

# Letterkenny Institute of Technology

---

## MSc in Computing (Games Development)

---

**Subject:** 2D and 3D Graphics

**Level:** 9

**Date:** May 2007

**Examiners:**

Dr. J.G. Campbell

Dr. M.D.J. McNeill

**Time Allowed:** Three hours

---

### INSTRUCTIONS

Answer **four** questions from six.

---

1. (a) The following often appears in the `main` of an OpenGL program; explain precisely what is being specified / enabled.

```
glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
```

[6 marks]

- (b) After *projection* (to 2-D), you might think that z-axis information may be discarded; however, there often is a need for it. Why?

[2 marks]

- (c) The code in Figure 3 draws the figures shown in Figures 1 and 2; in Figure 1 `glShadeModel (GL_SMOOTH)`; is used and in Figure 2 `glShadeModel (GL_FLAT)`; is used.

Explain the difference.

[2 marks]

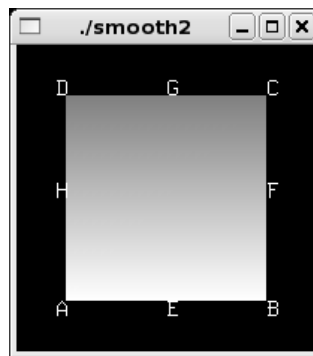


Figure 1: Rectangular figure — smooth shading

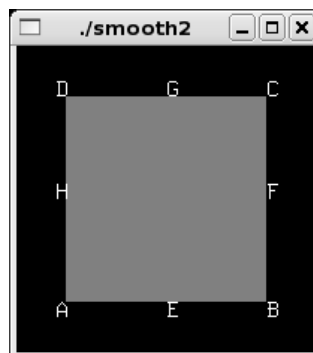


Figure 2: Rectangular figure — flat shading.

- (d) In Figure 2 (`glShadeModel (GL_FLAT)`), the colours at points A to H are all the same. What is that colour? Answer with an RGB triple.

[2 marks]

- (e) In Figure 1 (`glShadeModel (GL_SMOOTH)`), the colour gradually changes from bottom to top. Give the RGB triple for each of points A to H; six of them are obvious, two need calculation.

[4 marks]

```

void init(void) {
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glShadeModel (GL_SMOOTH); // 1
    glShadeModel (GL_FLAT); // 2
}

void rectangle(void){
    glBegin (GL_QUADS);
    glColor3f (1.0, 1.0, 1.0);
    glVertex3f (5.0, 5.0, 0.0);
    glVertex3f (25.0, 5.0, 0.0);
    glColor3f (0.5, 0.5, 0.5);
    glVertex3f (25.0, 25.0, 0.0);
    glVertex3f (5.0, 25.0, 0.0);
    glEnd();

void display(void){
    glClear (GL_COLOR_BUFFER_BIT);
    rectangle();
    glFlush ();
}

void reshape (int w, int h){
    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    if (w <= h)
        glOrtho(0.0, 30.0, 0.0, 30.0 * (GLfloat) h/(GLfloat) w, -1.0, 1.0);
    else
        glOrtho(0.0, 30.0 * (GLfloat) w/(GLfloat) h, 0.0, 30.0, -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
}

int main(int argc, char** argv){
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize (200, 200);
    glutInitWindowPosition (100, 100);
    glutCreateWindow (argv[0]);
    init ();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutKeyboardFunc (keyboard);
    glutMainLoop();
    return 0;
}

```

Figure 3: Smooth1.cpp

(f) Why might you prefer the geometric primitive `GL_TRIANGLE_STRIP` to `GL_TRIANGLES`? See Appendix F.

[2 marks]

(g) When using the geometric primitive `GL_TRIANGLE_STRIP`, OpenGL draws the triangles using vertices  $v_0, v_1, v_2$  and then  $v_2, v_1, v_3$ , then  $v_2, v_3, v_4$  and so on. Why this ordering? Refer to Appendix F as necessary.

[2 marks]

(h) Taking account of the statement in (g), rewrite `display` in Figure 3 using `GL_TRIANGLE_STRIP` instead of `GL_QUADS`; ignore colour. Include a diagram clearly showing your choice of vertices; label the vertices in the diagram and in the code, so that it is clear what is being attempted.

[5 marks]

2. (a) Roughly speaking, an *affine space* contains *vectors* and *points* and certain operations on them; in contrast a *vector space* contains just *vectors* and operations on the vectors. Outline the role of *affine spaces* in computer graphics.

[4 marks]

- (b) What is the most general form of an *affine transformation*? Use either 2-D or 3-D examples to illustrate your answer. Hint: it may be useful to contrast with *vector / linear transformation* and to include mention of the implementation of vector transformation as matrix multiplication.

[4 marks]

- (c) Explain how *homogeneous coordinates* may be used to allow general affine transformations to be implemented using matrix multiplication. In addition, explain **two** significant benefits of the latter in computer graphics / games / virtual reality.

[6 marks]

- (d) Eqn. 1 shows a 2-D *translation* transformation (call the matrix  $T$ ) and eqn. 2 shows a 2-D *rotation* transformation (call the matrix  $R$ ).

$$\begin{bmatrix} v_x \\ v_y \\ v_w \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_x \\ u_y \\ u_w \end{bmatrix}, \quad (1)$$

$$\begin{bmatrix} v_x \\ v_y \\ v_w \end{bmatrix} = \begin{bmatrix} \cos a & -\sin a & 0 \\ \sin a & \cos a & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_x \\ u_y \\ u_w \end{bmatrix}. \quad (2)$$

- (i) Compute the product (composition of matrices)  $TR$ ; call the result  $S$ .

[2 marks]

- (ii) Show that the transformation  $\mathbf{v} = \mathbf{S}\mathbf{u}$ , where  $S = TR$ , is equivalent to the two-step transformation:  $\mathbf{q} = \mathbf{R}\mathbf{u}$ ,  $\mathbf{v} = \mathbf{T}\mathbf{q}$ .

[4 marks]

**Note.** If you think it makes the answer easier, you may use concrete numbers in your answer, for example  $a = 30^\circ$ ,  $t_x = 4$ ,  $t_y = 6$ ,  $u_x = 2$ ,  $u_y = 3$ ,  $u_w = 1$ .

cont'd ...

