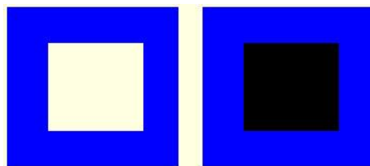
CS 535 Computer Graphics
Homework Assignment 3 Solution Set (40 points)

Due: 9/27/2024

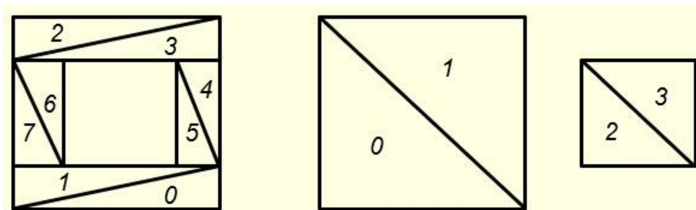
1. Based on example programs 1-6 in the notes "OpenGL and Shaders", write a simple C++/openGL program to animate 2D rotation of **one** of the following hollow blue squares of dimension 120x120 (pixels) at the center of your glfw window, the white (black) portion is of dimension 60x60 (pixels). Use "HW3 – Question 1" as the title of your glfw window and use white as the background color of your glfw window. Your vertex shader and fragment shader should be written in separate glsl files.



Turn in a screen shot of the rotating hollow square and your program with the submission of your HW3. . (10 points)

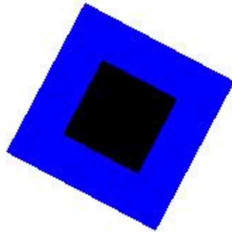
Sol:

If you choose to rotate the hollow blue square on the left (the inner, smaller square is white), one approach is to decompose the hollow blue square into 8 triangles such as the left case in the following figure, and then perform the rotation operation in the Vertex Shader. In this case your Fragment Shader needs to set one color (blue) only, for all the 8 triangles.

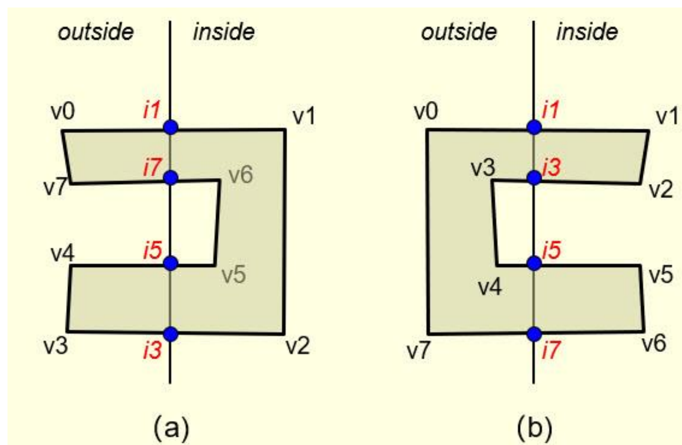


If you choose to rotate the hollow blue square on the right (the inner, smaller square is black), one approach is to decompose the hollow blue square into 4 triangles such as the middle and right cases in the above figure, and then perform the rotation operation in the Vertex Shader. In this case you need to call the function *glDrawArrays()* twice, first for triangles 0 and 1 with color blue, and then for triangles 2 and 3 with color black. This way, triangles 2 and 3 will

overwrite the central portion of the bigger blue square (union of triangles 0 and 1), so that the blue square and the black square will both be rotated, such as the case shown below.



2. If the two polygons shown in (a) and (b) are clipped against the left bounding edge of a 2D window using Sutherland-Hodgman and Weiler-Atherton algorithms, respectively, what would the outputs be in each case? In each case, the output by each algorithm is supposed to be a sequence of points (vertices of the original polygon or intersection points of the edges of the polygon with the left bounding edge of the window) or several sequences of points. For your convenience, intersection points of the polygon with the left bounding edge of the window are also shown below. (6 points)



Sol:

(For Sutherland-Hodgman and Weiler-Atherton algorithms, refer to 'Polygon clipping techniques' underneath the first programming assignment.)

