGL are **world coordinates**. These can either be 2D or 3D coordinates. We must map these world coordinates into **raster** or **screen coordinates**.

For 2D world coordinates we specify a **clipping rectangle** in the (x, y) plane.

For raster coordinates we specify a **window**, a rectangular region of the frame buffer.



For 3D world coordinates we must also deal with projections.

The OpenGL API: graphics functions

There are typically six types of functions in a graphics API:

1. **primitive functions**: define low-level objects such as points, line segments, polygons, pixels, text, curves and surfaces.
2. **attribute functions**: how primitives appear: colour, line thickness and style, fill patterns, font types.
3. **viewing functions**: specify camera: position, orientation,
4. **transformation functions**: rotation, scaling, translation.
5. **input functions**: deal with input devies, e.g. keyboard, mouse.
6. **control functions**: communicate with window system, deal with errors, initialize programs.

The OpenGL interface

- **GL** (Graphics Library): 'basic' OpenGL, contains only basic primitives.
- **GLU** (Graphics Utility Library): contains richer set of primitives, e.g. spheres, and only calls GL.
- **GLUT** (GL Utility Toolkit): for interfacing with the window system and input devices

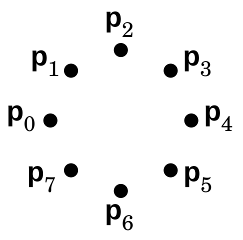
Primitive, attribute, viewing, and transformation functions are handled by GL and GLU.
Input and control functions are handled by GLUT.

Primitives

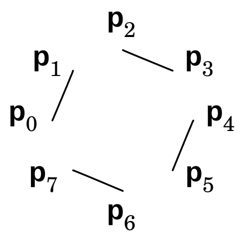
Basic OpenGL primitives are specified by sequences of points / vertices:

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

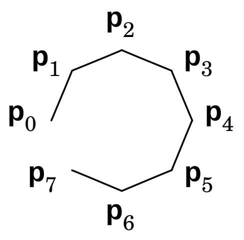
- `GL_POINTS`: Points
- `GL_LINES`: Line segments
- `GL_LINE_STRIP`: Polyline
- `GL_LINE_LOOP`: Closed polyline



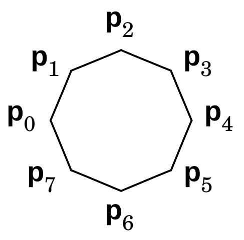
GL_POINTS



GL_LINES



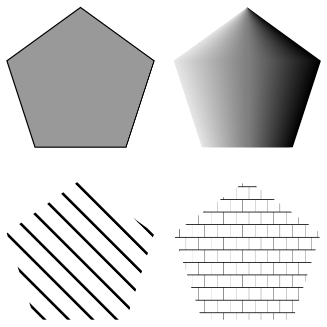
GL_LINE_STRIP



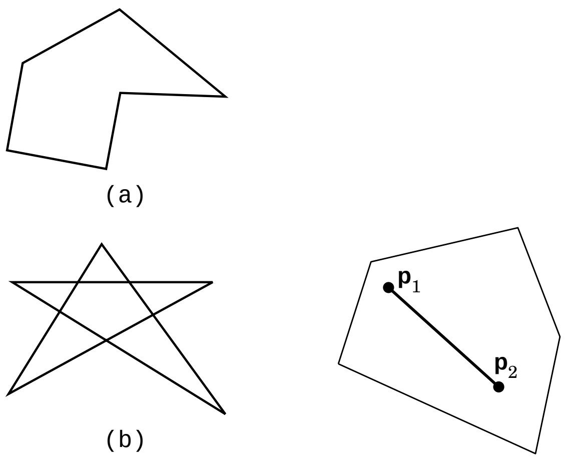
GL_LINE_LOOP

Polygons

Line segments and polylines can model the edges of objects. **Polygons** on the other hand have an interior, which can be **filled** with colour, pattern etc.