(e) Eqn. 3 shows a general 3-D affine transformation in homogeneous coordinates form.

$$\begin{bmatrix} v_x \\ v_y \\ v_z \\ v_w \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & t_x \\ a_{21} & a_{22} & a_{23} & t_y \\ a_{31} & a_{32} & a_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_x \\ u_y \\ u_z \\ u_w \end{bmatrix}. \quad (3)$$

Internally, OpenGL stores points (vertices) and vectors as 4-D homogeneous coordinates; normally, points/vertices have a  $w$  value of 1, or some non-zero value; vectors can be represented by having a  $w$  value of 0; an OpenGL example of a vector is when we specify a *directional* light, for example, a light far away at direction (1, 1, 1)

```
GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
```

Show that eqn. 3 has the desired effect of both (i) points and (ii) vectors, i.e. vectors are not translated but suffer only the effects of the  $a_{ij}$  components, but points suffer the effects of both the  $a_{ij}$  components and the translation components.

[5 marks]

3. (a) Figure 4 shows the transformation stages that vertices undergo in an OpenGL rendering pipeline.

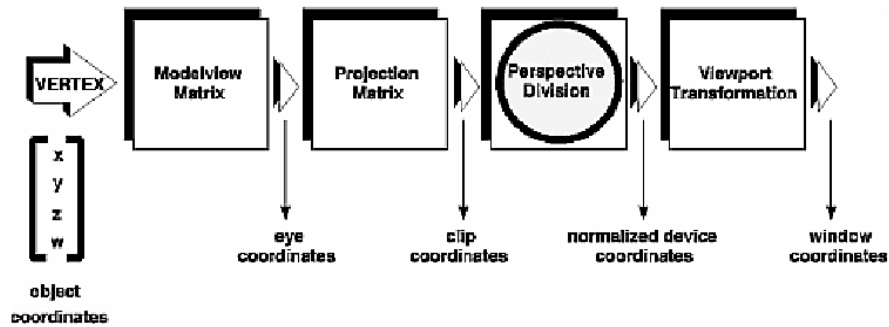


Figure 4: Vertex transformation pipeline.

Explain the terms *modelling* and *viewing* and why OpenGL is able to combine *modelling* and *viewing* transformations in the same `GL_MODELVIEW` transformation matrix.

[4 marks]

- (b) The final transformation in Figure 4 is the *viewport* transformation; using a diagram or otherwise, give a precise explanation of the purpose of this transformation.

[3 marks]

- (c) Referring as necessary to Figures 16 and 17 in Appendix D, show that the following call to `gluPerspective` is equivalent to the call to `glFrustum`. Figure 5 may be of assistance.

[2 marks]

```
tanBy2 = tan(fovY/2.0);
gluPerspective(fovY, aspectRatio, n, 20.0);
// glFrustum (-n*tanBy2*aspectRatio, n*tanBy2*aspectRatio,
             -n*tanBy2, n*tanBy2, n, 20.0);
```

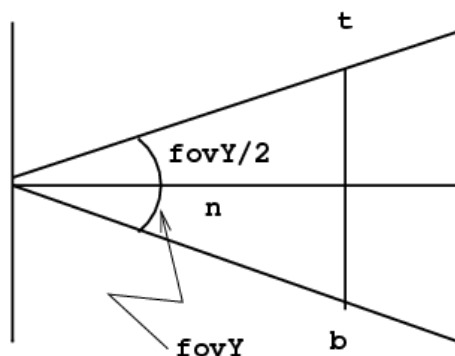


Figure 5: Field of View.

- (d) The code in Figure 6 draws the figures shown in Figure 7 and emits the output in Figure 8.
- (i) Explain the fourth column of matrix MV1; in addition, explain why matrix MV4 is the same as matrix MV1. [3 marks]
- (ii) If we replace  
`glTranslated(-2.0, -3.0, -12.0);`  
with  
`gluLookAt(eyeX, eyeY, eyeZ, 0.0, 0.0, atZ, 0.0, 1.0, 0.0);`  
what values should be in `eyeX`, `eyeY`, `eyeZ`, and `atZ`? Hint: In answering about `atZ`, think carefully about the default direction of the camera, when `gluLookAt` is not called; [3 marks]
- (iii) Explain the contents of matrix MV3; refer as necessary to the contents of matrix MV2; [4 marks]
- (iv) We have removed parts of matrix MV6, write down, with brief explanation the values that will be in A, B, C, D. [3 marks]
- (v) We have removed all of matrix MV8, write down, with brief explanation, the values that will be in matrix MV8. [3 marks]

```

void display(void){
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity ();
    matPrint(GL_MODELVIEW_MATRIX, "MV0");
    //gluLookAt(eyeX, eyeY, eyeZ, 0.0, 0.0, atZ, 0.0, 1.0, 0.0);
    glTranslated(-2.0, -3.0, -12.0);
    matPrint(GL_MODELVIEW_MATRIX, "MV1");

    drawAxes(-4., 2.8, -4., 2.8, -4., 4.5);

    house();

    glPushMatrix();
    glTranslated(4.0, 5.0, 0.0);
    matPrint(GL_MODELVIEW_MATRIX, "MV2");
    glRotated(50.0, 1.0, 0.0, 0.0);
    matPrint(GL_MODELVIEW_MATRIX, "MV3");
    house();
    glPopMatrix();

    matPrint(GL_MODELVIEW_MATRIX, "MV4");
    glPushMatrix();
    glTranslated(5.0, 0.0, 0.0);
    glRotated(30.0, 0.0, 1.0, 0.0);
    matPrint(GL_MODELVIEW_MATRIX, "MV5");
    glRotated(15.0, 0.0, 1.0, 0.0);
    matPrint(GL_MODELVIEW_MATRIX, "MV6");
    house();
    glPopMatrix();

    matPrint(GL_MODELVIEW_MATRIX, "MV7");
    glPushMatrix();
    glTranslated(0.0, 5.0, 0.0);
    glScaled(1.5, 2.0, 0.5);
    matPrint(GL_MODELVIEW_MATRIX, "MV8");
    house();
    glPopMatrix();

    glFlush();
}