For case (a), the putput of the SH algorithm is one polygon:

P: $i_1 v_1 v_2 i_3 i_5 v_5 v_6 i_7$

The output of the WA algorithm initially consists of two polygons:

P1: $i_1 v_1 v_2 i_3$ and P2: $i_5 v_5 v_6 i_7$

However, since the start vertex and end vertex of P2 are between the start vertex and end vertex of P1, P2 is then concatenated with P1 to form a single sequence. So the final output of WA algorithm is also one polygon:

P: $i_1 v_1 v_2 i_3 i_5 v_5 v_6 i_7$

For case (b), the output of the SH algorithm is one polygon:

P: $i_1 v_1 v_2 i_3 i_5 v_5 v_6 i_7$

The output of the WA algorithm consists of two polygons:

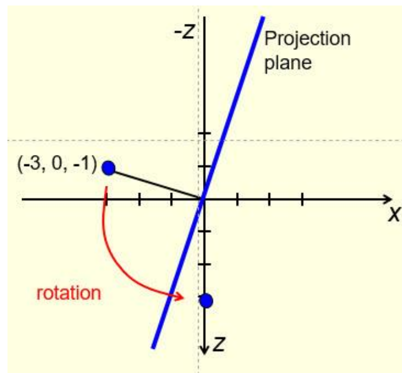
P1: $i_1 v_1 v_2 i_3$ and P2: $i_5 v_5 v_6 i_7$

P1 and P2 will not be merged into one sequence in this case (why?).

3. In perspective projection, if center of projection (COP) is at (-3, 0, -1) and the projection plane passes through the origin and is perpendicular to the vector (-3, 0, -1) then what (4x4) matrix should we use to compute the projection of a given point (x, y, z) on the projection plane in homogeneous coordinates? (5 points)

Sol:

First, as the figure shown below, we perform a rotation about the y-axis so that after the rotation, COP is on the z-axis and the projection plane coincides with the xy-plane.



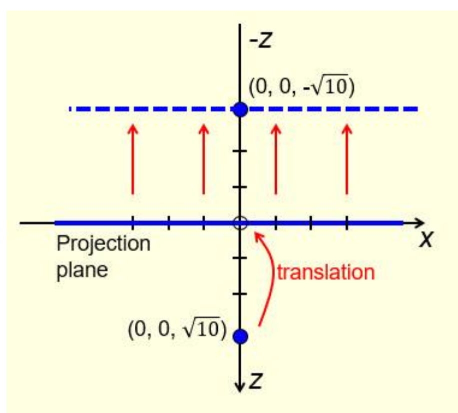
The rotation matrix for that purpose, called M_{r1} , is of the following form:

$$M_{r1} = \begin{bmatrix} -1/\sqrt{10} & 0 & 3/\sqrt{10} & 0 \\ 0 & 1 & 0 & 0 \\ -3/\sqrt{10} & 0 & -1/\sqrt{10} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

As shown below, after the rotation, the COP is at $(0\,,0,\ \sqrt{10})$.

$$M_{r1} \begin{bmatrix} -3 \\ 0 \\ -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \sqrt{10} \\ 1 \end{bmatrix}$$

The next step is to perform a translation to move the COP to the origin of the coordinate system, as shown in the figure below.



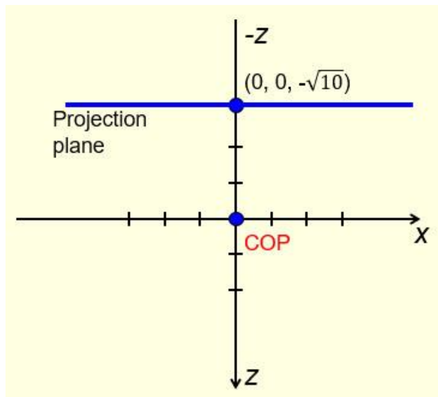
The translation matrix for that purpose, called M_{t1} , is of the following form:

$$M_{t1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -\sqrt{10} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

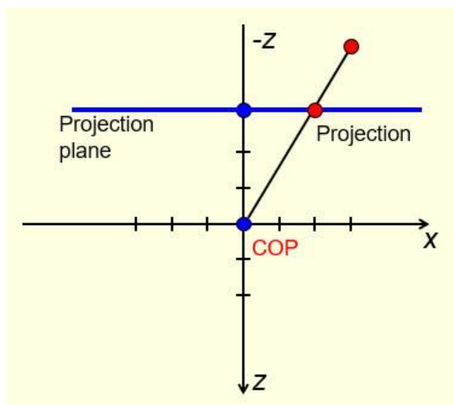
For instance, if we multiply COP, $(0, 0, \sqrt{10})$, by M_{t1} we get

$$M_{t1} \begin{bmatrix} 0 \\ 0 \\ \sqrt{10} \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

After the translation, the COP is at the origin of the coordinate system and the projection plane is at $z = -\sqrt{10}$ (see the following figure).



