Viewing Three-Dimensional Objects
The Synthetic Camera and Simple Perspective

This document describes some basic ideas and methods for displaying three-dimensional objects or shapes on a two-dimensional computer screen. Attaining full visual realism is a very complex problem which must take into account where the object is located relative to the observer, where the object appears to be located and what parts of it are visible to the observer, and how the object appears to the observer (color, surface texture, illumination, etc.). In this document, we look at some basic approaches to the first two issues.

We'll approach the topic by looking for answers to three questions:

Where is the object in relation to the observer? (The Synthetic Camera Model)
The answer to the question of "*where?*" always requires the specification of coordinates of points relative to an appropriate coordinate system. Once you know the coordinates of certain points on an object, you can join them by lines to form edges or by curves to form curved edges or surface grids of various sorts and so, develop the detailed shape of the object. We start off here by assuming that we can specify the object via a set of points with known fixed coordinates relative to a **world coordinate system**, consisting of the x-, y-, and z-axes.

Example
The "house" shape sketched in Figure 1 is determined by the location of the 14 labelled points at which various combinations of straight line edges meet. The first step is to adopt a coordinate system such as the one shown in the figure with origin at the back left lower corner of the house. The actual coordinates of the 14 points with respect to that coordinate system can be determined from given dimensions of the house (we need to start with actual physical measurements of the object to be rendered).

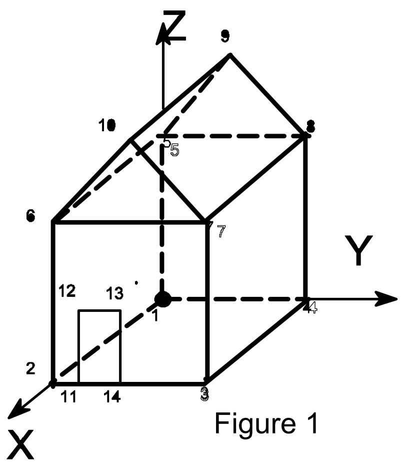


Figure 1

To have a specific set of coordinates with which to illustrate the use of the formulas and methods presented in this document, we'll adopt the dimensions of this house in the real world as shown in the front and side views in Figure 2 below to the left (assume whatever units of measurement you wish). This gives homogeneous coordinates (x, y, z, h) for the 14 points as listed in the table to the right of Figure 2.

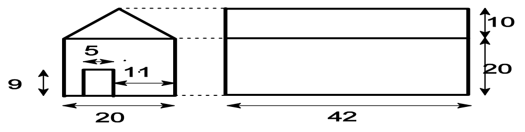


Figure 2

TABLE 1: World Coordinates

	x	y	z	h
1	0	0	0	1
2	42	0	0	1
3	42	20	0	1
4	0	20	0	1
5	0	0	20	1
6	42	0	20	1
7	42	20	20	1
8	0	20	20	1
9	0	10	30	1
10	42	10	30	1
11	42	4	0	1
12	42	4	9	1
13	42	9	9	1
14	42	9	0	1

All that we need to complete the full specification of this simple geomtric shape is the list of pairs of points joined by straight line edges. For the record here, there are 20 such edges:

1 to 2	5 to 6	1 to 5	6 to 10	9 to 10
2 to 3	6 to 7	2 to 6	7 to 10	11 to 12
3 to 4	7 to 8	3 to 7	5 to 9	12 to 13
4 to 1	8 to 1	4 to 8	8 to 9	13 to 14

In practice, this data would be stored as an array of pairs of whole numbers. Then to generate any image of this object, just have a computer program generate a line from the location of point 1 to the location of point 2 (the first pair), then a line from the location of point 5 to the location of point 6 (the second pair listed in the first row of the table), and so on, until all 20 edges have been drawn. As well, recall that the 4 column by 14 row array of coordinates boxed in Table 1 forms the matrix of homogeneous coordinates of the 14 vertices determining the shape of this object. The object as a whole can be scaled, rotated, translated, skewed, etc. by multiplying the appropriate 4 x 4 transformation matrix (or sequence of transformation matrices) from the right to generate new *world coordinates* for the vertices of the transformed object.
◆◆◆

The synthetic camera model provides a way of generating the image of this object from any specific viewpoint . Think of having the freedom of sighting the house through a camera lens from any point of view: how would you determine the image you expect to see? Obviously, this visual image will depend on your location relative to the location of the house, the direction in which you are looking, your orientation (if you are hanging upside down, the house itself will appear to be upside down!), and so forth. If you are above the house and looking upwards, you'd see nothing. If you were above the house and looking downwards, you'd see the roof, and perhaps one or two of the side walls.