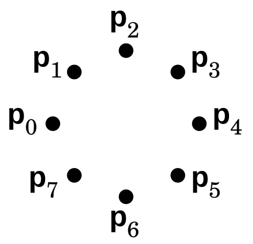


A 2D polygon is **simple** if no edges intersect at common endpoints. A 2D polygon is **convex** if the line segment joining any pair of points in the polygon lies inside the polygon. Non-simple and non-convex polygons are not always rendered correctly. In 3D only a triangle is really safe.

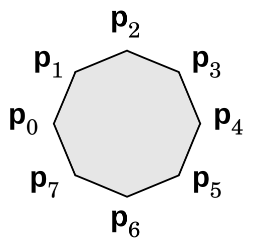


Polygon types:

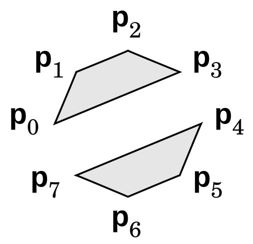
- `GL_POLYGON`: same edges as `GL_LOOP`
- `GL_TRIANGLES`: polygon with three sides
- `GL_QUADS`: polygon with four sides
- `GL_TRIANGLE_STRIP`: strip of triangles
- `GL_QUAD_STRIP`: strip of quadrilaterals
- `GL_TRIANGLE_FAN`: sequence of triangles sharing one vertex.



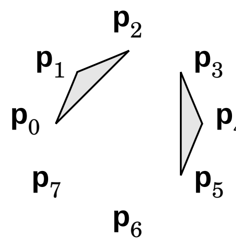
GL_POINTS



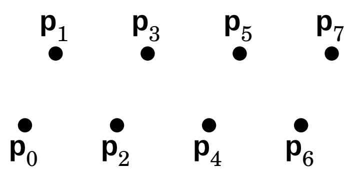
GL_POLYGON



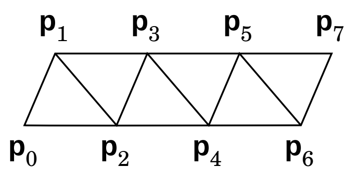
GL_QUADS



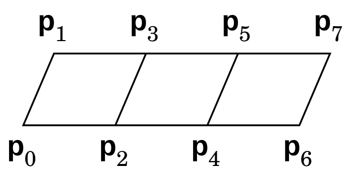
GL_TRIANGLES



GL_POINTS



GL_TRIANGLE_STRIP



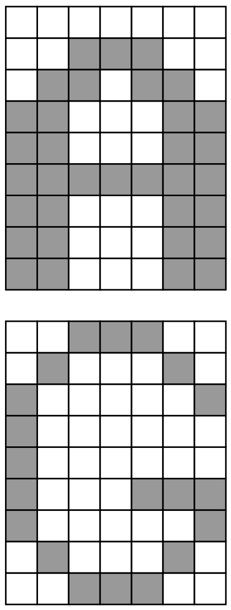
GL_QUAD_STRIP

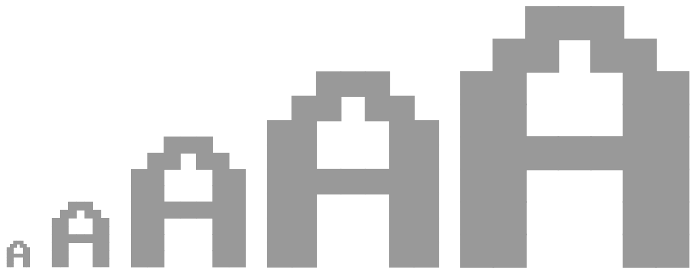
Curved objects

Curves and surfaces, such as spheres, can be tessellated, i.e. approximated by triangle meshes. Alternatively, some surfaces, such as Bezier surfaces, can be defined by *control points.*

Text comes in two kinds:

- **Stroke text** is defined by lines and curves which is later rasterized. The advantage is that it can be scaled and rotated without change of appearance.
- **Raster text** is defined by a block of bits. Its advantage is that it's faster to render than stroke text. A raster character can be placed in the frame buffer using a **block-bit-transfer**. However raster text does not scale properly and rotation is almost impossible.