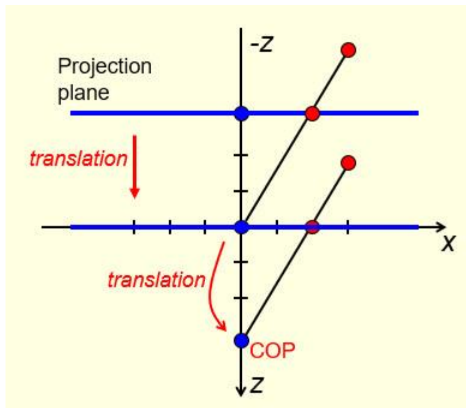
So we can use the standard perspective projection matrix M_{per} to perform perspective projection, as follows:



M_{per} is of the following form:

$$M_{per} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/\sqrt{10} & 0 \end{bmatrix}$$

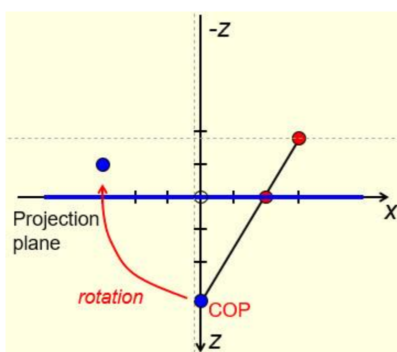
After the projection is performed, we translate the COP from the origin back to (0, 0,$\sqrt{10}$), as follows:



The translation matrix for this purpose, called M_{t2}, is of the following form:

$$M_{t2} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \sqrt{10} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Next, as shown in the following figure, we perform a rotation about the y-axis to move the COP from $(0, 0, \sqrt{10})$ back to its original location (-3, 0, -1), as follows:



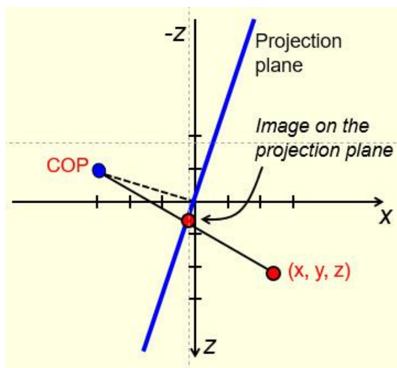
The rotation matrix for that purpose, called M_{r2} , is of the following form:

$$M_{r2} = \begin{bmatrix} -1/\sqrt{10} & 0 & -3/\sqrt{10} & 0 \\ 0 & 1 & 0 & 0 \\ 3/\sqrt{10} & 0 & -1/\sqrt{10} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

M_{r2} is the inverse of M_{r1}. Indeed, if we multiply $(0, 0, \sqrt{10})$ by M_{r2}, we get (-3, 0, -1), as shown below.

$$M_{r2} \begin{bmatrix} 0 \\ 0 \\ \sqrt{10} \\ 1 \end{bmatrix} = \begin{bmatrix} -3 \\ 0 \\ -1 \\ 1 \end{bmatrix}$$

So the projected image of (x, y, z), as shown in the following figure,



is M(x, y, z, 1)^t where

$$M = M_{r2} M_{t2} M_{per} M_{t1} M_{r1} = \begin{bmatrix} 1/10 & 0 & -3/10 & 0 \\ 0 & 1 & 0 & 0 \\ -3/10 & 0 & 9/10 & 0 \\ 3/10 & 0 & 1/10 & 1 \end{bmatrix}$$

You don't need to compute the entries of M. It is sufficient as long as all the component matrices of M are listed.

4. In perspective projection, if the projection plane is perpendicular to *z* axis at z = -4 and **vanishing point** of a line passing through A=(-5, 0, 0) and B=(0, 0, -5) is at (5, 0, -4), then where is the **center of projection**? (5 points)

Sol:

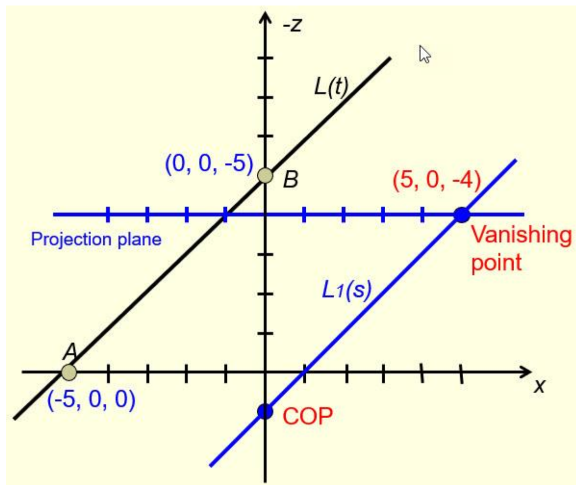
Let L(t) be a parametric representation of the line that passes through A = (-5, 0, 0) and B = (0,0, -5), as shown in the following figure. We have