```

Figure 6: House code

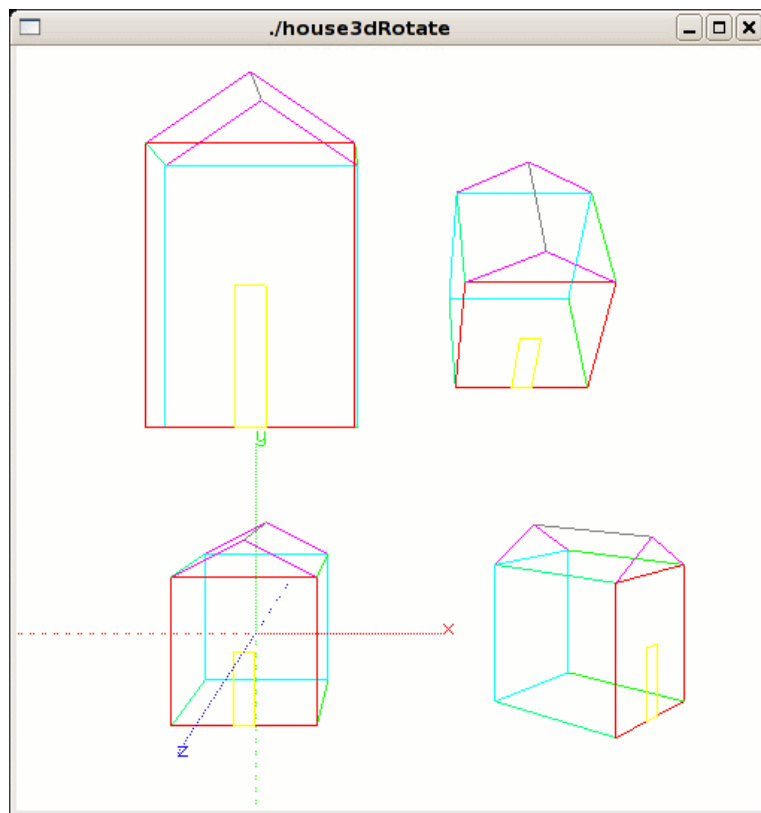


Figure 7: House figures.

MV0  
 [( 1, 0, 0, 0)  
 ( 0, 1, 0, 0)  
 ( 0, 0, 1, 0)  
 ( 0, 0, 0, 1)]

MV1  
 [( 1, 0, 0, -2)  
 ( 0, 1, 0, -3)  
 ( 0, 0, 1, -12)  
 ( 0, 0, 0, 1)]

MV2  
 [( 1, 0, 0, 2)  
 ( 0, 1, 0, 2)  
 ( 0, 0, 1, -12)  
 ( 0, 0, 0, 1)]

MV3  
 [( 1, 0, 0, 2)  
 ( 0, 0.6428, -0.7661, 2)  
 ( 0, 0.7661, 0.6428, -12)  
 ( 0, 0, 0, 1)]

MV4  
 [( 1, 0, 0, -2)  
 ( 0, 1, 0, -3)  
 ( 0, 0, 1, -12)  
 ( 0, 0, 0, 1)]

MV5  
 [(0.866, 0, 0.5, 3)  
 ( 0, 1, 0, -3)  
 ( -0.5, 0, 0.866, -12)  
 ( 0, 0, 0, 1)]

MV6  
 [( A, 0, B, 3)  
 ( 0, 1, 0, -3)  
 ( C, 0, D, -12)  
 ( 0, 0, 0, 1)]

MV7  
 [( 1, 0, 0, -2)  
 ( 0, 1, 0, -3)  
 ( 0, 0, 1, -12)  
 ( 0, 0, 0, 1)]

MV8  
 ???

Figure 8: Modelview matrices.

4. Eqn. 4 summarises the version of the Phong lighting model used by OpenGL,

$$I_{total} = m_e + \sum_{i=1}^{N_l} sa^i att^i [I_a^i m_a + I_d^i m_d \max(\mathbf{N} \cdot \mathbf{L}^i, 0) + I_s^i m_s \max((\mathbf{V} \cdot \mathbf{R}^i), 0)^s]. \quad (4)$$

$\mathbf{N}$  is the normal to the surface at the vertex in question;  $\mathbf{L}^i$  is the direction of light  $i$ ;  $\mathbf{V}$  is the viewing (eye) direction and  $\mathbf{R}^i$  is the specular reflection direction for light  $i$ ;  $sa$  is the spotlight attenuation factor and  $att$  distance attenuation;  $\mathbf{R}$  is given by eqn. 5.

$$\mathbf{R} = 2(\mathbf{N} \cdot \mathbf{L})\mathbf{N} - \mathbf{L}. \quad (5)$$

(a) Explain how *colour* enters eqn. 4

[4 marks]

(b) Based on eqn. 4, explain the following types of lighting, source, and material:

- (i) *Ambient*;
- (ii) *Diffuse*;
- (iii) *Specular*;
- (iv) *Emissive*.

[10 marks]

(There are  $4 \times 2$  marks for (i) to (iv), *plus* 2 additional marks for a diagram showing the vectors mentioned in eqn. 4.)

(c) Figure 9 shows some OpenGL code involving lighting and Figure 10 shows the resulting graphic.

Explain lines //A, //B, and //C.

[6 marks]

Part 4(d) continued on page 15.

```

void display(void){
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity ();
    glTranslated(-2.0, -3.0, -12.0);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_COLOR_MATERIAL); //A
    glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE); //B
    GLfloat l_amb[] = { 0.2, 0.2, 0.2, 1.0 };
    GLfloat l_dif[] = { 0.2, 1.0, 0.2, 1.0 };
    GLfloat l_spc[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat m_spc[] = { 1.0, 1.0, 1.0, 1.0 };
    //GLfloat m_spc[] = { 0.0, 0.0, 0.0, 1.0 };
    GLfloat m_shn[] = { 5.0 };

    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, l_amb); //C
    GLfloat l_pos[] = { 20.0, 20.0, 20.0, 0.0 };
    glLightfv (GL_LIGHT0, GL_POSITION, l_pos);
    glLightfv (GL_LIGHT0, GL_DIFFUSE, l_dif);
    glLightfv (GL_LIGHT0, GL_SPECULAR, l_spc);

    glMaterialfv(GL_FRONT, GL_SPECULAR, m_spc);
    glMaterialfv(GL_FRONT, GL_SHININESS, m_shn);

    house();

    glPushMatrix();
    glTranslated(4.0, 5.0, 0.0);
    glRotated(50.0, 1.0, 0.0, 0.0);
    glutSolidTeapot(1.0);
    glPopMatrix();

    glPushMatrix();
    glTranslated(5.0, 0.0, 0.0);
    glRotated(30.0, 0.0, 1.0, 0.0);
    house();
    glPopMatrix();

    glPushMatrix();
    glTranslated(0.0, 5.0, 0.0);
    glScaled(1.0, 1.0, 1.0);
    house();
    glPopMatrix();