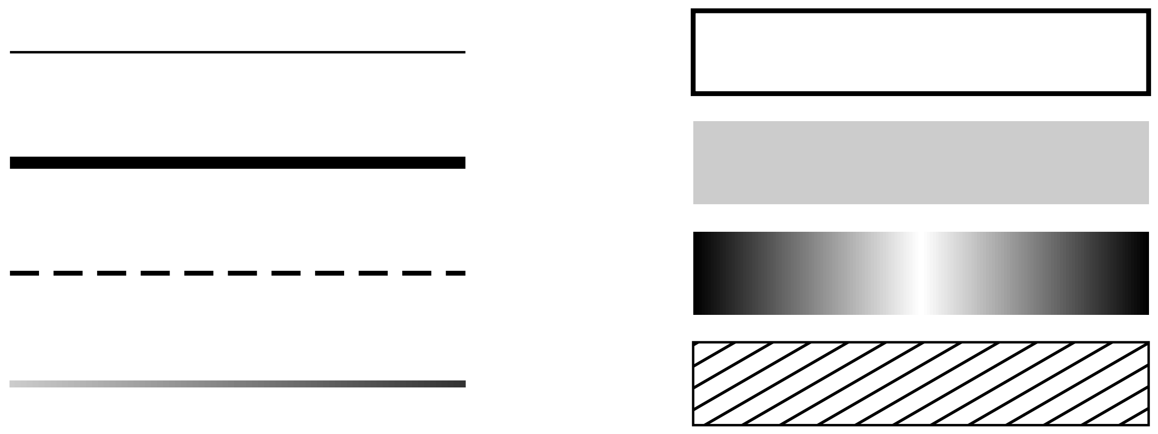




Text is positioned at the current **raster position**, controlled by the function `glRasterPos*`.

Attributes

Attributes could be colour, line thickness, or fill pattern. The present values of attributes are a part of the state of the graphics system. When a primitive is defined, the present attributes for that type are used, and it is displayed immediately. This is **immediate mode**. Primitives are not stored in the system (unless one uses 'display lists').



(a) (b)

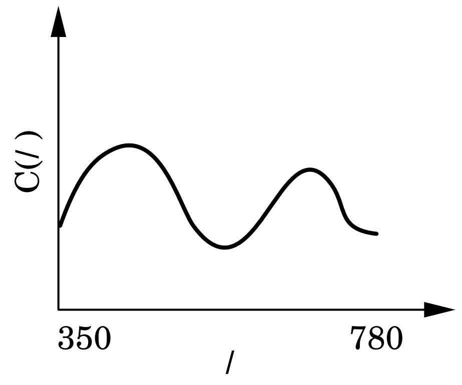
- **Points** have colour and size.
- **Line segments** have colour, thickness, and style (solid, dashed, dotted).
- **Filled primitives** have colour and pattern. We can display any combination of the interior and border.

RGB Colour

We usually use the **three-colour theory**. A colour C is taken to be a linear combination of **red** R, **green** G, and **blue** B, (**RGB**),

$$C = T_1 R + T_2 G + T_3 B.$$

The coefficients T_1, T_2, T_3 are the **tristimulus values**. This theory only approximates reality, in which a colour can be regarded as a function C giving the strength $C(\lambda)$ of each wavelength λ.