$L(t) = A + t(B-A) = (-5, 0, 0) + t(5, 0, -5), \quad t \in R$

Then any point on the line

$L_1(s) = (5, 0, -4) + s(5, 0, -5) = (5+5s, 0, -4-5s), \quad s \in R$

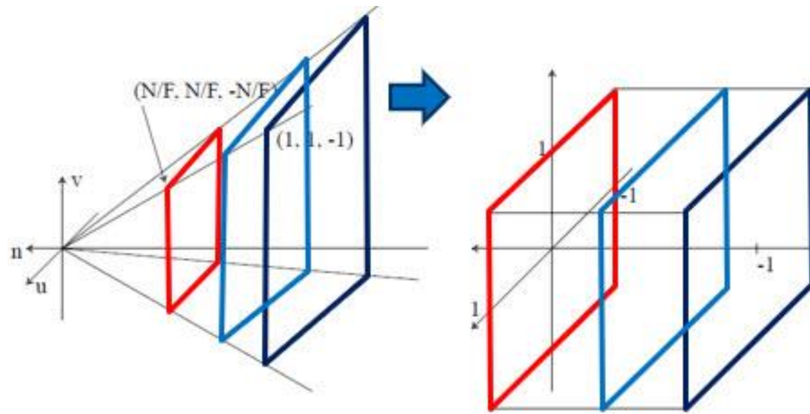
with a negative parameter ($s < 0$) can be a center of projection (why?). For instance, to choose a point of $L_1(s)$ that is also a point of the z-axis to be the center of projection (such a point's x-component and y-component are both zero), we need to set $s = -1$. In such case, the center of projection is (0, 0, 1).



5. In 3D graphics, for perspective projection, we do clipping in homogeneous coordinates, i.e, before the perspective division step. Why? (5 points)

Sol:

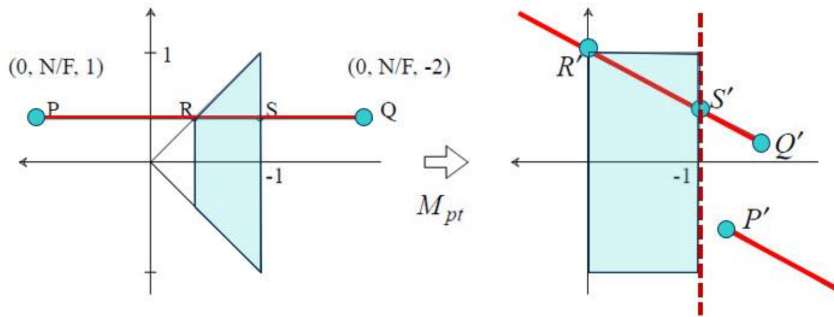
In OpenGl's implementation of 3D graphics, perspective projection is implemented as a perspective transformation followed by an orthographic (parallel) projection. Perspective transformation is to convert a canonical view volume for Perspective projection to a quasi-canonical view volume for parallel projection (see the figure below).



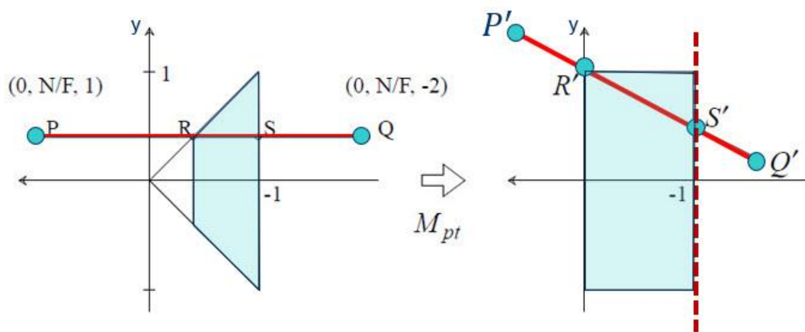
Perspective transformation is performed by multiplying each point (x, y, z, 1) by the so-called perspective transformation matrix M_{pt} as follows:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{F}{F-N} & \frac{N}{F-N} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ \frac{Fz+N}{F-N} \\ -z \end{bmatrix} = \begin{bmatrix} -x/z \\ -y/z \\ \frac{Fz+N}{-z(F-N)} \\ 1 \end{bmatrix}$$