To keep track of all of these features we need to define a second coordinate system which will provide the image of the object.[♣] We will call this the **image coordinate system** to distinguish it from the world coordinate system described above. The coordinate axes in this system will have the fairly conventional labels (u, v, n) to further distinguish them from the (x, y, z) coordinates of the world coordinate system, and for this reason, this new system is also often called the **uvn-coordinate** system. (Of course, we need to use homogeneous coordinates in actual calculations, and in either coordinate system, we'll indicate the homogeneous coordinate generically by h.) Roughly speaking, the **n**-direction will be the line of sight, and the plane formed by the **u** and **v** axes will be the plane in which the image of the object is to be formed, with **v** being "up" in the view, and **u** being "rightwards" in the view. To illustrate, Figure 3 shows an arrangement in which the house is being viewed from above and to the right.

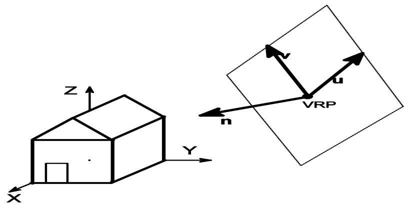


Figure 3

Think of the uv-plane as being a transparent viewport through which you are viewing the object (this means that the observer's eye will have a negative n-coordinate). Then, think of replacing this transparent viewport by a computer video screen. The problem we solve here is to determine what image would have to be drawn on this computer screen/viewport so that the image you see on the screen would be the same as the image you'd see if the screen was a transparent viewport through which you viewed the actual object.

Each point on the object corresponds to a unique set of (x, y, z) coordinates and a corresponding unique set of (u, v, n) coordinates. We first need to develop a way of computing (u, v, n) coordinates of a point from its known (x, y, z) coordinates. In doing this and subsequent calculations to produce an image on the computer screen, we must be very careful to maintain the distinction between these two coordinate systems. Our notation must be quite precise (though the brief description of the synthetic camera approach given here avoids the worst of the proliferation of abbreviations and acronyms you'll find in some standard reference books on the topic).

Obviously[♠] we must first specify where the viewplane is located and how it is oriented. First, pick a location for the VRP or viewplane reference point, which will be the origin of the uvn coordinate system. We will

[♣] Why do we need a second coordinate system? Because the object itself will not change when we change our view of it. The world coordinate system and the coordinates of points of the object relative to that world coordinate system is unaffected by us simply changing how we view the object.
[♠] It's probably not obvious at all right now, but it should be obvious by the end of this section. The word "obvious" is used in this particular document to indicate ideas that you should take some time to visualize or understand intuitively. There is some formal mathematics to follow, but it is just making precise ideas that are not too difficult to visualize mentally. In this

represent the coordinates of this point by the symbol **r** = (r_x , r_y , r_z), the coordinates of this point with respect to the (x, y, z)-coordinate system. Then, pick the **n**-direction. This is specified by the vector **n**, which should be normalized if necessary so that it has length 1. This will make subsequent formulas appear simpler.

Example
Following the example of the house shape above, we might select as our VRP, a point a bit in front of and to the right of the right-front edge of the building. For example, if we came out 28 units in front (along x), 20 units right (along y) and 5 units above ground (along z), for our VRP, the point **r** would have coordinates (x = 42 + 28, y = 20 + 20, z = 0 + 5) = (70, 40, 5); that is, $r_x = 70$, $r_y = 40$ and $r_z = 5$.

To get an **n**, we might decide that the line of sight should be from the VRP towards the lower left-front corner of the house at location (42, 0, 0). Then, **n** would be a unit vector in the direction of the vector with head at this point, and tail at the VRP: having components

(42 - 70, 0 - 40, 0 - 5) = (-28, -40, -5)

Since the length of this vector is

$$\sqrt{(-28)^2+(-40)^2+(-5)^2} = \sqrt{2409} \cong 49.08156$$

the desired vector **n** here is

$$\mathbf{n} = \left(\frac{-28}{\sqrt{2409}},\ \frac{-40}{\sqrt{2409}},\ \frac{-5}{\sqrt{2409}}\right) \cong \left(-0.57048, -0.81497, -0.10187\right)$$

You can see that this is a vector pointing backwards along x, y, and z — the sort of thing you would expect if you were looking back and down to the left at the house from a location up and to the right in front of the house. You should be able to verify easily that the vector **n** given above has length = 1 to the precision the numbers are stated. You should also be able to see that it would be very easy to calculate **n** for any other selected location for the VRP, and any other point on or near the house specifying the line of sight.

