INDIAN INSTITUTE OF TECHNOLOGY, DELHI

MATHEMATICAL MODEL

Design and Implementation of a software package for Engineering drawing

Authors:
Shreshth TULI
Sankalan Pal CHOWDHURY

Supervisor:
Dr. Subashish BANERJEE

Course: Design Practices - COP290
Department of Computer Science and Engineering

Developed on LaTeX

January 18, 2018

Chapter 1

3D Transformations

Once a 3D object has been built, we need to be able to transform it in various ways. Below, we describe three families of such transforms

1.1 Translation of 3D Object

Translation is perhaps one of the most basic operations that can be performed on an object. It can be defined simply as the elementary transform which can preserve relative positions of any two points in space. Mathematically, it is defined as:

$$T_{x_0 y_0 z_0}((x, y, z)) = ((x + x_0, y + y_0, z + z_0))$$

While this is almost trivial to implement, we would still resort to libraries for the sake of continuity.

1.2 Scaling of 3D Object

Scaling is a linear transformation that enlarges or shrinks an object by scale factors that may or may not be equal in all directions. The scaling matrix is any diagonal matrix with positive reals as elements. Should the scaling be uniform, the scaling matrix takes the special form kI where $k \epsilon \mathbb{R}^+$ is the scaling factor and I is the 3^{rd} order Identity matrix.
While scaling is definitely an important type of transform, It is not generally used in Engineering Drawing.

1.3 Rotation of a 3D Object

Rotation in 3D may be imagined in multiple intuitive ways. However, for mathematical treatment, it is necessary to formulate rotation formally. Hence, we define rotation as any linear function from $\mathbb{R}^3$ to $\mathbb{R}^3$ that has the following properties:

- There exists a line in the space that is same as its image. This line is called the Axis of Rotation(AoS). For simplicity, we assume that $(0, 0, 0)^T$ lies on this axis.
- The distance between any two points is same as the distance between their images.

1.4 Matrix Representation

Since our function is a linear map from $\mathbb{R}^3$ to $\mathbb{R}^3$, it can be represented in the form of a 3x3 Matrix. We begin with the most general form of this matrix:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

This has a total of 9 degrees of freedom. Hereon, we try to discover constraints that can reduce this.

1.5 Preservation of Distances

Since rotations preserve distances, between any two arbitrarily chosen points it must also preserve distance between $(0,0,0)^T$ and $(1,0,0)^T$. Now the image of $(0,0,0)^T$ is $(0,0,0)^T$ whereas the image of $(1,0,0)^T$ is $(a_{11}, a_{21}, a_{31})^T$, so we must have

$$a_{11}^2 + a_{21}^2 + a_{31}^2 = 1$$

This same technique can be applied to $(0,1,0)^T$ and $(0,0,1)^T$ to obtain the relations:

$$a_{12}^2 + a_{22}^2 + a_{32}^2 = 1$$
$$a_{13}^2 + a_{23}^2 + a_{33}^2 = 1$$

These equations show that in each column, only two parameters are independent. Now, consider the three points $O(0,0,0)^T$, $B(1,0,0)^T$ and $C(0,1,0)^T$. Since rotation would preserve pairwise distances, the triangle formed by their images $(O'B'C')$ would be congruent to the triangle OBC. Since we know that $O' = O$,

$$\angle B'OC' = \angle BOC = 90^\circ$$

Which implies that the dot product between B′ and C′ must be zero. This implies that

$$a_{11}a_{21} + a_{12}a_{22} + a_{13}a_23 = 0$$

Similar manipulations with other points can show that any pair of columns are orthogonal. These, and the previous relations can all be captured in the following constraint:

$$\sum_{j=1}^{3} a_{ij}a_{kj} = \delta_{ik}$$

Where δ_{ik} is the Kronecker delta. Two implications follow:

- The matrix under question is Orthogonal, i.e. its transpose is same as its inverse
- Since there are 6 constraints on 9 variables, only three of these variables can be independent

Till now we have only used the fact that the Origin is invariant under rotation. The fact that the entire axis remains invariant actually adds the further constraint that

the determinant of the Matrix is +1 (and not -1).
Specifying an arbitrary axis passing through the origin takes two parameters. To define a rotation completely, we must have one more parameter. This is taken to be the Angle of rotation.

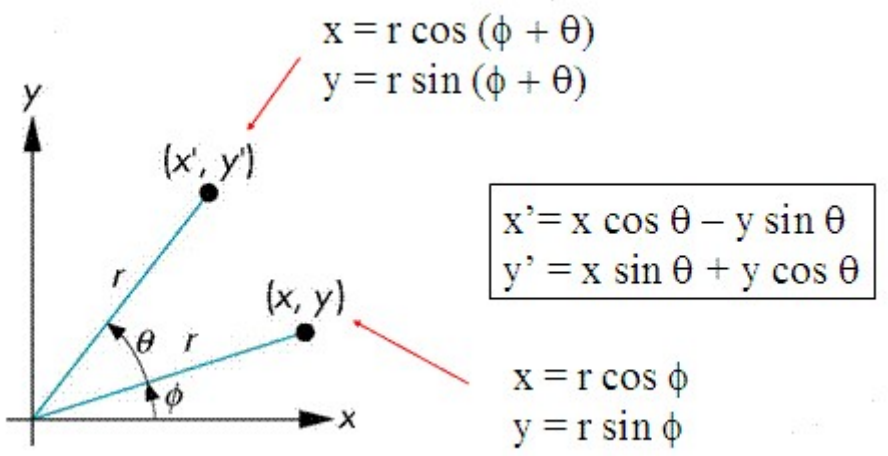
Definition:
Let P be a point not lying on the Axis of rotation. Let P' be its image and P'' be its projection onto the AoS. $\angle PP''P'$ can be proven to be independent of the choice of P and is known as the axis of rotation.

1.6 Simpler Cases

The 2D version of the rotation problem is rather easy to calculate. A 2D rotation by an angle θ about the origin is characterized by the matrix:

$$\begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix}$$

The derivation is rather easy and can be inferred from the following figure:



Now, rotation in 3D about Z axis by angle θ can be seen as simply transforming the X and Y coordinates using the above matrix and keeping the Z coordinate unchanged. This gives us the matrix:

$$M_Z(\theta) = \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

But there is nothing special about the the Z axis, and we can similarly obtain rotation matrix for rotation about X and Y Axes in the same way.

$$M_X(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\theta & -sin\theta \\ 0 & sin\theta & cos\theta \end{bmatrix}$$

$$M_Y(\theta) = \begin{bmatrix} cos\theta & 0 & sin\theta \\ 0 & 1 & 0 \\ -sin\theta & 0 & cos\theta \end{bmatrix}$$

1.7 Putting it together

We now construct the general 3D rotation matrix.
Let the axis of rotation be define parametrically as $(r\,sin\alpha\,cos\beta, r\,sin\alpha\,sin\beta, r\,cos\alpha)$ with r as a parameter. Also, let θ be the Angle of rotation. We seek to find out $M(\alpha, \beta, \theta)$.

Had our axis coincided with the Z axis, we would have $M(0, 0, \theta) = M_Z(\theta)$. However, when that is not the case, we can still use this if we can perform suitable rotations to bring the Axis of Rotation to Z axis.

Once the rotation has been performed, our rotations can be reversed, and the Axis of rotation brought to its original rotation. This involves 5 steps:

1. Rotate around Z axis by angle β to bring AoS to the XZ-Plane.
2. Rotate around Y by angle α to make AoS coincide with Z axis.
3. Rotate around Z axis by θ.
4. Invert step 2.
5. Invert step 1.