After the multiplication, the x-, y-, z- and the w-components are all divided by the fourth component (the w-component) to make the fourth component equal to 1 again (see the second part and the third part of the above equation). This step is called the perspective division step. This step is not a continuous numerical operation. Points close to each other before this step might not be close to each other again after this step, and this would cause problem. Consider the line segment PQ on the left side of the following figure. The sub-segment RS of PQ is inside the canonical view volume for perspective projection and is mapped to the sub-segment $R'S'$ on the right side of the following figure. Since $R'S'$ is inside the quasi-canonical view volume for parallel projection, it should be kept and sent for rendering subsequently. However, since P and Q are mapped to P' and Q' and they are both to the right of the plane z=-1, so the clipping algorithm would think the entire line segment $P'Q'$ is to the right of the plane z=-1 and, consequently, outside the view volume for parallel projection, so the entire line segment $P'Q'$ would be discarded even though $R'S'$ is actually inside the canonical view volume for parallel projection.



The reason that P is mapped to a point to the right of the plane z=-1 is because the perspective division step changed the sign of the third component (z-component) in doing the perspective transformation process. If we don't do the perspective division immediately (i.e., keeping the fourth component as -z) after perspective transformation, then P' would remain as a point to the left the xy-plane as the case shown on the right side of the following figure. Consequently we wouldn't have the problem of discarding the entire line segment $P'Q'$ in the clipping process because now P' and Q' are on different sides of the canonical view volume. We would still have to do the perspective division step, but only after the clipping step.



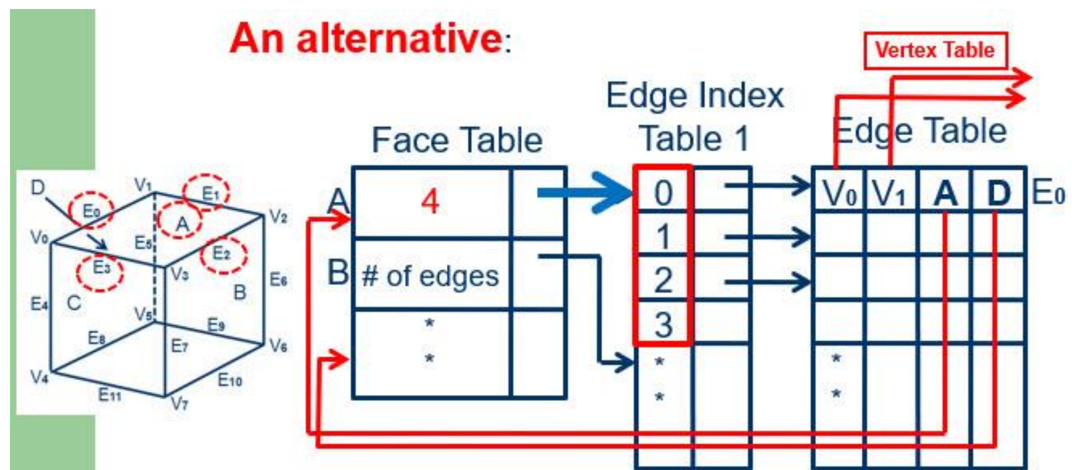
6. For the alternative data structure given in slides 3-6 of the notes "3D Data Structures" for 3D objects with bounding faces (polygons) represented as sequences of bounding edges, which of the following queries can be answered in constant time? (4 points)

   (a) Given a face, which faces are adjacent to it?
   (b) Given an edge, which faces are sharing it as a bounding edge?
   (c) Given a vertex, which faces are adjacent to it?
   (d) Given a vertex, which edges are adjacent to it?