```

Figure 9: Lighted house code

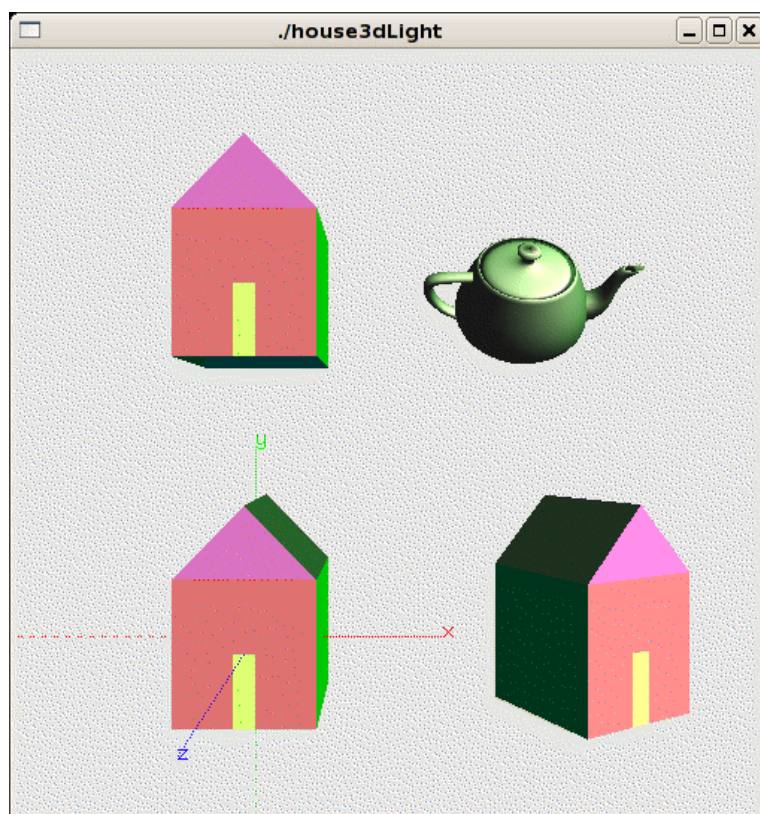


Figure 10: House figures.

- (d) Given the code fragment below, explain what will be the colour /shade at vertex `vf[0]`. Assume that material has zero specular reflection; consider ambient and diffuse only; assume that the light direction is in the direction of the normal; **no attenuation**. You should carefully explain your calculations.

[5 marks]

```
GLfloat l_amb[] = { 0.2, 0.3, 0.8, 1.0 };
GLfloat l_dif[] = { 0.4, 0.5, 0.9, 1.0 };
GLfloat l_spc[] = { 1.0, 1.0, 1.0, 1.0 };
//GLfloat m_spc[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat m_spc[] = { 0.0, 0.0, 0.0, 1.0 };
GLfloat m_shn[] = { 5.0 };
GLfloat m_amb[] = { 0.5, 0.5, 0.5, 1.0 };
GLfloat m_dif[] = { 0.8, 0.8, 0.8, 1.0 };

GLfloat l_pos[] = { 20.0, 20.0, 20.0, 0.0 };
glLightfv (GL_LIGHT0, GL_POSITION, l_pos);
glLightfv (GL_LIGHT0, GL_DIFFUSE, l_dif);
glLightfv (GL_LIGHT0, GL_AMBIENT, l_amb);
glLightfv (GL_LIGHT0, GL_SPECULAR, l_spc);

glMaterialfv(GL_FRONT, GL_SPECULAR, m_spc);
glMaterialfv(GL_FRONT, GL_SHININESS, m_shn);
glMaterialfv(GL_FRONT, GL_AMBIENT, m_amb);
glMaterialfv(GL_FRONT, GL_DIFFUSE, m_dif);

glVertex3fv(vf[0]);
```

5. (a) Antialiasing. Assume that Figure 11 shows an intended **black** (value 0) line on a **white** (value 1) background, with one pixel occupying each of the ruled squares.
- (i) Draw a rough diagram to show how the line will look *without* antialiasing applied. [2 marks]
- (ii) Assuming now that antialiasing is applied, based on pixel coverage, write down the (grey level, 0 ... 1) values that will be found in the twelve (12) pixels indicated A1, A1, ... C4. Give a brief explanation of how you arrived at your answers. **Note again:** the line is *black* (grey level 0) on a *white* (grey level 1) background. [5 marks]

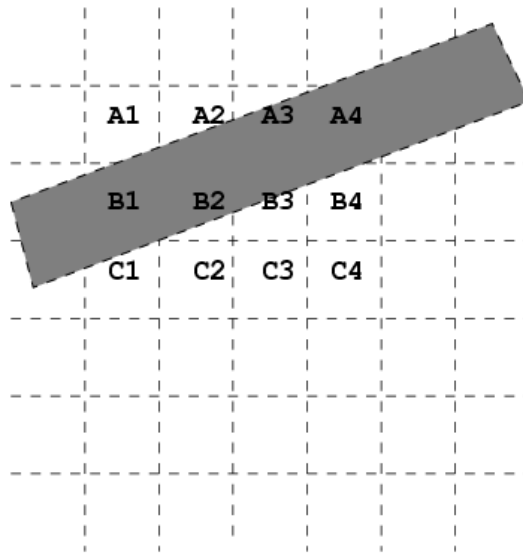


Figure 11: Antialiasing.

- (b) Blending.
- (i) *Without* `glEnable(GL_DEPTH_TEST);` and `glEnable(GL_BLENDING);`, how does OpenGL choose between two potential occupants of a (rendered) pixel;
- (ii) *With* `glEnable(GL_DEPTH_TEST);`, but *without* `glEnable(GL_BLENDING);`, how does OpenGL choose between two potential occupants of a (rendered) pixel.
- (iii) What is ALPHA in RGBA?

[4 marks]

- (c) Blending. OpenGL blends by accumulating colour, from what it calls the *source* ( $C_s$ ), into the current value in display buffer, which it calls the *destination* ( $C_d$ ). It adds  $C_s$  to  $C_d$  and assigns the result to  $C_d$ , i.e.  $C_d = C_sS + C_dD$ , where  $S$  and  $D$  are blending factors chosen by `glBlendFunc`. The following fragment gives a brief indication of the use of `glBlendFunc`:

```
glEnable (GL_BLEND);
glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
//          source weight S destination weight D
//glBlendFunc (GL_ONE, GL_ONE_MINUS_SRC_ALPHA);
```

- (i) In the case of the following call to `glBlendFunc`, how will source and destination colours be blended? Mention the case where source-ALPHA is 0.
- ```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

[3 marks]

- (ii) Explain how

```
glBlendFunc (GL_ONE, GL_ONE_MINUS_SRC_ALPHA);
```

effectively removes blending when the *alpha* of the *source* is 1.

[2 marks]

- (d) Consider the following code applied to a single vertex,