All in all, we get:

$$M(\alpha, \beta, \theta) = M_Z^T(\beta)\, M_Y^T(\alpha)\, M_Z(\theta)\, M_Y(\alpha)\, M_Z(\beta)$$

At this point, Simplification becomes a mere computational task. For our purpose, we would leave this up to standard Libraries.

Chapter 2

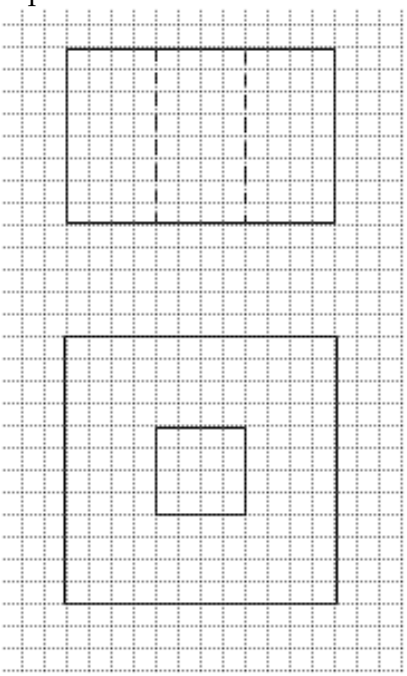
3D - 2D Conversion

A basic purpose of the tool we are working on would be to generate a 3D model from a given set of orthographic projections. Further, we should also be able to generate all possible projections given a 3D model. In this aspect, being able to Convert between 2D and 3D becomes essential.

Note: In the rest of the document it is assumed that objects do not consist of curved edges or surfaces

2.1 Representation

2D projections are normally presented in Engineering drawing as a labelled wireframe with separate patterns for visible(solid) and hidden(dashed) lines. To represent this in our program, we shall use a Eulerian graph with labeled vertices and separate sets of visible and hidden edges.



```
<top>
    <points>
        <pt nm=a x=3 y=2/>
        <pt nm=b x=15 y=2/>
        <pt nm=c x=7 y=2/>
        <pt nm=d x=11 y=2/>
        <pt nm=e x=3 y=10/>
        <pt nm=f x=15 y=10/>
        <pt nm=g x=7 y=10/>
        <pt nm=h x=11 y=10/>
    </points>
    <solid>
        <eg sr=a en=c/>
        <eg sr=c en=d/>
        <eg sr=d en=b/>
        <eg sr=e en=g/>
        <eg sr=g en=h/>
        <eg sr=h en=f/>
        <eg sr=a en=e/>
        <eg sr=b en=f/>
    </solid>
    <dashed>
        <ed sr=c en=g/>
        <ed sr=d ed=h/>
    </dashed>
</top>
<front>
    ...
</front>
```

(description shown here is not final, and may change subject to programming requirements)

The 3D representation is much more internal, and would be visible to the user only through projections. We propose to use a similar description, except for having faces enumerated as well. This would become important for separating solid edges from dashed ones.

2.2 Projection: From 3D to 2D

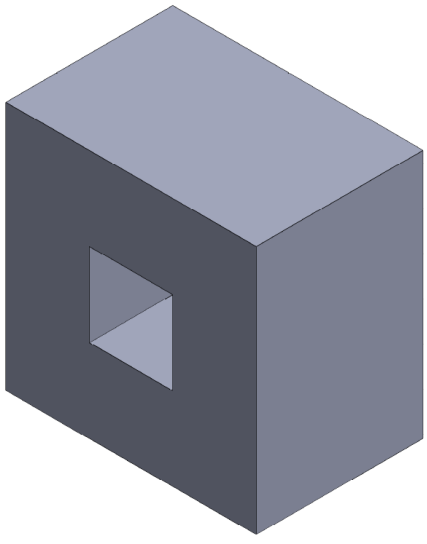
We have already discussed at length how points can be rotated and transformed in 3D space. Since edges and planes are attached to points that surround them, they also get rotated along with points. To project a certain orthographic view, we shall use the following technique:

1. Rotate the object so that the plane of interest coincides with the XY plane.
2. Start off from the point having Z coordinate farthest from the viewer.
3. Drop the Z coordinate to project points onto the drawing plane. Do this in the order that points farther from the viewer get projected first.
4. Whenever both end points of an edge have been projected, create the edge as solid.
5. Whenever all bounding points of a plane have been projected, calculate the region covered by the plane. Change any solid lines in this region to dashed lined.
6. Finish when all points have been projected.

A good implementation of this routine should be able to run in $O(F + E + V)$. Using Euler's Characteristic formula, this is same as $O(V)$ where V is the number of vertices. Note that at this point, we have not lost invertibility. We can show this using the example considered above.

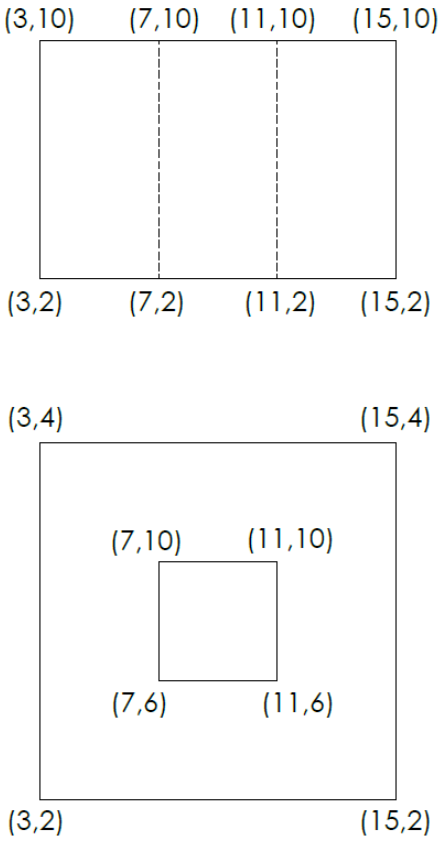
2.3 Reconstructing: From 2D to 3D

Here, we consider the problem of generating a 3D wireframe given an orthographic projection (front and top) that was generated from a wireframe using the above algorithm at the first place.



Exemplar 3D Solid object

2.3.1 Modeling though example



Otho graphic view

The input here consists of a pair of vectors of tuples giving the projected points in top and front views. The edge sets would be identical, except some solid edges becoming hidden and vice versa. The most important fact here is that there must be a one-to one correspondence between vertices in both (or all three) of the views.

```
Top:    Vertices:
[(3,2),(3,2),(15,2),(15,2),(7,2),(7,2),(11,2),(11,2),(3,10),(3,10),(15,10),(15,10),(7,10),(7,10),(11,10),(11,10)]
        S_Edges :   [(0,1),(1,2),(2,3),(3,0),(4,5),(5,6),(6,7),(7,4),(1,8),(2,9),(3,10),(4,11),(8,9),(9,10),(10,1
        1),(11,8),(12,13),(13,14),(14,15),(15,12)]
        D_Edges :   [(4,12),(5,13),(6,14),(7,15)]

Front:  Vertices:
[(3,2),(3,14),(15,14),(15,2),(7,6),(7,10),(11,10),(11,6),(3,2),(3,14),(15,14),(15,2),(7,6),(7,10),(11,10),(11,6)]
        S_Edges :   [(0,1),(1,2),(2,3),(3,0),(4,5),(5,6),(6,7),(7,4),(1,8),(2,9),(3,10),(4,11),(8,9),(9,10),(10,1
        1),(11,8),(12,13),(13,14),(14,15),(15,12),(4,12),(5,13),(6,14),(7,15)]
        D_Edges :   []
```

Note that the order of vertices is actually defining a labeling for them, and is therefore fixed. The order of edges is, however, variable.