Sol:

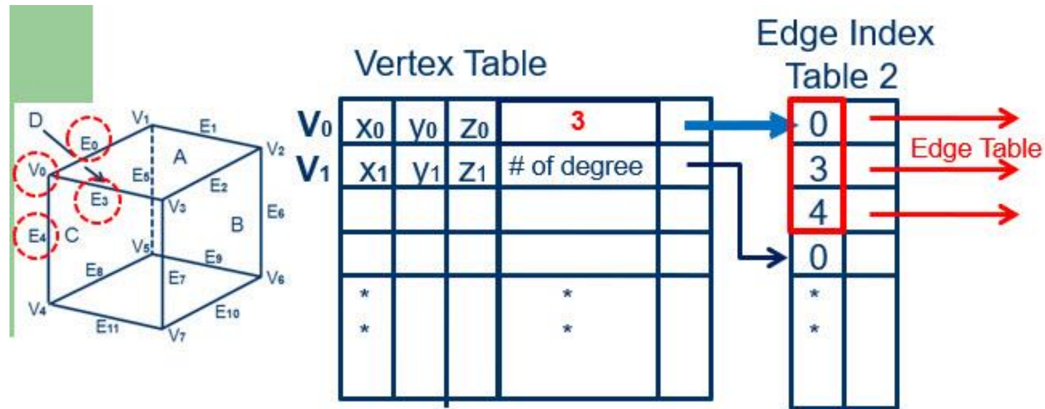
(a) NO. The answer is LINEAR TIME (w.r.t. the number of edges of the face). For instance, in the following figure, if face A is given, we can follow the pointers from Face Table to Edge Index Table 1 and then to Edge Table to find all the edges of A (i.e., E0, E1, E2 and E3) and then follow pointers from Edge Table to Face Table to find all the faces that share those edges with A (i.e., D, C, B, …).

(b) YES. If we know the index of that edge, we can go directly to that edge in the Edge Table and then follow the two pointers in that entry of the Edge Table to the Face Table to get the two faces that share this edge.

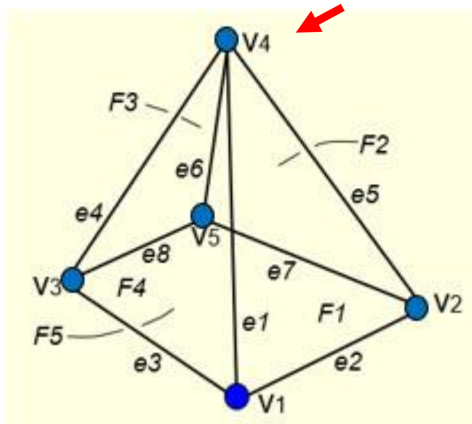


(d) YES. Given a vertex, if we know the index of that vertex, say V0, we can go directly to that vertex in the Vertex Table. From the fourth entry of that vertex, we would know how many edges share this vertex as an endpoint (in this case, 3; see the following figure) and then follow the pointer in the fifth entry of that vertex to Edge Index Table 2 and then to Edge Table, we would be able to find all the edges in the Edge Table that share V0 as an endpoint (i.e., E0, E3 and E4; see the following figure) immediately.

(c) NO. The answer is LINEAR TIME (w.r.t. the number of edges adjacent to the vertex). Given a vertex, we can first use the technique of (d) to find all edges adjacent to that vertex, and then use the technique of (b) to find all faces that share these edges. These faces are faces that are adjacent to the given vertex.



7. To use the winged-edge data structure to represent the following pyramid, we need an Edge Table, a Vertex-Edge Table and a Face-Edge Table. To find all the **adjacent edges** of vertex v4, how many times do we have to access the Edge Table, the Vertex-Edge Table and the Face-Edge Table, respectively? Justify your answer. (5 points)



Sol:

We start with the **Vertex-Edge Table**. If the edge selected for vertex V4 in the Vertex-Edge Table is, say e5, then we go to edge e5 in the **Edge Table**. If the direction of e5 is from V2 to V4, then in the entry for e5 in the Edge Table, we find the successor edge of e5 in counter-clockwise direction. Otherwise we find the successor edge in clockwise direction. In either case we will get e6. Then we go to the entry for e6 in the Edge table. There again, we find the successor edge of e6 in counter-clockwise direction if the direction of e6 is from V5 to V4. Otherwise, we find the successor edge of e6 in clockwise direction. In either case, we will get e4. When we go to the entry in Edge Table for e4 to repeat the same process, we would get e1 as

the successor edge of e4 either in counter-clockwise direction or clockwise direction. When we go to the entry in Edge Table to repeat the same process, we would get e5 as the successor edge of e1 either in counter-clcokwise direction of clockwise direction. So we know the edge we started with has been reached and we stop, and e5, e6, e4, and e1 are the adjacent edges of V4.

Totally, we have accessed the Edge Table 4 times, the Vertex-Edge Table once and the Face-Edge Table 0 times.