```
glEnable (GL_BLEND);
glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
//          source(S) destination(D)
glClearColor (0.0, 0.0, 0.0, 0.0);
glClear(GL_COLOR_BUFFER_BIT);

glColor4f(1.0, 1.0, 0.0, 0.75); //A

glColor4f(0.0, 1.0, 1.0, 0.75); //B

glColor3f(0.75, 0.75+0.25*0.75, 0.25*0.75); //C
```

- (i) What will be the RGBA *after* //A has been executed? Hint:

$$C_d = (R_d, G_d, B_d, A_d) = (R_sS + R_dD, G_sS + G_dD, B_sS + B_dD, A_sS + A_dD). \quad (6)$$

[5 marks]

- (ii) Show that after //A and then //B have been executed, we produce the *same* RGB colour as if //C had been executed alone.

[4 marks]

6. Bézier curves.

The four Bernstein polynomial blending functions,  $B_0^3, B_1^3, B_2^3$ , and  $B_3^3$ , upon which Cubic Bézier curves are based are shown in Figure 12, along with linear blending functions.

$$B_0^3(u) = u^3, \quad (7)$$

$$B_1^3(u) = 3u^2(1-u), \quad (8)$$

$$B_2^3(u) = 3u(1-u)^2, \quad (9)$$

$$B_3^3(u) = (1-u)^3. \quad (10)$$

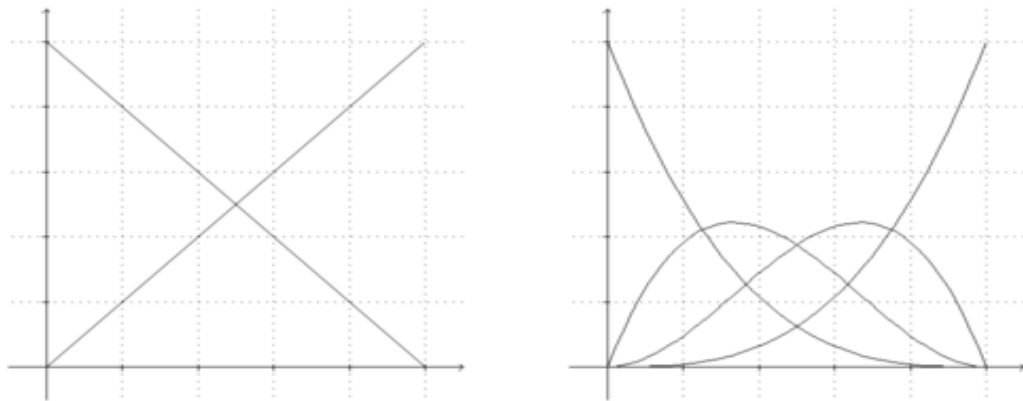


Figure 12: Blending functions. Linear (left) and cubic Bernstein (Bézier) (right).

```

GLfloat ctrlpoints[4][3] = {
    { -4.0, -4.0, 0.0}, { -2.0, 4.0, 0.0},
    {2.0, -4.0, 0.0}, {4.0, 4.0, 0.0}};

void init(void){
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glShadeModel(GL_FLAT);
    glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, &ctrlpoints[0][0]);
    glEnable(GL_MAP1_VERTEX_3);
}

void display(void){
    int i;

    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 1.0, 1.0);
    glBegin(GL_LINE_STRIP);
        for (i = 0; i <= 30; i++)
            glEvalCoord1f((GLfloat) i/30.0);
    glEnd();
    /* The following code displays the control points as dots. */
    glPointSize(5.0);
    glColor3f(1.0, 1.0, 0.0);
    glBegin(GL_POINTS);
        for (i = 0; i < 4; i++)
            glVertex3fv(&ctrlpoints[i][0]);
    glEnd();
    glFlush();
}

```

Figure 13: Bézier curve, bezcurve.c

- (a) Discuss Cubic Bézier curves and their use in OpenGL evaluators as shown in the program in Figure 13. You must include a precise description on how  $B_0^3$ ,  $B_1^3$ ,  $B_2^3$ , and  $B_3^3$  are used in function `glEvalCoord1f` to compute the curve. The graphic produced by Figure 13 is shown in Figure 14

[10 marks]

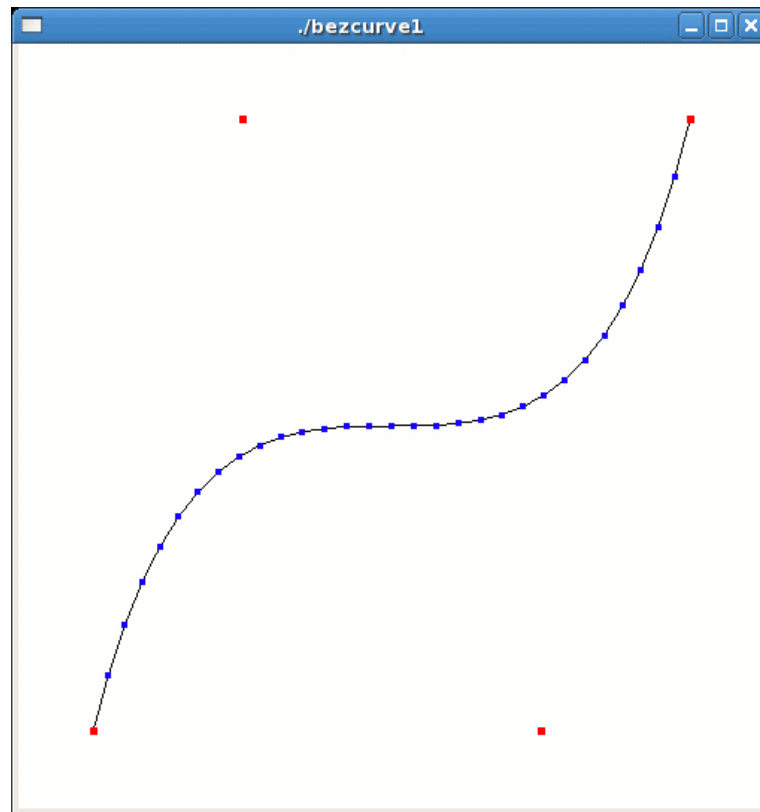


Figure 14: Bézier curve.

(b) For the control points given below,