◆◆◆

This gives us the components of the vector **n** = (n_x , n_y , n_z). Here (n_x , n_y , n_z) are the components of **n** with respect to the (x, y, z)-coordinate system. The next step is to work out the vector **v**, giving the upwards direction in the view. Generally, here, we pick an approximate upwards direction, say **up**, and then adjust it so that it is perpendicular to **n**. This adjustment procedure is just finding the part of **up** which is perpendicular to **n**. This is given by the formula (see the notes on vectors):

$$\mathbf{up}_\perp = \mathbf{up} - (\mathbf{up}\bullet\mathbf{n})\,\mathbf{n} \qquad \text{(VIEW-1)}$$

(This formula is correct only if **n** is a unit vector.) The vector **v** is then obtained by normalizing $\mathbf{up}_\perp$.

Example
Continuing the house example, we might calculate **v** by trying to maintain the upwards direction the same as in the real world as much as possible. So, pick **up** = (0, 0, 1), a vector pointing straight up in the world coordinate system. Then,

$$\mathbf{up}\bullet\mathbf{n} \cong (0, 0, 1) \bullet(-0.57048, -0.81497, -0.10187) = (0)(-0.57048) + (0)(-0.81497) + (1)(-0.10187)$$
$$\cong -0.10187$$

So,

$$(\mathbf{up}_\perp)_x = (\mathbf{up})_x - (-0.10187)\, n_x = 0 - (-0.10187)(-0.57048) \cong -0.05812$$
$$(\mathbf{up}_\perp)_y = (\mathbf{up})_y - (-0.10187)\, n_y = 0 - (-0.10187)(-0.81497) \cong -0.08302$$
$$(\mathbf{up}_\perp)_z = (\mathbf{up})_z - (-0.10187)\, n_z = 1 - (-0.10187)(-0.10187) \cong 0.989622$$

This vector $\mathbf{up}_\perp \cong (-0.05812, -0.08302, 0.989622)$ has length approximately 0.994798, which is not quite 1. Dividing the components by this value gives the (x, y, z)-components of the vector **v**:

$$\mathbf{v} \cong (-0.05812/0.994798, -0.08302/0.994798, 0.989622/0.994798)$$
$$\cong (-0.05842, -0.08346, 0.994798)$$

case, It will be helpful to develop a "mental image" of viewing a scene through a camera lens (as suggested by the term "synthetic camera") and try to relate the mathematical symbols introduced here to actual features of the photographic composition process. Thus, the VRP, which is the origin of the uvn coordinate axes system, corresponds to the position of the camera. The VPN or viewplane normal, corresponds to the direction in which the camera is pointed. The VRP is an actual location in space, so is specified by a set of coordinates relative to some coordinate system. The VPN is a direction is space, and so must be specified not by coordinates of a point, but components of a vector.

You can easily verify that this vector has length of 1, and that it is perpendicular to **n**. Notice that It is very close in orientation to the original **up**, pointing mostly along z, but somewhat backwards along x and y, so that the result is perpendicular to **n**.

◆◆◆

This leaves the task of computing the components of the vector **u,** which must be perpendicular to both **n** and **v** and have length of 1. Recall from the summary of properties of vectors covered earlier in the course that the so-called *cross product* of two vectors produces a third vector perpendicular to both of the original two vectors. Thus, here we can simply compute **u** = **n** × **v**. Doing the cross product in this orientation will give the uvn-axes oriented as shown in Figure 3 making the uvn-system left-handed. Further, since **n** and **v** are already mutually perpendicular unit vectors, **u** is guaranteed to be a unit vector as well.

Example
Substituting the components of **u** and **v** obtained earlier into the formulas for the components of a vector cross product , we get

$$\begin{aligned} u_x &= n_y v_z - n_z v_y \cong (-0.81497)(0.994798) - (-0.10187)(-0.08346) \cong -0.81923 \\ u_y &= n_z v_x - n_x v_z \cong (-0.10187)(-0.05842) - (-0.57048)(0.994798) \cong 0.573462 \qquad \text{(VIEW-2)} \\ u_z &= n_x v_y - n_y v_x \cong (-0.57048)(-0.08346) - (-0.81497)(-0.05842) \cong 0.0 \end{aligned}$$

You can verify directly here that **u** ≅ (-0.81923, 0.573462, 0.0) has length of 1 and is perpendicular to both **n** and **v**. Notice that **u** is pointing to the right (positive direction in y) and backwards (negative direction in x), and is a horizontal vector (the z-component is zero — in some sense, a matter of coincidence here).