Should we have this kind of an input, we can easily create a deterministic algorithm.

Let (X_{ti}, Y_{ti}) be the i^{th} vertex in the top view, (X_{fi}, Y_{fi}) be the i^{th} vertex in the front view, and N_t, N_V be the cardinality of these sets respectively.

Further, let $S_t\, H_t \subset \{(a,b) \,|\, a,b \subset (0,1,2,...N_t), a \neq b\}$ be the set of Solid and Hidden edges for the top view and, $S_f\, H_f \subset \{(a,b) \,|\, a,b \subset (0,1,2,...N_f), a \neq b\}$ be that for the front view.

The given set of inputs is a valid orthographic projection if:

1. $N_t = N_V$
2. $x_{ti} = x_{fi}$ for all valid i
3. $S_t \cup H_t = S_f \cup H_f$

The original members vertex set $V(x_i, y_i, z_i)$ can then be represented as: $x_i = y_{ti}$, $y_i = -y_{fi}$, $z_i = -x_{ti}$. This would be correct except the possibility of a translation to the entire object, and fully correct when seen in relative terms.

The edge set is even easier to recover. We simply have $E = S_t \cup H_t$.

Recovering the set of faces is more complex though. In principle, every cycle in the graph may represent a face. This is in fact a good place to begin. Thereafter, we need to apply the following two filters: The given set of inputs is a valid orthographic projection if:

1. If one face completely contains another face, then the second face must actually be a hole in the first.
2. No three faces can share the same edge as boundary. If they do, nne of the faces must be falsely identified.

This kind of an input, though easy to invert, is hardly ever available from a user. The closest one can get then, is when all vertices of the drawing are labeled. Even then, more often than not, there can be a discrepancy w.r.t. which edges exist and which do not.

2.3.2 Implementation of example

We try to apply this procedure to the example considered above. First comes the part of Recovering vertices. In O(V) time, a rather naive algorithm can return:

```
Vertices:  [(0,0,0),(-12,0,0),(-12,0,-12),(0,0,-12),(-4,0,-4),(-8,0,-4),(-8,0,-8),(-4,0,-8),
            (0,-8,0),(-12,-8,0),(-12,-8,-12),(0,-8,-12),(-4,-8,-4),(-8,-8,-4),(-8,-8,-8),(-4,-8,-8)]
```

Note that apart from matching, the origin has also been shifted to the first point. This choice is arbitrary, but, nevertheless, logical. The edges are essentially just copied.

```
Edges   :   [(0,1),(1,2),(2,3),(3,0),(4,5),(5,6),(6,7),(7,4),(1,8),(2,9),(3,10),(4,11),(8,9),(9,10),(10,1
             1),(11,8),(12,13),(13,14),(14,15),(15,12),(4,12),(5,13),(6,14),(7,15)]
```

Next comes the faces. An algorithm for discovering all cycles in a graph is not too hard, and there exists efficient algorithms like Johnson's Algorithm to do this in less than quadratic time. We, however, need not use such an implementation, as not all cycles are of interest to us. In fact, we are only looking at cycles which have all points lying on the same plane. But first, we state an obvious theorem.

From every vertex defined in a polyhedron, there must exist three or more edges originating from it of which no three can be coplanar, unless two are collinear

This fact can be exploited by our algorithm, which would continue down an edge only if that edge is in the same edge of the previous plane. The steps then are:

- Select an edge which has not been covered by at least two cycles.
- Select any one of its end-vertices.
- If this edge has already been covered in a cycle, choose one of the edges from this vertex that was not a part of that cycle. Otherwise, choose any other edge from this vertex and proceed. Note that these two edges should not be collinear.
- Now a plane is defined. At each successive vertex, keep choosing the edge that lies on this plane (but is mot collinear to it's predecessor) and continue till the cycle is completed.
- Repeat till every edge has been covered by at least two cycles.

Running this on our example would give the following output:

```
Cycles: [(0,1,2,3),(4,5,6,7),(0,1,8,9),(1,2,9,10),(2,3,10,11),(3,0,11,8),(4,5,12,13),
        (5,6,13,14),(6,7,14,15),(7,4,15,12),(8,9,10,11),(12,13,14,15)]
```

Here the cycles have been reported using vertices, but there is nothing to stop us from keeping connecting pointers between edges and cycles as well.

Now we run the two cleaning step (the order is important).

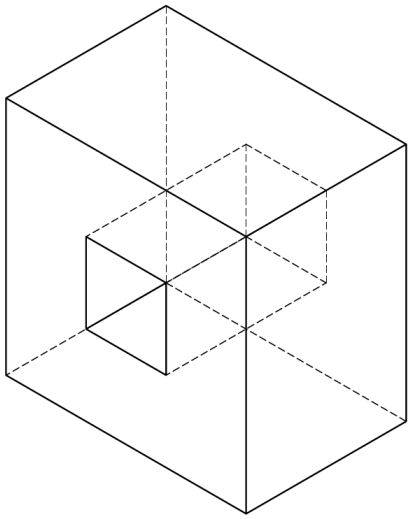
In our example, it is easy to see that (4,5,6,7) is completely covered by (0,1,2,3), while (12,13,14,15) is completely covered by (8,9,10,11). These two pairs are merged, giving:

```
Planes: [(0,1,2,3||4,5,6,7),(0,1,8,9),(1,2,9,10),(2,3,10,11),(3,0,11,8),(4,5,12,13),
        (5,6,13,14),(6,7,14,15),(7,4,15,12),(8,9,10,11||12,13,14,15)]
```

This step can easily be performed in time quadratic on the number of cycles, but better algorithms may exist.

The second cleansing step does not affect our example, but would basically involve removing of any planes which are bounded by edges all of which have "degree" three or more. This can be performed in time linear in the number of edges.

This kind of an input, though easy to invert, is hardly ever available from a user. The closest one can get then, is when all vertices of the drawing are labeled. Even then, more often than not, there can be a discrepancy w.r.t. which edges exist and which do not.



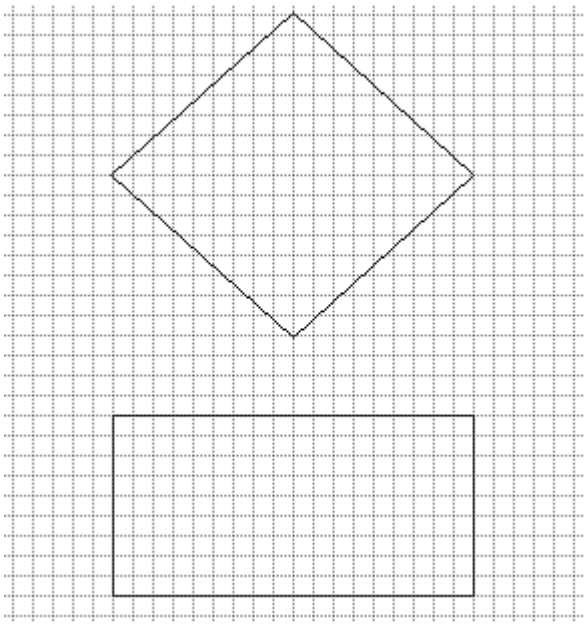
Isometric view of the exemplar object

Hereafter, we try to build a more generic approach which would work for unlabeled drawings.