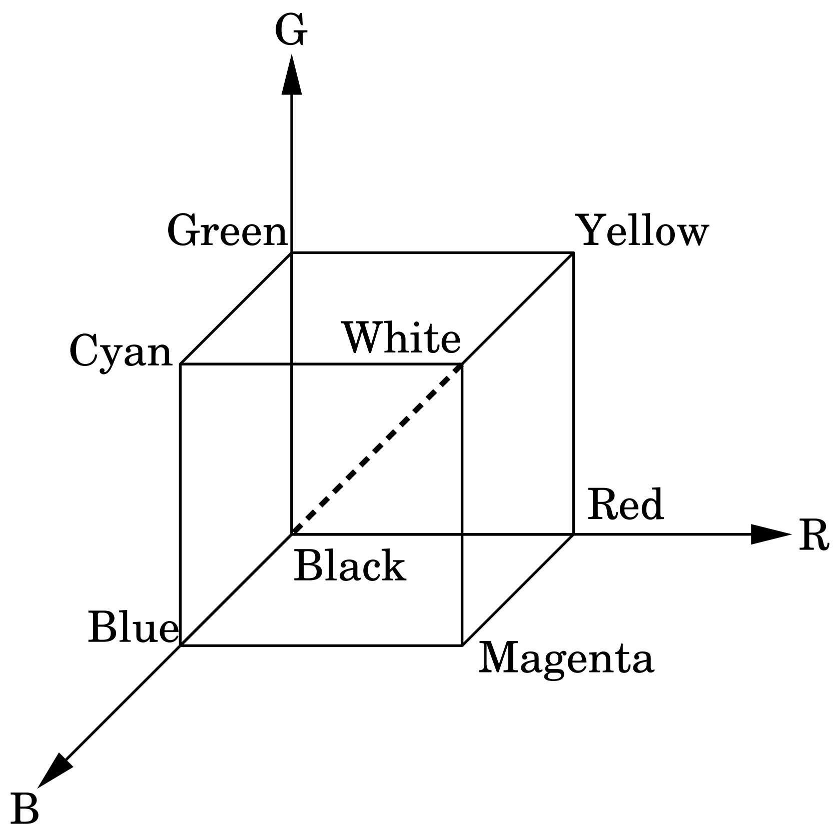


But in practice the brain interprets colour through a triple (A_1, A_2, A_3), where

$$A_i = \int S_i(\lambda) C(\lambda)\, d\lambda,$$

and so the approximation is quite good. Two colours that match visually are called a 'metameric pair', though they may not have the same colour distributions.

We can view an RGB colour as a point in a **colour solid**.



The RGB colour model is an **additive colour model**: we add combinations of red, green, and blue to a black background.

In a **subtractive colour model** we subtract combinations of colours from a white background (e.g. a sheet of paper). Subtracting **cyan**, **magenta**, and **yellow** (CMY), is equivalent to adding red, green, and blue. Cyan, magenta, and yellow are the **complementary colours** of red, green, and blue, respectively.

RGB colour model

Each pixel has red, green, and blue components, typically 8 bits each, 24 in total. In OpenGL we specify colour independent of hardware, so we use the colour cube. The following function sets red:

```
    glColor3f(1.0, 0.0, 0.0);
```

There is a fourth 'colour' in the frame buffer, the **alpha channel**. It is used to specify **opacity**, the oppositive of **transparency**. To clear a window with opaque white, we call

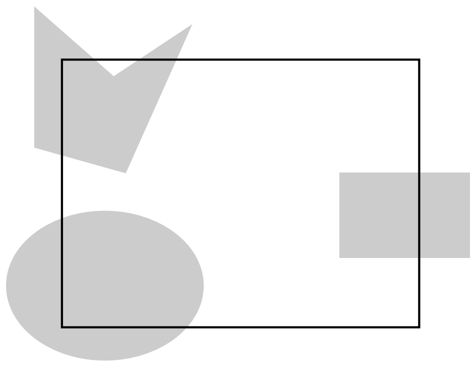
```
    glClearColor(1.0, 1.0, 1.0, 1.0);
```

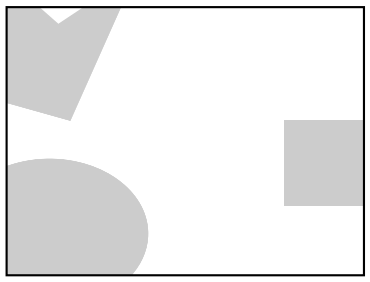
Indexed colour. A 24-bit-deep frame buffer with a resolution of 1280×1024 requires about 3 megabytes of memory (a lot). A way of avoiding this is to use a **colour look-up table**, specifying only a small range of colours among the $2^{24} \approx 16M$ colours.