◆◆◆

Now that we have a way of calculating the vectors **u**, **v**, **n**, with respect to the world coordinate system, it is relatively easy to come up with a formula relating the (u, v, n) coordinates of a point to its (x, y, z) coordinates and vice versa. Although our goal is to be able to calculate (u, v, n) coordinates from (x, y, z) coordinates, it is easiest the understand the process by looking at the reverse calculation. For a particular point in three-dimensional space, we have:

$$[\,x, y, z, 1] = [\,u, v, n, 1] \bullet \mathbf{M} \bullet \mathbf{T}_r \qquad \text{(VIEW-3)}$$

where

[x, y, z, 1] are the homogeneous coordinates of the point with respect to the (x, y, z) - axes,
[u, v, n, 1] are the homogeneous coordinates of the point with respect to the (u, v, n) - axes,

$$\mathbf{M} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

is a 4 x 4 transformation matrix with first three rows containing the components of the vectors **u**, **v**, and **n** in order, and

$$\mathbf{T}_r = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ r_x & r_y & r_z & 1 \end{bmatrix}$$

is a 4 x 4 transformation matrix representing a translation from the origin of the world coordinate system to the location of the VRP (the origin of the viewing coordinate system).

Remark:
This is where the "intuitively obvious" bit finally expires! Right?
Well, not really. Think of a point which has coordinates (1, 0, 0) in the uvn-system — that is, it is on the u-axis one unit from the origin. Then, to get to that point from the origin of the world coordinate system, we need to travel from (x, y, z) = (0, 0, 0) first to the VRP at (r_x , r_y , r_z), and then a distance of one unit along **u**,

which breaks down to a distance u_x in the x-direction, combined with a distance u_y in the y-direction, combined with a distance u_z in the z-direction — this is just what saying $\mathbf{u} = (u_x, u_y, u_z)$ means. Thus, to get to the point (u, v, n) = (1, 0, 0) from the origin (0, 0, 0) in the (x, y, z)-system, we need to travel a distance $r_x + u_x$ in the x-direction, a distance $r_y + u_y$ in the y-direction, and a distance $r_z + u_z$ in the z-direction. But this just means that the point (u, v, n) = (1, 0, 0) corresponds to

$$(x, y, z) = (r_x + u_x, r_y + u_y, r_z + u_z)$$

which is exactly the result given by the formula above. What the multiplication by the matrix **M** does is calculate how far along x-, y-, and z- respectively, the point (u, v, n) is from the VRP. Then the translation $\mathbf{T}_r$ adds on how far along x-, y-, and z- respectively one has to move to get from the origin of the world coordinate system to the VRP. The result has to be the (x, y, z) coordinates of the point with coordinates (u, v, n) in the viewing system.

■■■

We used square brackets in formula (VIEW-3) above because it really is a matrix equation, indicating that the 1 x 4 matrix of homogeneous coordinates of the point in the (x, y, z) system is obtained by multiplying the 1 x 4 matrix of homogeneous coordinates of the point in the viewing system by the 4 x 4 matrices **M** and $\mathbf{T}_r$, in that order from the right. Thus, to get a formula for [u, v, n, 1] in terms of [x, y, z, 1], we simply multiply formula (VIEW-3) from the right by the appropriate inverse transformation matrices:

$$\begin{aligned}[x, y, z, 1] \bullet (\mathbf{T}_r)^{-1} \bullet (\mathbf{M})^{-1} &= [u, v, n, 1] \bullet \mathbf{M} \bullet \mathbf{T}_r \bullet (\mathbf{T}_r)^{-1} \bullet (\mathbf{M})^{-1} \\ &= [u, v, n, 1] \bullet \mathbf{M} \bullet (\mathbf{M})^{-1} \\ &= [u, v, n, 1]\end{aligned}$$

since $\mathbf{T}_r \bullet (\mathbf{T}_r)^{-1}$ simplifies to an identity matrix and so can be dropped, and then $\mathbf{M} \bullet (\mathbf{M})^{-1}$ also simplifies to an identity matrix. Thus, we have initially that

$$[u, v, n, 1] = [x, y, z, 1] \bullet (\mathbf{T}_r)^{-1} \bullet (\mathbf{M})^{-1}$$