2.3.3 Generalized algorithm

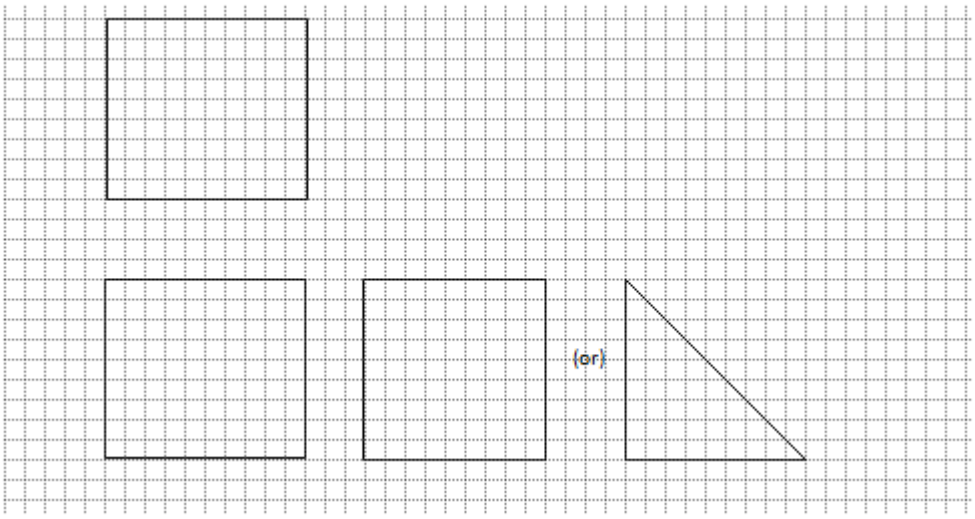
Since Projection drops a dimension, reverting from 2D to 3D is always a complex process. Almost any orthographic projection provides us two views, but this is clearly neither necessary nor sufficient.

It is not necessary because it is possible to have pairs of a top view and a front view which can never represent an object. One of the simplest examples is a pair of lines, both parallel to the reference line but having different lines. In this view, we may say that is the projections of both the views onto the reference plane is not identical, then there exists no object satisfying the drawing. This is still not sufficient however Ex: the figure shown probably is not realizable.



In general, any edge vertex must have a degree of at least three. If any vertex has an out degree of only two in any view, it must be also present in the other view. In the adjacent drawing, this is clearly not the case.

With all these limitations, two views are still not sufficient. Shown here is a set of top and frond views with two possible side views.



Given three views, the 3D structure is completely defined, but it also creates the possibility of several types of discrepancies including, but not limited to, the ones discussed for the 2D case.

Then what is to be done? Well, we would just rely on what is provided to us and hope for the best. The bare algorithm we shall start with is roughly described as the following:

- Check that all discovered constraints are satisfied. If not, throw Exception. Note that discovering of constraints is an ongoing process.
- Project any points present is Top view and not in front view to all lines on the same common coordinate, if doing so does not violate constraints.
- Repeat above with role of views exchanged.
- If side view is available apply above procedure to all other pairs of views.
- Create the points space as a cross product of coordinate sets. Create all edges that are present on atleast one view.
- Creation of faces is more difficult, but in general, Solid edges need to be covered before dashed ones.
- Apply any discovered sanity checks.
- Show the drawing to the user to suggest any changes.

Our primary aim in the rest of the design would be to eliminate the importance of the last step as much as possible.

As a final note, the overall process requires us to match sets of points in three drawings, so it definitely reminds one of 3D matching. While 3D matching is surely a much more involved process, it is also proven NP Hard. All we can hope then is, that we never meet it again.

2.4 Minimum number of views for reconstruction

Note: This section is developed from the theory of Reference number: 1 Now, we will discuss the theoretical minimum number of views required for the reconstruction of 3D conics. This number is important as it limits the possibility of exactly reconstructing an object with quadratic surfaces in theory.

We first define non-degenerate parallel projection which is useful for subsequent discussions.

Definition1:
Under a parallel projection, if the plane containing the space conic is not perpendicular to the projection plane, then the parallel projection is non-degenerate.

Under the non-degenerate parallel projections, all conics are equivalent, i.e. conics are mapped to conics. It follows, that ellipses, parabolas, and hyperbolas in the drawings are projections of ellipses, parabolas, and hyperbolas respectively. Therefore, if one of the plane curve is conic then the curve in 3D is also conic.

Suppose a space conic lies on a plane P and that an object coordinate system C_P is defined such that its X_P and Y_P lie on the plane P. Let C be a global coordinate system in space. Then the global representation $\mathbf{x} = [x \;\; y \;\; z \;\; 1]^T$ in C of a global point $x_P = [x_p \; y_p \; z_p \; 1]^T$ in C_P can be derived by applying transformation of the form:

$$x \;=\; Rx_p + t \tag{2.5.1}$$

where **R** and **t** are rotation and translation matrices respectively as discussed before.

Thus,

$$\mathbf{X} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} r_{00} & r_{01} & t_0 & \\ r_{10} & r_{11} & t_1 & \\ r_{20} & r_{21} & t_2 & \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_P \\ y_P \\ 1 \end{bmatrix} = P\, u_P \tag{2.5.2}$$

We now consider the relationship between space conic and its orthographic projections onto some projection planes. Let c_i (i = 1,2,3, ...q) denote the 2D local coordinate system associated with the i^{th} projection plane, and $u_i = [x_i, y_i, 1]^T$ denote the homogeneous coordinates of an arbitrary point in its local coordinate system. If C_i is a 3X4 matrix whose three columns form an orthogonal basis for this projection subspace, then the transformation from a point **x** in 3D space to point u_i in the projection plane is given by the relation:

$$u_i \;=\; C_i\, x_i \tag{2.5.3}$$

Now we will discuss some theorems that would be useful for our further discussions.

Theorem: If a vertex is projected onto three views, its image should be intersection point of two or more non-collinear line segments in at least on view.

Proof: The proof is derived by contradiction and the fact that all 2D vertices are intersection points of two or more non-collinear line segments. Suppose that all images of a vertex $v \in V$ in the three views are intersection points (denoted by P_1, P_2, P_3) of only two collinear line segments. A vertex can be projected as an intersection point of two collinear line segments only if a face f containing that vertex is perpendicular to the projection plane. To satisfy the assumption, all adjacent faces which share that vertex must be perpendicular to the three face planes simultaneously. However a face of an object cannot be perpendicular to all three projection planes simultaneously. Thus contradiction. Hence proved. ■

Theorem: A tangency edge is projected as a tangency vertex in only one view.

Proof: Let f_1 and f_2 be two curved faces tangent to each other. These faces are projected as two arcs s_1 and s_2 and they are also tangent to each other since both faces are parallel to the same principal axis. A 2D vertex which is an intersection point of two arcs tangent to each other has tangency type f_1 and f_2 are projected as two polygons in the other two views. Since two curved faces have C^1 continuity, no edges nor vertices would be present in their intersection. Hence proved. ■

Theorem: Three distinct orthographic projections are sufficient to uniquely recover a space conic.

Proof: First of all, suppose that none of the three projections is degenerate. The degenerate case will be considered later.
Let **A** be a space conic that lies on a plane P.

$$i_P^T \, A \, u_P = 0 \tag{2.5.4}$$

and its projection curves A_i are represented by,

$$u_i \, A_i \, u_i = 0, \quad i = 1, 2, ...q \tag{2.5.5}$$

Substituting the linear transformation of the form $u_i = G_i \, u_P$ into equation 2.5.5, we obtain

$$u_P^T \, G_i^T \, A_i \, G_i \, u_P = 0 \tag{2.5.6}$$

From eqns. 2.5.4 and 2.5.6 we get,

$$G_i^T \, A_i \, G_i \; = \; P^T \, C_i^t \, A_i \, C_i \, P \; = \; A, \quad i = 1, 2, ...q \tag{2.5.7}$$

where **A** and **P** are known matrices. By Bernstein special theorem [2], we can derive that the system of polynomial equations in eqn. 2.5.7 is solvable. Hence sufficient.