```
GLfloat ctrlpoints[4][3] = {  
    { -4.0, -4.0, 0.0}, { -2.0, 4.0, 0.0},  
    {2.0, -4.0, 0.0}, {4.0, 4.0, 0.0}};
```

show that the Bézier curve correctly passes through the end points  $(-4.0, -4.0)$ , hint:  $u = 0$ , and  $(4.0, 4.0)$ , hint:  $u = 1$ .

[5 marks]

(c) Show that halfway along the curve ( $u = 0.5$ ), it passes through  $(x = 0, y = 0)$ ; ignore  $z$  values; it will suffice to calculate the  $x$  value.

[5 marks]

(d) Discuss *linear interpolation* and how it could be implemented using the linear blending functions shown in Figure 12.

[5 marks]

## Appendix A. Trigonometry.

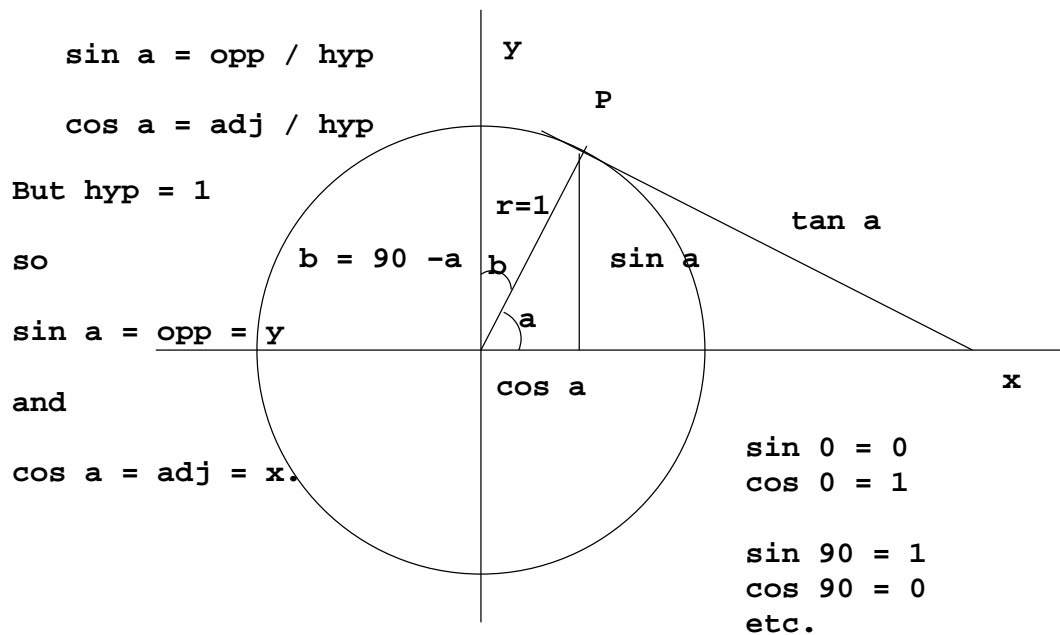


Figure 15: Sin, cos and circle.

From Figure 15 we can see from Pythagoras's Theorem that

$$\sin^2 \theta + \cos^2 \theta = 1. \quad (11)$$

Some values.  $\sin 0^\circ = 0$ ,  $\sin 30^\circ = 0.5$ ,  $\sin 60^\circ = \sqrt{3}/2 = 0.866$ ,  $\sin 90^\circ = 1$ .  
 $\cos 0^\circ = 1$ ,  $\cos 30^\circ = \sqrt{3}/2 = 0.866$ ,  $\cos 60^\circ = 0.5$ ,  $\cos 90^\circ = 0$ .

### Radians.

A radian is about  $57^\circ$ ; it is the angle subtended by an arc of length  $r$  on a circle of radius  $r$ .

$\pi/2$  radians =  $90^\circ$ ,  $\pi$  radians =  $180^\circ$ ,  $2\pi$  radians =  $360^\circ$ , etc.

$\pi = 3.141592635 \dots$

Degrees,  $d$ , to radians,  $r$ :  $r = (180/\pi) \times d$ .

### Cos and Sin are periodic over $2\pi$ radians or $360^\circ$ .

cos and sin repeat themselves after  $2\pi$  radians or  $360^\circ$ .

Sin is an odd function:  $\sin -\theta = -\sin \theta$ .

Cos is an even function:  $\cos -\theta = \cos \theta$ .

### Useful equations.

$$\sin(\theta + \phi) = \sin \theta \cos \phi + \cos \theta \sin \phi, \quad (12)$$

$$\sin(\theta - \phi) = \sin \theta \cos \phi - \cos \theta \sin \phi, \quad (13)$$

$$\cos(\theta + \phi) = \cos \theta \cos \phi - \sin \theta \sin \phi, \quad (14)$$

$$\cos(\theta - \phi) = \cos \theta \cos \phi + \sin \theta \sin \phi. \quad (15)$$

$$\sin^2 \theta + \cos^2 \theta = 1. \quad (16)$$

$$\sin^2 \theta = \frac{1}{2}(1 - \cos 2\theta). \quad (17)$$

$$\cos^2 \theta = \frac{1}{2}(1 + \cos 2\theta). \quad (18)$$

$$\cos 2\theta = \cos^2 \theta - \sin^2 \theta = 1 - \sin^2 \theta = 2 \cos^2 \theta - 1. \quad (19)$$

## Appendix B. 2D Linear Transformations.

### Scaling

$$\begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} u_x \\ u_y \end{bmatrix}. \quad (20)$$

### Rotation

$$\begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{bmatrix} \cos b & -\sin b \\ \sin b & \cos b \end{bmatrix} \begin{bmatrix} u_x \\ u_y \end{bmatrix}. \quad (21)$$

### Shear

 Shear along the x axis,

$$\begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_x \\ u_y \end{bmatrix}. \quad (22)$$

Shear along the y axis,

$$\begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ b & 1 \end{bmatrix} \begin{bmatrix} u_x \\ u_y \end{bmatrix}. \quad (23)$$

### Reflection

 Reflect about the y axis (x-coordinates are negated),

$$\begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_x \\ u_y \end{bmatrix}. \quad (24)$$

Reflect about the x axis (y-coordinates are negated),

$$\begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} u_x \\ u_y \end{bmatrix}. \quad (25)$$

### Projection

 Projection onto x-axis,

$$\begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} u_x \\ u_y \end{bmatrix}. \quad (26)$$

Projection onto the y-axis,

$$\begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_x \\ u_y \end{bmatrix}. \quad (27)$$

### Translation

$$\begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{bmatrix} t_x \\ t_y \end{bmatrix} + \begin{bmatrix} u_x \\ u_y \end{bmatrix}. \quad (28)$$

## Appendix C. 3D Affine Transformations using Homogeneous Coordinates.

### Translation

$$\begin{bmatrix} v_x \\ v_y \\ v_z \\ v_w \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_x \\ u_y \\ u_z \\ u_w \end{bmatrix}. \quad (29)$$

### Rotation about the z-axis

$$R_z(b) = \begin{bmatrix} \cos b & -\sin b & 0 & 0 \\ \sin b & \cos b & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (30)$$

### Rotation about the x-axis

$$R_x(b) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos b & -\sin b & 0 \\ 0 & \sin b & \cos b & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (31)$$

### Rotation about the y-axis

$$R_y(b) = \begin{bmatrix} \cos b & 0 & \sin b & 0 \\ 0 & 1 & 0 & 0 \\ -\sin b & 0 & \cos b & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (32)$$

## Appendix D. 3D Projection Transformations using Homogeneous Coordinates.

### Perspective Transformation

$$\begin{bmatrix} -x'p_z \\ -y'p_z \\ -z'p_z \\ w' \end{bmatrix} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2nf}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}. \quad (33)$$