However, $(\mathbf{T}_r)^{-1}$ is just a translation from the VRP to the origin of the world coordinate system, and so is the matrix

$$\mathbf{T}_{-r} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -r_x & -r_y & -r_z & 1 \end{bmatrix}$$

and because of the special form of **M**, having rows which are mutually perpendicular normalized vectors (**M** is said to be an *orthogonal matrix* for you jargon junkies), the inverse of **M** is just its transpose (swap rows and columns):

$$\mathbf{M}^{-1} = \mathbf{M}^T = \begin{bmatrix} u_x & v_x & n_x & 0 \\ u_y & v_y & n_y & 0 \\ u_z & v_z & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Thus,

$$[u, v, n, 1] = [x, y, z, 1] \bullet \mathbf{T}_{-r} \bullet \mathbf{M}^T = [x, y, z, 1] \bullet \mathbf{A}_{WV} \qquad \text{(VIEW-4)}$$

where

$$\mathbf{A}_{WV} = \mathbf{T}_{-r} \bullet \mathbf{M}^T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -r_x & -r_y & -r_z & 1 \end{bmatrix} \begin{bmatrix} u_x & v_x & n_x & 0 \\ u_y & v_y & n_y & 0 \\ u_z & v_z & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} u_x & v_x & n_x & 0 \\ u_y & v_y & n_y & 0 \\ u_z & v_z & n_z & 0 \\ -\mathbf{r} \bullet \mathbf{u} & -\mathbf{r} \bullet \mathbf{v} & -\mathbf{r} \bullet \mathbf{n} & 1 \end{bmatrix} \qquad (5)$$

is the overall transformation matrix for converting (x, y, z) coordinates into (u, v, n) coordinates (the notation $\mathbf{A}_{WV}$ is similar to that used by Hill, page 392). The matrix elements on the bottom row of the final result are the usual vector dot products:

$$-\mathbf{r}\bullet\mathbf{u} = -(r_x u_x + r_y u_y + r_z u_z)$$

and so on.

Before completing the example, we need to summarize what's been done. We started by working out the components of the image coordinate system axes, **u**, **v**, and **n**. These vectors describe the orientation of the "view" of the object to be represented. Using these vectors and the coordinates of the VRP, the origin of the image coordinate system, we then set up the matrix $\mathbf{A}_{WV}$, a 4 x 4 matrix. Multiplying our matrix of homogeneous world coordinates by $\mathbf{A}_{WV}$ from the right will then give a matrix of homogeneous coordinates of the same points, but now with respect to the **uvn-**coordinate axes.

Example

We can now complete our "view" of the house for the particular viewpoint developed in earlier steps of this example here. First, form the matrix $\mathbf{A}_{WV}$**:**

$$-\mathbf{r}\bullet\mathbf{u} \cong -(r_x u_x + r_y u_y + r_z u_z) = -[(70)(-0.81923) + (40)(0.573462) + (5)(0)] \cong 34.40774$$

$$-\mathbf{r}\bullet\mathbf{v} \cong -(r_x v_x + r_y v_y + r_z v_z) = -[(70)(-0.05842) + (40)(-0.08346) + (5)(0.994798)] \cong 2.453612$$

$$-\mathbf{r}\bullet\mathbf{n} \cong -(r_x n_x + r_y n_y + r_z n_z) = -[(70)(-0.57048) + (40)(-0.81497) + (5)(-0.10187)] \cong 73.04168$$

Thus,

$$\mathbf{A}_{WV} = \begin{bmatrix} u_x & v_x & n_x & 0 \\ u_y & v_y & n_y & 0 \\ u_z & v_z & n_z & 0 \\ -\mathbf{r}\bullet\mathbf{u} & -\mathbf{r}\bullet\mathbf{v} & -\mathbf{r}\bullet\mathbf{n} & 1 \end{bmatrix} \cong \begin{bmatrix} -0.81923 & -0.05842 & -0.57048 & 0 \\ 0.573462 & -0.08346 & -0.81497 & 0 \\ 0 & 0.994798 & -0.10187 & 0 \\ 34.40774 & 2.453612 & 73.04168 & 1 \end{bmatrix}$$

We need to multiply this 4 x 4 transformation from the right onto the 14 x 4 matrix of world coordinates given on the first page of this document, to get the 14 x 4 matrix of uvn-coordinates for the view:

	x	y	z	h
1	0	0	0	1
2	42	0	0	1
3	42	20	0	1
4	0	20	0	1
5	0	0	20	1
6	42	0	20	1
7	42	20	20	1
8	0	20	20	1
9	0	10	30	1
10	42	10	30	1
11	42	4	0	1
12	42	4	9	1
13	42	9	9	1
14	42	9	0	1

-0.81923	-0.05842	-0.57048	0
0.573462	-0.08346	-0.81497	0
0	0.994798	-0.10187	0
34.40774	2.453612	73.04168	1