Further, consider the special case where at least one of the orthographic projections is degenerate. If so, then the projection of conic onto this plane is a straight line. By definition of orthographic projections, we can determine the plane on which the conic lies, which is obtained by extending the straight line along the degenerate projection direction. Since at least one of the projections of the conic is also a conic, we locate the center point of the space conic by finding its corresponding points in the other two views. Accordingly the matrix **P** is obtained. To reconstruct the space conic we solve $P^T C_f^T A_f C_f P = A$ for A, where subscript f indicates front view. In this case also sufficient. Hence proved. ■

Other such theorems can be seen in reference 8.

Chapter 3

Mathematical Model : OpenGL

3.1 Introduction

We implement perspective projection using a perspective transformation. Here we describe the perspective transformation used by OpenGL; DirectX uses a slightly different version, we mention that.

In its raw form, perspective transformation involves division; it is decidedly neither linear, see Chapter 1, and looks rather ugly, and so might not be expected to be implementable by our beloved matrices. However, owing again to the marvel of homogeneous coordinates, we are able to make it implementable by a matrix.

The essence of perspective projection is shown in Figure 3.1; a line PQ in the 3D world is projected onto P'Q' on the projection plane. C is the centre of projection. In orthographic (parallel) projection, Figure 8.4(b) the rays PP' and QQ' are parallel to one another and to the line of sight; being parallel they do not meet at a point.

Note: in cameras and in eyes, the projection plane is behind the centre of projection. In our virtual camera in OpenGL or DirectX we can place the camera in front of the centre of projection; there is no difference in the effect — except that now image objects do not appear upside down.

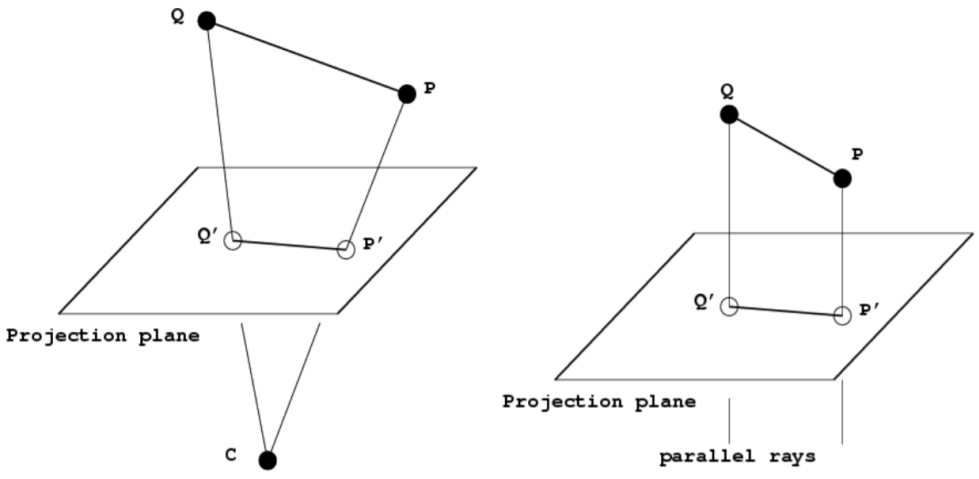


Figure 3.1: Projections: a

3.2 Geometry of Perspective

Figure 3.2 shows a horizontal cross-section of geometry of perspective projection; the vertical cross-section would look similar and would show the x-axis in place of the y-axis. The z-axis is pointing in the negative direction — in OpenGL the centre of projection is always at the origin and pointing along the negative z-axis.

The projective plane is at a distance e in front of the centre of projection. A point $P = (p_x, p_y, p_z)$ is projected onto $(x, y, -e)$.

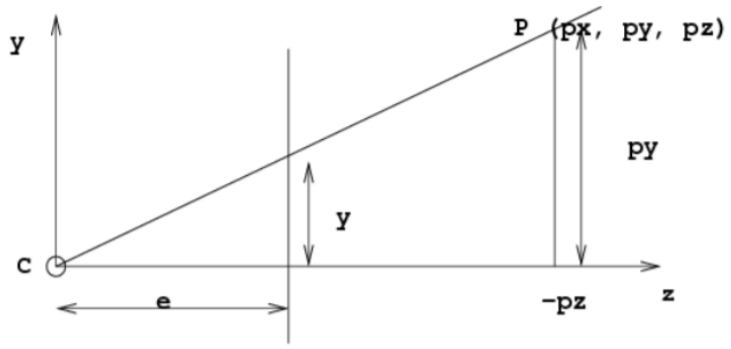


Figure 3.2: Perspective projection, cross-section

By similar triangles, we have:

$$y = -ep_y/p_z \tag{3.1}$$

If we imagine a similar diagram for x, we have:

$$x = -ep_x/p_z \tag{3.2}$$

Also,

$$z = -e \tag{3.3}$$

that is all z values get reduced to -e, rather like the simple orthographic projections in chapter 2 where projection onto the x-axis simply reduces the y value to 0 and projection onto the y-axis reduces the x value to 0. Later, we'll see that OpenGL does not squash z coordinates to one value, and that it transforms them into a set of values that are useful for determining hidden surfaces during rendering.

3.3 OpenGL glFrustum

The chief method of defining a perspective projection in OpenGL is to define a frustum of a pyramid using glFrustum, see Figure 3.3.

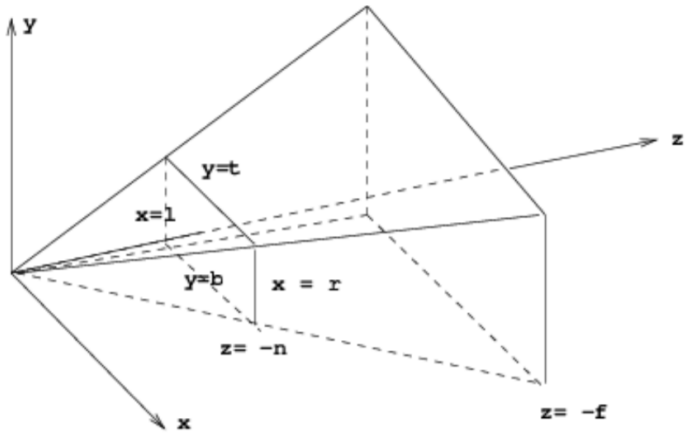


Figure 3.3: Perspective view frustum volume defined by glFrustum

glFrustum takes six arguments:

$void\, dlFrustum(GLdoubleleft, GLdoubleright, GLdoublebottom, GLdoubletop, GLdoublenear, GLdoublefar)$.

Respectively, these the left and right (x-axis), and the bottom and top (y-axis) extents of the field of view; -near is the z position of the projection plane. In addition, rendering is limited to the frustum defined by left, right, bottom, top, near, far; anything outside is not rendered and these form a 3D clipping region. In Figure 3.3 these are abbreviated l, r, b, t, n, f .

OpenGL maps the frustum given by l, r, b, t, n, f to the homogeneous clip space cube shown in Figure 3.4. It maps: $l \mapsto -1, r \mapsto +1, b \mapsto -1, t \mapsto +1$.

Also, to use in hidden surface removal, it retains mapped z coordinates; it maps z coordinates thus: $n7 \mapsto -1, f \mapsto +1$; notice that this reverses the direction of the z axis.