```
void glFrustum(GLdouble left, GLdouble right,
               GLdouble bottom, GLdouble top,
               GLdouble near, GLdouble far).
```

Figure 16 shows what the arguments mean.

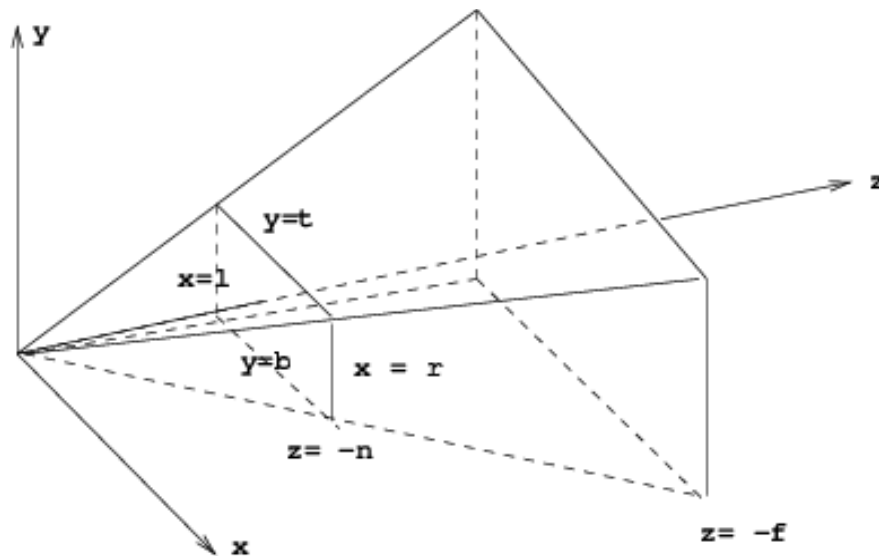


Figure 16: Perspective view frustum volume defined by glFrustum.

gluPerspective is more intuitive to use:

```
gluPerspective(float fovAngleY, float aspectRatio ,
               float near, float far);