=

u	v	n	h	
34.4	2.5	73.0	1	1
0.0	0.0	49.1	1	2
11.5	-1.7	32.8	1	3
45.9	0.8	56.7	1	4
34.4	22.3	71.0	1	5
0.0	19.9	47.0	1	6
11.5	18.2	30.7	1	7
45.9	20.7	54.7	1	8
40.1	31.5	61.8	1	9
5.7	29.0	37.9	1	10
2.3	-0.3	45.8	1	11
2.3	8.6	44.9	1	12
5.2	8.2	40.8	1	13
5.2	-0.8	41.7	1	14

Now, to get the picture, just set up coordinate axes (the u-axis pointing to the right, much like the usual x-axis, and the v-axis pointing upwards, much like the usual y-axis is drawn). The scale along the u-axis must run from a minimum of 0.9 (points 2 and 6) to a maximum of 45.9 (points 4 and 8), or say, 50 in round numbers. The scale along the v-axis needs to run from a minimum of -1.7 (point 3) to +31.5 (point 9) or say, -5 to +35 in round figures. Then, plot all 14 points using (u, v)-coordinates from the table above, and join them appropriately to form the 20 edges comprising the outline shape. The result will be something like:

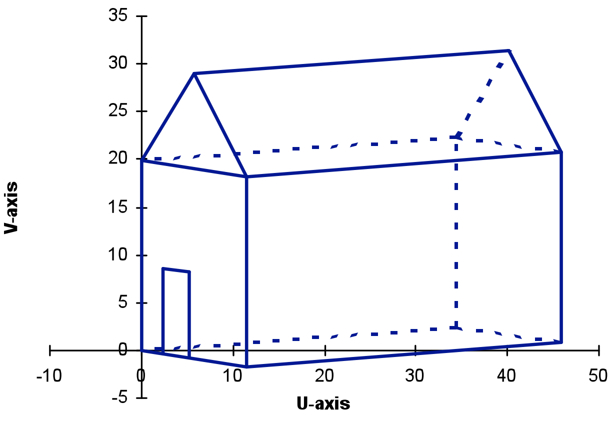


Figure 4

We haven't labeled the fourteen vertices in this figure to reduce clutter, but you should be able to identify each of them. For example, vertex #10, the roof peak at the front of the house, has (u, v) coordinates in this view of (u = 5.7, v = 29.0), which is where it is plotted in Figure 4 (just a bit below v = 30, and about midway between u = 0 and u = 10). The dotted lines indicate edges which are known to be out of sight on the sides of the building opposite our implicit viewpoint. This was done here by visual inspection, but before this document is finished, you'll see how to determine by computation which faces (and so, which edges) of an object like this are not visible, and so be able to program the automatic drawing of dotted lines for invisible edges if desired.

■■■

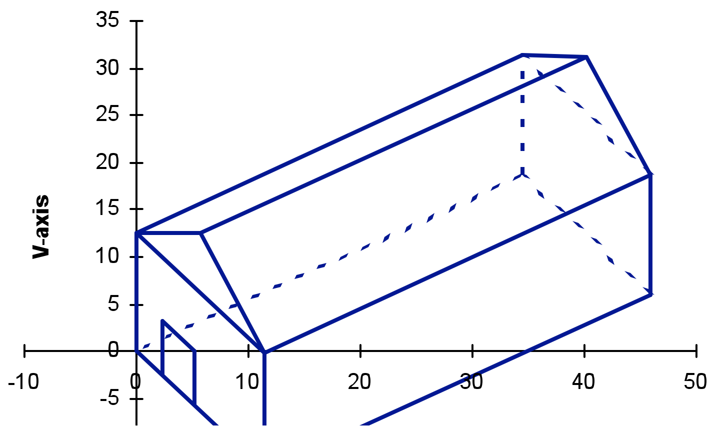
Take a minute to satisfy yourself that Figure 4 is the sort of image you'd expect to see if you were looking at the house from a location above, in front, and to the right of the right-front edge indicated in Figure 1. At the same time, you should have a feeling that the image is not quite right in some respect. In fact, it almost appears as if the right-side face of the building is getting wider as you move from front to back. With a ruler, you can confirm that this is not the case. However, because of the way our eyes work, we expect this right-side face to appear to narrow as we sight from front (closer) to back (farther) along the house. This is such a fundamental feature of the way we see the world that our brain automatically interprets the scene to mean that the right-side wall is actually getting wider as you move back along the house! To overcome this mental distortion or optical illusion or whatever you want to call it, we need to introduce some simulation of perspective.

<u>Exercise</u>:
Here's an example you can use to test if you understand the computations described and illustrated so far. Determine the image of the house when the VRP is quite a bit higher, say, at the location (70, 40, 60). Select the line of sight (the direction of **n** to be from this VRP to the left front lower corner of the house at (42, 0, 0) as above, and use **up** = (0, 0, 1) as before.