Simple viewing

When viewing objects in 2D, we have only to specify the **viewing rectangle** or **clipping rectangle**, through the function

```
    gluOrtho2D(GLdouble left, GLdouble right,
               GLdouble bottom, GLdouble top);
```

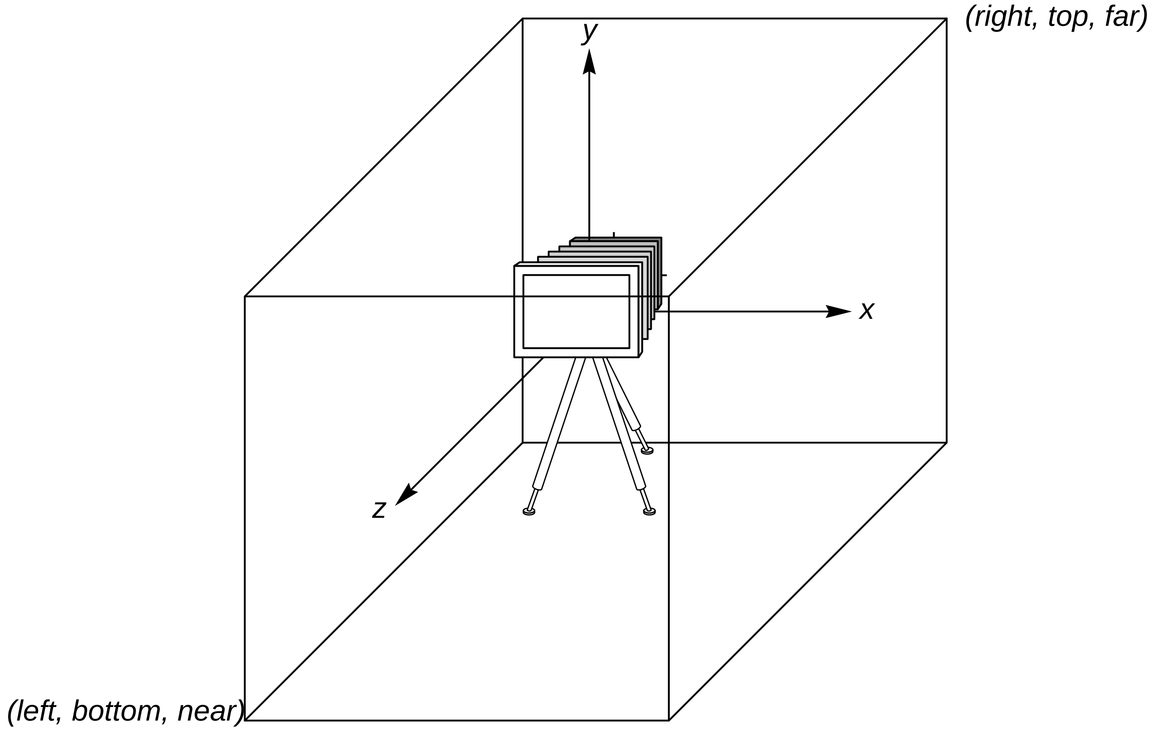

(a)


(b)

The function `gluOrtho2D` is a special case of the 3D viewing function

```
glOrtho(GLdouble left, GLdouble right, GLdouble bottom,
        GLdouble top, GLdouble near, GLdouble far);
```

which specifies the **viewing volume** and the use of **orthographic projection** for viewing in 3D. The default values are $-1, 1, -1, 1, -1, 1$. A point (x, y, z) is mapped to $(x, y, 0)$.



Transformation

Transformations are carried out by 4×4 matrices. The two most important are the **model-view** and **projection** matrices. At any time the state includes values for both. Their defaults are the identity matrix. The following sequence is a common way of setting a 2D viewing rectangle.