// these two are equivalent
tanBy2 = tan(fovY/2.0);
gluPerspective(fovY, aspectRatio, n, 20.0);
// glFrustum (-n*tanBy2, n*tanBy2, -n*tanBy2*aspectRatio,
               n*tanBy2*aspectRatio, n, 20.0);
```

Figure 17 shows what the arguments mean. Compare with the equivalent glFrustum diagram, Figure 16.

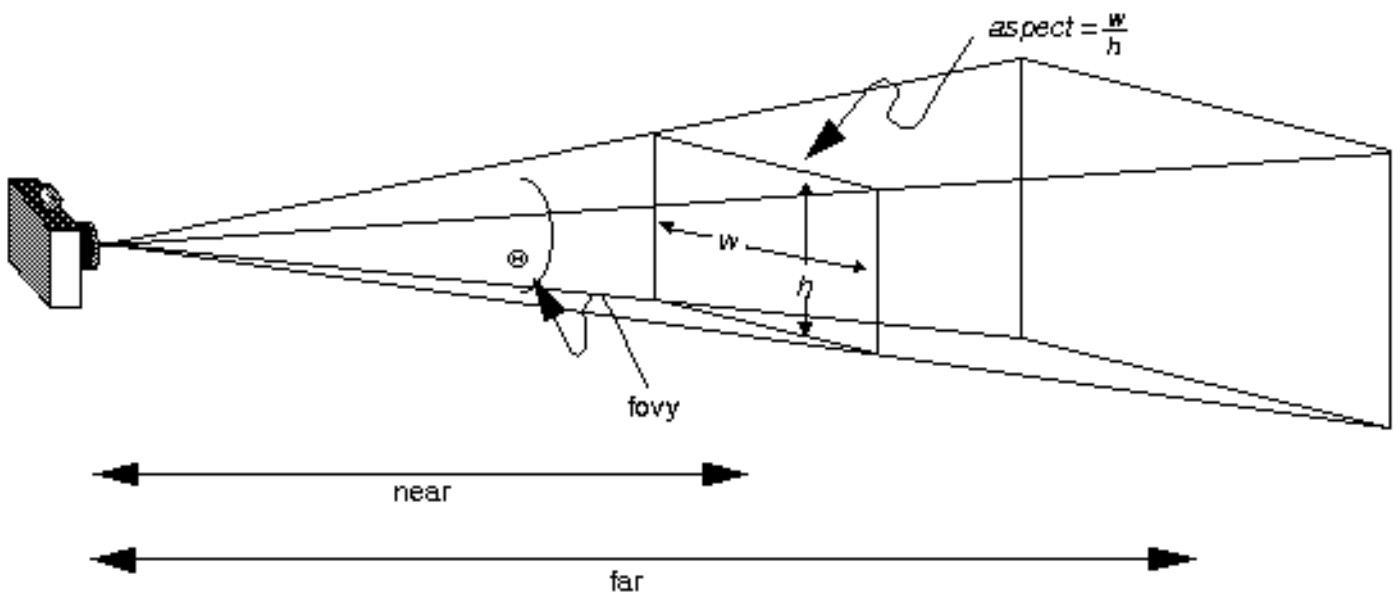


Figure 17: gluPerspective.

## Orthographic Transformation

$$\begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (34)$$

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

## Appendix E. Cubic Bézier curves.

Bernstein polynomials can be defined as follows:

$$B_i^n(u) = \binom{n}{i} u^i (1-u)^{n-i}, \quad (35)$$

where the value of the *binomial* coefficient  $\binom{n}{i}$  is given by

$$\binom{n}{i} = {}_n C_i = \frac{n!}{i!(n-i)!}. \quad (36)$$

Cubic Bézier curves are based on the four Bernstein polynomial blending functions,  $B_0^3$ ,  $B_1^3$ ,  $B_2^3$ , and  $B_3^3$  shown in Figure 18.

$$\begin{aligned} B_0^3(u) &= u^3, \\ B_1^3(u) &= 3u^2(1-u), \\ B_2^3(u) &= 3u(1-u)^2, \\ B_3^3(u) &= (1-u)^3. \end{aligned} \quad (37)$$

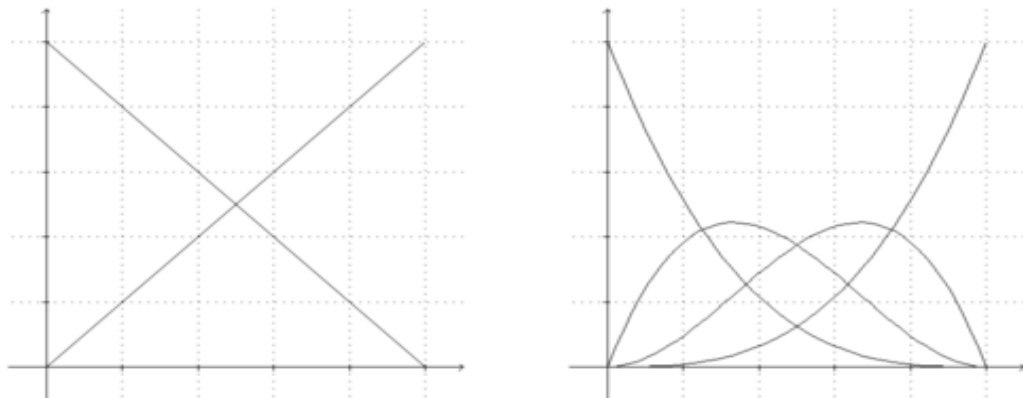


Figure 18: Blending functions. Linear (left) and cubic Bernstein (Bézier) (right).

## Appendix F. OpenGL Graphics Primitives.

Figure 19 shows the list of possible graphics primitives and possible arguments for `glBegin` and Figure 20 gives a diagrammatic explanation.

|                                |                                                                     |
|--------------------------------|---------------------------------------------------------------------|
| <code>GL_POINTS</code>         | individual points                                                   |
| <code>GL_LINES</code>          | pairs of vertices interpreted as individual line segments           |
| <code>GL_POLYGON</code>        | boundary of a simple, convex polygon                                |
| <code>GL_TRIANGLES</code>      | triples of vertices interpreted as triangles                        |
| <code>GL_QUADS</code>          | quadruples of vertices interpreted as four-sided polygons           |
| <code>GL_LINE_STRIP</code>     | series of connected line segments                                   |
| <code>GL_LINE_LOOP</code>      | same as above, with a segment added between last and first vertices |
| <code>GL_TRIANGLE_STRIP</code> | linked strip of triangles                                           |
| <code>GL_TRIANGLE_FAN</code>   | linked fan of triangles                                             |
| <code>GL_QUAD_STRIP</code>     | linked strip of quadrilaterals                                      |

Figure 19: Geometric Primitive Names and Meanings.

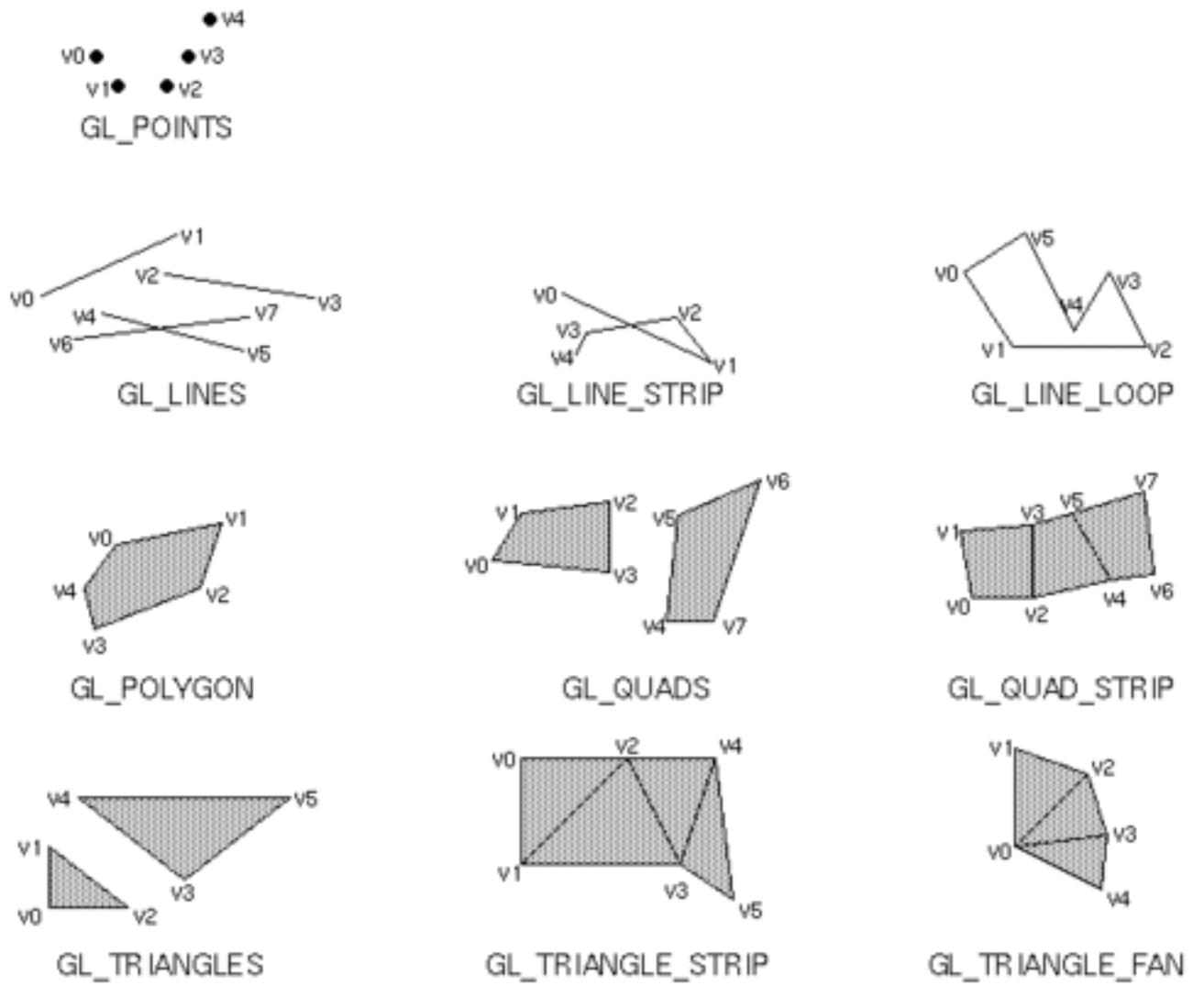


Figure 20: Geometric Primitive Types.