Solution
Here we'll give just the matrix A_{WV} , the viewing coordinates, and the final view, so you can check your calculations:

-0.81923	-0.44480	-0.36196	0
0.57346	-0.63542	-0.51709	0
0.00000	0.63119	-0.77563	0
34.40774	18.68141	92.55870	1

u	v	n	h	
34.4	18.7	92.6	1	*1*
0.0	0.0	77.4	1	*2*
11.5	-12.7	67.0	1	*3*
45.9	6.0	82.2	1	*4*
34.4	31.3	77.0	1	*5*
0.0	12.6	61.8	1	*6*
11.5	-0.1	51.5	1	*7*
45.9	18.6	66.7	1	*8*
40.1	31.3	64.1	1	*9*
5.7	12.6	48.9	1	*10*
2.3	-2.5	75.3	1	*11*
2.3	3.1	68.3	1	*12*
5.2	0.0	65.7	1	*13*
5.2	-5.7	72.7	1	*14*



The VRP is so high relative to the house that now both halves of the roof become visible. We didn't have to decide independently whether this would happen or not — it was an automatic consequence of the computations. This begins to demonstrate the power of this relatively simple synthetic camera model. Having developed some program code to generate a single view of this sort, all we need to do is surround it with a loop that successively repeats the calculations for a sequence of positions for the VRP and you have what amounts to a program that simulates something like a "fly-past".

✹✹✹

What does the image of the object actually look like at the location of the uv-plane?
This is the second of the three major questions addressed in this document. It's answer involves the introduction of a perspective transformation, which we will describe in its simplest terms here.

Recall that our overall objective is to determine how to draw an image on the uv-plane (the viewplane) so that the image drawn exactly mimics the image we'd see if the uv-plane was a transparent viewport through which we actually viewed the object. Since we can calculate (u, v, n) coordinates from world coordinates, we now know where the points determining the shape of the object are located relative to the uv-plane. What this step requires is simply to calculate where the lines of sight from the observer's eye to the points on the object actually intersect the viewplane.

Until now, we've just had some vague notion that the object is located on one side of the uv-plane (the side for which the n-coordinate is positive), and the observer is on the other side of the uv-plane (negative n coordinates) looking through the uv-plane at the object. Now we must take explicit and detailed account of the position of the observer's eye. Since the view is being constructed in the uvn-coordinate system, This means specifying the (u, v, n) coordinates of the observer's eye: (e_u, e_v, e_n). Figure 5 below diagrams the situation.

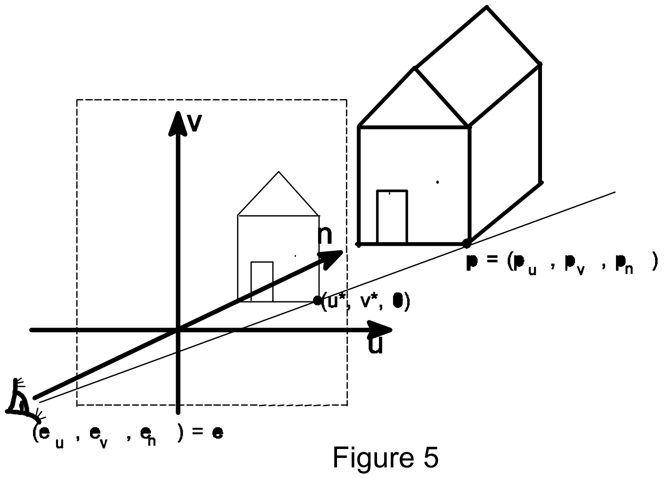

v
n
u
$(u^*, v^*, 0)$
$\mathbf{p} = (\mathbf{p}_u, \mathbf{p}_v, \mathbf{p}_n)$
$(e_u, e_v, e_n) = \mathbf{e}$

Figure 5

The dotted-line rectangle in the figure represents the uv-plane in which the image of the house is being constructed. The actual house itself is located on the positive n side of the uv-plane, and the observer is located on the negative n side of the uv-plane. The image drawn on the uv-plane is set up so that points on the image fall on the same line of sight as the point on the actual house being imaged. This is illustrated for a single line of sight in the figure. The image of the lower right front corner of the house, actually located at (p_u, p_v, p_n) will be drawn on the uv-plane at $(u^*, v^*, 0)$, the point at which the line of sight from the location of the observer's eye at $\mathbf{e} = (e_u, e_v, e_n)$ to the point $\mathbf{p} = (p_u, p_v, p_n)$ passes through the uv-plane. To construct the image of the entire house then requires determination of the $(u^*, v^*, 0)$ coordinates for each of the points on/in the object/image, plotting these points, and joining them by the necessary straight line segments in this case.