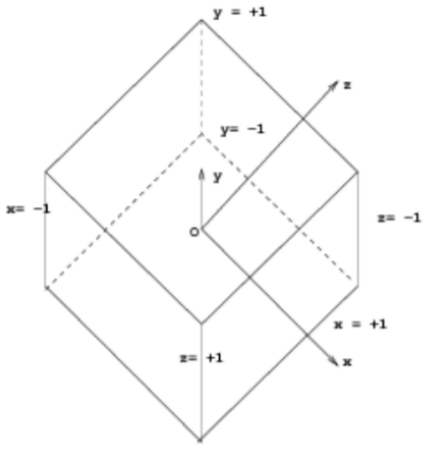


Figure 3.4 : Homogeneous clip space cube

3.4 Derivation of Perspective Transformations

In Figure 3.3 the e of the Figure 3.2 becomes n; consequently the simple raw projection eqns 3.1 and 3.2 become

$$y = -np_y/p_z \tag{3.4}$$

For x, we likewise have

$$x = -np_x/p_z \tag{3.5}$$

Now, as noted in the previous subsection, OpenGL maps $x = l \mapsto x' = -1, x = r \mapsto x' = +1 and y = b \mapsto y' = -1, y = t \mapsto y' = +1$, see Figure 3.4. We can use a handy formula, namely eqn. 3.6,

$$v = (u - u_0)\frac{v_1 - v_0}{u_1 - u_0} + v_0 \tag{3.6}$$

which maps $u_0 \mapsto v_0 and u_1 \mapsto v_1$, i.e. in the present case $x = l \mapsto x' = -1 and x = r \mapsto x' = +1$ and likewise for y, y'.

For the x mapping, eqn 3.6 gives:

$$\begin{aligned} x' &= (x - l)\frac{1 - (-1)}{r - l} - 1, \\ &= (x - l)\frac{2}{r - l} - 1, \\ &= \frac{2(x - l) - r + l}{r - l}, \\ &= \frac{2x - 2l - r + l}{r - l} \end{aligned} \tag{3.7}$$

So,

$$x' = \frac{2x - (r + l)}{r - l} \tag{3.8}$$

Substituting eqn 3.5 into 3.8 gives,

$$x' = \frac{2n\frac{-p_x}{p_z} - (r + l)}{r - l} \tag{3.9}$$

and there is a similar equation for y',

$$y' = \frac{2n\frac{-p_y}{p_z} - (t + b)}{t - b} \tag{3.10}$$

Thinking ahead to the required form, eqns 3.9 and 3.10 can be rewritten

$$-x'p_z = \frac{2np_x + (r + l)p_z}{r - l} \tag{3.11}$$

and,

$$-y'p_z = \frac{2np_y + (t+b)p_z}{t-b} \tag{3.12}$$

Now p_z has to be mapped; the problem here is that the rasterisation stage needs the reciprocal of $p_z(\frac{1}{p_z})$ rather than p_z so that we need $z' = a(\frac{1}{p_z}) + b$. Eqn 3.6 is still usable if we define $s = \frac{1}{p_z}$ and define the mapping $s = -\frac{1}{n} \mapsto z' = -1 and s = -\frac{1}{f} \mapsto z' = +1$, so eqn 3.6 gives,

$$\begin{aligned} z' &= \frac{(\frac{1}{p_z} + \frac{1}{n})(1+1)}{(-\frac{1}{f} + \frac{1}{n})} - 1, \\ &= \frac{2\frac{fn}{p_z} + 2f}{-n+f} - 1, \\ &= \frac{2fn\frac{-1}{p_z} + 2f}{f-n}, \\ -z'p_z &= \frac{-2fn - 2fp_z + p_z f - p_z n}{f-n} + p_z, \end{aligned} \tag{3.13}$$

and finaly,

$$-z'p_z = \frac{-p_z(f+n) - 2fn}{f-n} \tag{3.14}$$

From the projected 3D point $\mathbf{p'} = (x', y', z')$ (the mapped image of (p_x, p_y, p_z)) we now create the 'homogeneous' coordinates for$\mathbf{p'} = (-x'p_z, y'p_z, -z'p_z, -p_z)$; note the $-p_z$ in the w component. Then summarize eqns. 3.11, 3.13 and 3.14 as

$$-x'p_z = \frac{2np_x}{r-l}p_x + \frac{r+l}{r-l}p_z, \tag{3.15}$$

$$-y'p_z = \frac{2n}{t-b}p_y + \frac{t+b}{t-b}p_z, \tag{3.16}$$

and

$$-z'p_z = -\frac{f+n}{f-n}p_z - \frac{2nf}{f-n} \tag{3.17}$$

The whole projective transformation can now be written in terms of a matrix multiplication and homogeneous coordinates,

$$\begin{bmatrix} -x'p_z \\ -y'p_z \\ -z'p_z \\ w' \end{bmatrix} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{-2nf}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} \tag{3.18}$$

Recall *glFrustum*, where we now use the same symbols as in eqn. 3.18:

$void\, glfrustum(GLdoublel, GLdoubler, GLdoubleb, GLdoublet, GLdoublen, GLdoublef)$.

Figure 3.8 shows the final transformation stages in full context.

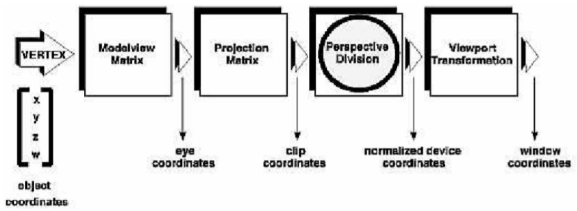


Figure 3.8: Vertex transformation pipeline.

3.5 Orthographic Projection

In an orthographic or parallel projection, the projection rays are parallel to one another and to the camera viewing direction. In addition lack of perspective distortion means that (in OpenGL) true z coordinates can be interpolated linearly (rather than reciprocal (1/z) values).

Consequently all that needs to be done for x and y is a mapping as given by eqn. 3.8 for x,

$$x' = \frac{2x - (r+l)}{r-l}, \tag{3.19}$$

$$= \frac{2x}{r-l} - \frac{r+l}{r-l}, \tag{3.20}$$

and the same for y,

$$y' = \frac{2x}{t-b} - \frac{t+b}{t-b}, \tag{3.21}$$

The z coordinate mapping is $-f \mapsto -1, -n \mapsto +1$, so,

$$z' = \frac{-2z}{f-n} - \frac{f+n}{f-n}, \tag{3.22}$$

We note that eqns. 3.20, 3.21 and 3.22 represent an affine transformation, i.e. scale plus translate. Their matrix representation is given by eqn. 3.23; this is the projection matrix generated by OpenGL function glOrtho:

$void\, glOrtho(GLdoubleleft, GLdoubleright,$
$GLdoublebottom, GLdoubletop, GLdoublenear, GLdoublefar)$.

$$\begin{bmatrix} -x' \\ -y' \\ -z' \\ w' \end{bmatrix} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} \tag{3.23}$$

Now these transformation matrices are used with the algorithm discussed in Chapter 2 to find the correct orthographic projections of the 3D model.

3.6 Reconstruction: 3D back from 2D

We now discuss the phase of reconstruction, i.e. using 2D orthographic projections to develop the corresponding 3D model and build on the ideas discussed in chapter 2 to formulate an implementable algorithm suitable for unlabeled vertices and unlabeled edges.

For convenience, probable 3D vertices, probable 3D edges, probable faces, probable 3D sub-objects are written as *p-vertices, p-edges, pfaces, p-subobjects*, respectively.

3.6.1 Step 1 : 2D vertices and 2D lines

In this step we process the input data and remove certain redundancies. All the points and lines that are input by the user are 2D vertices and 2D lines. First, all the duplicate specifications of these vertices, if any, are removed. Next. since the user is not required to input the intersection points of two straight lines, it is determined if an intersection point exists and if it does, it is added to the list of vertices. Finally, the collinear lines of the same type, namely, solid or dashed, are combined to form one line.