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0.0, 500.0, 0.0, 500.0);
glMatrixMode(GL_MODELVIEW);
```

At the end we switch the matrix mode back to model-view.

Control

Interaction with the window system and input devices depends on the specific environment, e.g.,

- X Windows on a unix platform,
- Windows on a PC.

GLUT provides a simple interface, independent of the window and operating systems. More sophisticted interfaces such as Qt provide customized menus, slide bars, widgets, etc.

A **window** is a rectangular part of the CRT screen or an equivalent part of the frame buffer. Positions within the window are measured in pixels, with $(0, 0)$ the bottom left hand corner. When positioning the window on the screen, $(0, 0)$ indicates the top left hand corner.

In addition to the size and position of the window on the screen, we can specify:

- RGB or indexed colour: `GLUT_RGB` or `GLUT_INDEX`,
- hidden-surface removal: `GLUT_DEPTH`,
- single or double buffering: `GLUT_SINGLE` or `GLUT_DOUBLE`.

The `main` and `myinit` functions for the Sierpinski Gasket:

```
int main(int argc, char** argv)
{
  glutInit(&argc,argv);
  glutInitDisplayMode(GLUT_SINGLE   GLUT_RGB);
  glutInitWindowSize(500,500);
  glutInitWindowPosition(0,0);
  glutCreateWindow("simple OpenGL example");
  glutDisplayFunc(display);
  myinit();
  glutMainLoop();
}
```

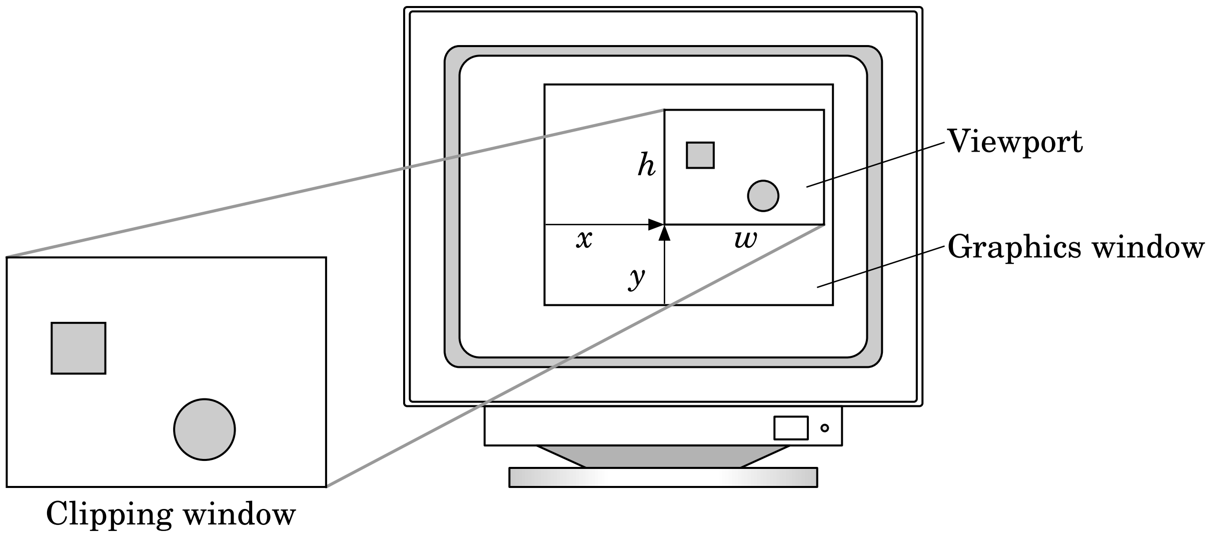
```
void myinit(void)
{
  glClearColor(1.0, 1.0, 1.0, 1.0);
  glColor3f(1.0, 0.0, 0.0);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluOrtho2D(0.0, 500.0, 0.0, 500.0);
  glMatrixMode(GL_MODELVIEW);
}
```

Aspect Ratio and Viewports

The aspect ratio of a rectangle is the ratio of its width to its height. If the aspect ratios of the viewing rectangle specified by `glOrtho` and the window specified by `glutInitWindowSize` are not the same, the image will be distorted. The solution is to use a **viewport**, which is an arbitrary rectangular region of the window. We can ensure that the aspect ratio of the viewport is the same as that of the viewing rectangle. We call

```
glViewport(GLint x, GLint y, GLsizei w, GLsizei h);
```

where (x, y) is the position of the lower-left corner of the viewport and w and h are its width and height.



Viewport
Graphics window
h
x
w
y
Clipping window