The intersection of two lines is determined using the following algorithm. Let the two lines under consideration be KL and MN. The parametric equations for these lines can be written as:

$$\begin{aligned} x &= x_K + (x_L - x_K)s \\ y &= y_K + (y_L - y_K)s \end{aligned} \tag{3.24}$$

for line KL and

$$\begin{aligned} x &= x_M + (x_N - x_M)t \\ y &= y_M + (y_N - y_M)t \end{aligned} \tag{3.25}$$

for line MN, where s and t are the parameters. The point of intersection, J, satisfies both Eqns. (3.24) and (3.25). One can obtain the values of the parameters s and t at J as:

$$s_J = \frac{(x_N - x_M)(y_M - y_K) - (y_N - y_M)(x_M - x_K)}{(x_N - x_M)(y_L - y_K) - (y_N - y_M)(x_L - x_K)} \tag{3.26}$$

$$t_J = \frac{(x_L - x_K)(y_M - y_K) - (y_L - y_K)(x_M - x_K)}{(x_N - x_M)(y_L - y_K) - (y_N - y_M)(x_L - x_K)} \tag{3.27}$$

If both s_J and t_J are in the range of 0 to 1, then the intersection occurs within the two line segments and is given by

$$\begin{aligned} x_J &= x_k + (x_L - x_K)s_J = x_M + (x_N - x_M)t_j \\ y_J &= y_K + (y_L - y_K)s_j = y_m + (y_N - y_M)t_J \end{aligned} \tag{3.28}$$

This point is then added as a 2D vertex. If $s_J < 0$ or $s_J > 1$ or $t_J < 0$ or $t_J > 1$, then the line segments intersect outside their span and intersection not valid.

3.6.2 Step 2 : Probable 3D vertices

A list of probable 3D vertices is constructed in this step. If any two 2D vertices, belonging to different views, have the same coordinate .value for the shared coordinate axis, then the third view is searched for the 2D vertex which has the same values as the remaining two coordinates from the original two 2D vertices under consideration. If the search is successful then a 3D p-vertex containing the common x, y, z coordinates from three 2D vertices is found. This procedure is carried out for all the 2D vertices.

3.6.3 Step 3 : Probable 3D edges

Recursive in nature, this step involves both the generation of p-edges and checking the validity of p-vertices. If any p-vertex is found to be invalid, that p-vertex is deleted and the procedure goes back to the beginning of this step.

A straight line which connects any two p-vertices is a p-edge provided the projections of this 3D edge can be found in all the three input views. Since this algorithm deals with planar objects, these projections can appear as 2D lines or as a 2D vertex. Whenever this test is satisfied the p-edge formed under consideration is compared with the previously generated p-edges. If the most recently generated p-edge contains any one of the previously generated p-edges, then the most recently generated p-edge is not included in the p-edges table. On the other hand, if any of the previously generated p-edges contain the most recently generated p-edge then all those previously generated p-edges are deleted. The criterion behind this test is not to store overlapping p-edges.

The p-edges are stored in terms of their end points. From this, a table that gives the p-edge numbers to which each p-vertex belongs is created. Since the algorithm deals with solid objects, each p-vertex should belong to at least three p-edges. Ira p-vertex belongs to less than three p-edges, then it is assumed to be a false p-vertex and is deleted. Whenever a p-vertex is deleted, the validity of the p-edges is no longer guaranteed; hence the process returns to the beginning of this step (i.e.. step 3). If none of the p-vertices is deleted then the process advances to the next step.

3.6.4 Step 4 : Probable faces

In this step, a list of probable planar faces is constructed. This step is composed of the following sub-steps:

1. Determination of planar surfaces
2. Generation of p-edge closed loops

3. P-edge loop relationships
4. Formation of p-faces
5. Testing of 2D dashed lines

Determination of planar surfaces:
The list of p-edges is searched to find out a pair of p-edges with a common p-vertex. A planar surface passing through these p-edges can be represented mathematically as:

$$ax + by + cz + d = 0$$

Given KJ and JL as the p-edges with $(x_J, y_J, z_J), (x_K, y_K, z_K) and (x_L, y_L, z_L)$ as the co-ordinates of the p-vertices J, K and L, respectively, the equation of the plane through these points is given as

$$\begin{bmatrix} x - x_J & y - y_J & z - z_J \\ x_K - x_J & y_K - y_J & z_K - z_J \\ x_L - x_J & y_L - y_J & z_L - z_J \end{bmatrix} = 0$$

Then, coefficients a,b,c,d can be determined.

3.6.5 Step 5 : Generation of 3D Object

In this step the p-faces are connected to form p-sub-objects which are then classified as certain and uncertain p-sub-objects. To connect p-faces, the cross-product and dot product between the surface normals are used.

Any p-face is chosen as the starting p-face, With this p-face as the reference p-face, the connecting p-faces are found by making use of Table 1. Next, with any one of the just found p-faces as the reference p-face. its connecting p-faces are found. This procedure continues until all the connecting p-faces are found. At this stage, a p-sub-object is said to have been found. In order to find the next p-sub-object, any p-face that has not been used so far is chosen as the reference p-face and the procedure repeated. When both sides of all the p-faces have been used, the search for all the p-sub-objects is finished. While forming a p-sub-object, if both sides of a p-face are required, then that p-face is assumed to be a false one and is deleted: the process in that case goes back to the beginning of step 5.

Reference p-face	Direction of vector crossproduct w.r.t reference p-edge	Direction of traversal of shared p-edge	face connecting angle	Side to be chosen
o	opposite	same	360-θ	s
o	opposite	opposite	180-θ	o
o	same	same	θ	s
o	same	opposite	180+θ	o
s	opposite	same	θ	o
s	opposite	opposite	180+θ	s
s	same	same	360-θ	o
s	same	opposite	180-θ	s

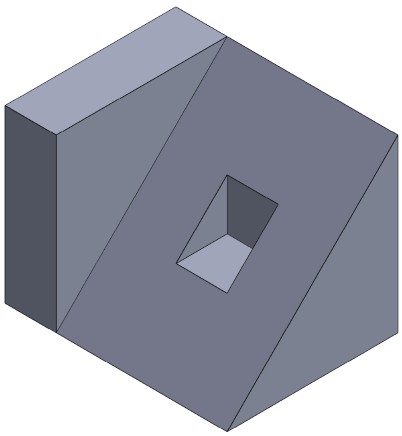
Note: o means the face that bounds the opposite side of the surface normal, s means the face that bounds the same side of the surface normal.

After getting all the 2D lines, collinear 2D lines which are of the same type are joined together and stored as a single 2D line. The regenerated 2D lines are compared with the original 2D lines; if they match exactly, then the assembled p-object is one of the solutions or the only solution.

This set of vertices and edges may contain several **ghosts** (non existing vertices, edges or faces), these can be removed using heuristic approach to find the minimum number of vertices and edges required to match the given projections. To start off, **we first consider the reconstruction problem with labeled vertices and unlabeled edges** which is the case mostly, and then tackle the general problem of unlabeled vertices and unlabeled edges later. An example explaining this approach only for the case of edge reduction is discussed next, where labeled vertices and unlabeled edges are used for reconstruction. This algorithm can be extended for face reduction as well.

3.7 Exemplar problem for 2D to 3D reconstruction

Since real drawings never contain labeled edges, we here try to use a simplified heuristic implementation of algorithm to illustrate how to recover the edges of the wireframe, given all the points and projected edges and remove redundant vertices and edges that came up by the last algorithm discussed. This can be further extended for faces and computation can be done to determine full model. The Algorithm takes as input:



Solid object used for this example

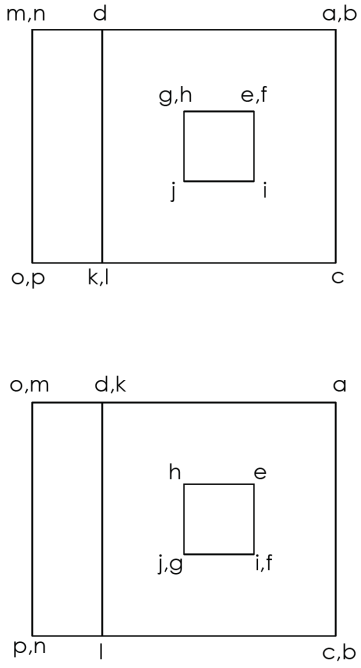
- Projected vertices and which real vertices they correspond to. Real vertices are named using some symbols, and this name is identical in all projections.
- Edges between projected vertices. As a convention, if a projected edge passes over multiple vertices, it is written as a set of smaller edges between two vertices.

Given these, our algorithm proceeds in following steps:

1. True vertex set is extracted. This is possible using the procedure already described, using labels instead of projections.
2. The adjacency matrix is initialized for the complete graph.
3. Iterate over each view:
   (a) Add missing edges, i.e. edges that can be formed by adding to collinear connected edges
   (b) Remove all edges whose projection does not exist on this view from the adjacency matrix
4. Complete graph by heuristics, if required
   (a) Each edge must get at least 3 non-coplanar edges
   (b) Three edges from a vertex are coplanar iff two of them are collinear
   (c) Two edges should not intersect except at vertices.
   (d) No vertex may lie on an edge other than its end points

- No vertex may lie on an edge other than its end points
- The edge is completely covered by another edge

The first of these is guaranteed to be overcome with even two views: two points can be same in at most two coordinates. The second one is a bit harder, eg 3 edges of a cube are not visible even after looking at 3 views. Therefore, some heuristics may be needed. Note that while this does not guarantee a solution, is does so almost always. Consider the following example:



(A) Orthographic projection of the example

Vertices:
{[a,b,(23,3)],[c,(23,16)],
[d,(6,3)],[e,f,(18,8)],[g,h,
(11,8)],[i,(18,11)],[j,(11,
11)],[k,l,(6,16)],[o,p,(3,
16)],[m,n,(3,3)]}

Edges:
{(0,1),(1,7),(7,2),(2,0),
(3,4),(4,6),(6,5),(5,3),
(7,9),(8,2),(8,9)]

Vertices:
{...}

Edges:
{(0,1),(1,7),(7,2),(2,0),
(3,4),(4,6),(6,5),(5,3),
(7,9),(8,2),(8,9)]

(B) Vertices and edges in top and front views

Note that the naming does not follow some of the conventions.
The first step is skipped as it is rather easy.

We start off with the adjacency matrix: (to show the power of this approach, let us assume that excessive edges have been declared probable and in the extreme case - all)

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p
a																
b	1															
c	1	1														
d	1	1	1													
e	1	1	1	1												
f	1	1	1	1	1											
g	1	1	1	1	1	1										
h	1	1	1	1	1	1	1									
i	1	1	1	1	1	1	1	1								
j	1	1	1	1	1	1	1	1	1							
k	1	1	1	1	1	1	1	1	1	1						
l	1	1	1	1	1	1	1	1	1	1	1					
m	1	1	1	1	1	1	1	1	1	1	1	1				
n	1	1	1	1	1	1	1	1	1	1	1	1	1			
o	1	1	1	1	1	1	1	1	1	1	1	1	1	1		
p	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	

Now, we look at the top view, and eliminate all edges not present there:

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p
a																
b	1															
c	1	1														
d	1	1														
e																
f					1											
g					1	1										
h					1	1	1									
i					1	1										
j							1	1	1							
k			1	1												
l			1	1							1					
m	1	1		1												
n	1	1											1			
o			1								1	1	1	1		
p			1								1	1	1	1	1	

And then the front view:

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p
a																
b	1															
c	1	1														
d	1															
e																
f					1											
g						1										
h					1		1									
i					1	1										
j							1	1	1							
k				1												
l			1	1							1					
m	1			1												
n		1											1			
o											1		1	1		
p			1									1	1	1	1	

Now we begin applying heuristics.

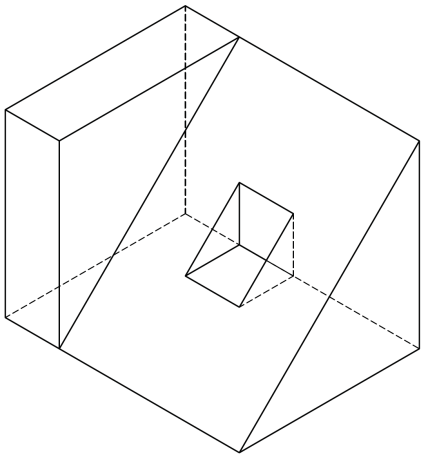
The second constraint is violated at m as the edges MN, MO and MP cannot all exist. Also, it is violated at N as MN,NO and NP cannot all exist together. Further, MP and no cannot coexist as the intersect. So, exactly one of MN, MO, MP; exactly one of MN, NO, NP and atleast one of MP, NO must be dropped. An exhaustive search can give that the only stable solution is when MP and NO are dropped. Note that this would be unnecessary if an LHS view was provided and simplified somewhat if a RHS view was provided.

The fourth constraint is violated by AM and CP. These two are also dropped. At this point all constraints are satisfied and our algorithm stops.

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p
a																
b	1															
c	1	1														
d	1															
e																
f					1											
g						1										
h					1		1									
i					1	1										
j							1	1	1							
k				1												
l			1	1							1					
m				1												
n		1											1			
o											1		1			
p												1		1	1	

It is easy to see that this is in fact, the correct adjacency matrix.

The generated wireframe is shown below.



Generated 3D model's edges

As this heuristic approach provides us a correct way to obtain edges of the model, it can be extended to obtain the required faces. This edge limited implementation gives us reasonable confidence to proceed further with our model and implement it to build the required program.

References

1. A matrix-based approach to reconstruction of 3D objects from three orthographic views:
   $http://ieeexplore.ieee.org/document/883948/$

2. DN Berstein, The number of roots of a system of equations, Functional Analysis and its applications, 1975, 9(3), pp. 1-4

3. OpenGL : SuperBible

4. 3D Projections Wikipedia:
   $https://en.wikipedia.org/wiki/3D_projection$

5. ScratchPixel OpenGL Perspective projections:
   $https://www.scratchapixel.com/lessons/3d-basic-rendering/perspective-and-orthographic-projection-matrix/building-basic-perspective-projection-matrix$

6. MultiResolution Mathematics of 3D projections:
   $http://www.multiresolutions.com/strule/jon/www-jgcampbell-com/bscgp1/grmaths.pdf$

7. Construction of 3D Solid Objects from Orthographic Views:
   $https://ac.els-cdn.com/0097849389900125/1-s2.0-0097849389900125-main.pdf?_tid=687e87a8-f9d3-11e7-aeef-00000aacb360\&acdnat=1516007385_9bbc8703eeef965232455a47c69c334$

8. Fast 3D solid model reconstruction: $https://ac.els-cdn.com/S0010448597000547/1-s2.0-S0010448597000547-main.pdf?_tid=02520032-f9dc-11e7-a247-00000aacb362\&acdnat=1516011079_ef49322c2d3a1bc2136796b1399ff1b9$