The easiest way to compute u* and v* is to make use of the parametric form of the equation of the straight line passing through points **e** and **p**. We can write this equation in vector form:

$$\rho(t) = \mathbf{e}\bullet(1 - t) + \mathbf{p}\bullet t \qquad \text{(VIEW-6)}$$

which actually stands for three separate equations, one for each of the three coordinate directions:

$$\begin{aligned} \rho_u &= e_u(1 - t) + p_u t \\ \rho_v &= e_v(1 - t) + p_v t \\ \rho_n &= e_n(1 - t) + p_n t \end{aligned} \qquad \text{(VIEW-7)}$$

Here, (ρ_u, ρ_v, ρ_n) are the coordinates of a point on this line of sight, and by varying t from $-\infty$ to $+\infty$, the entire line is constructed. In particular, t = 0 gives ρ at **e**, and t = 1 gives ρ at **p**. The notation ρ(t) used on the left side of formula (VIEW-6) indicates that ρ depends on the value of t, and different values of t give different points ρ along this line of sight.

To determine the coordinates $(u^*, v^*, 0)$ of the point where this line of sight cuts the uv-plane, we need to determine the coordinates of the point ρ on this line for which the n-coordinate is 0 (the uv-plane corresponds to n = 0). But, the point on this line for which $\rho_n = 0$ must be the point at which we have

$$e_n(1 - t) + p_n t = 0$$

This is easily solved to give

$$t^* = \frac{e_n}{e_n - p_n} \qquad \text{(VIEW-8)}$$

To get u* and v* we just have to substitute this value of t into the formulas for ρ_u and ρ_v, respectively:

$$u^* = e_u\left(1 - \frac{e_n}{e_n - p_n}\right) + p_u\left(\frac{e_n}{e_n - p_n}\right) = \frac{p_u e_n - e_u p_n}{e_n - p_n}$$

$$v^* = e_v\left(1 - \frac{e_n}{e_n - p_n}\right) + p_v\left(\frac{e_n}{e_n - p_n}\right) = \frac{p_v e_n - e_v p_n}{e_n - p_n} \qquad \text{(VIEW-9)}$$

In the special case that the observer's eye is on the n-axis (so $e_u = 0$ and $e_v = 0$), these formulas simplify to:

$$u^* = \frac{p_u e_n}{e_n - p_n} = \frac{p_u}{1 - \frac{p_n}{e_n}}, \qquad v^* = \frac{p_v e_n}{e_n - p_n} = \frac{p_v}{1 - \frac{p_n}{e_n}} \qquad \text{(VIEW-10)}$$

These formulas look sort of ugly at first glance, but they were obtained with relatively straightforward algebra, and are easy to use in principle, by just substituting known numbers p_u, p_v, p_n, e_u, etc. To make them even more easy to incorporate into a program, it is usual to rewrite them in the form of (you probably guessed it!) a matrix transformation:

$$[\, u^{*\prime} \;\; v^{*\prime} \;\; n^{*\prime} \;\; h^{*\prime} \,] = [\, p_u \;\; p_v \;\; p_n \;\; 1 \,] \bullet \mathbf{P} = [\, p_u \;\; p_v \;\; p_n \;\; 1 \,] \bullet \mathbf{M}_s \bullet \mathbf{M}_p \qquad \text{(VIEW-11)}$$

where

$$\mathbf{P} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -\frac{e_u}{e_n} & -\frac{e_v}{e_n} & 1 & -\frac{1}{e_n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \text{(VIEW-12)}$$

is the overall "perspective" transformation (often referred to generically as a "viewing transformation"), which some author's find convenient to write as a product of the two simpler transformation matrices:

$$\mathbf{M}_s = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -\frac{e_u}{e_n} & -\frac{e_v}{e_n} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ and } \mathbf{M}_p = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -\frac{1}{e_n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \text{(VIEW-13)}$$

In practice, formula (VIEW-11) introduces a number of steps into the process of creating the image of the object. We already have an m x 4 matrix of homogeneous uvn-coordinates for the m points describing the object. Then,

(1) set up the transformation matrix **P** either directly using formula (VIEW-12) or if more convenient (see below) using formulas (13).

(2) multiply **P** onto the m x 4 matrix of uvn-coordinates from the right, producing an m x 4 matrix of "coordinates" with rows of the form [u*' v*' n*' h*']. The primes on each symbol are to distinguish these coordinates from the u* and v* which we need to draw the image. The problem here is that in general these are not properly homogeneous coordinates, since the fourth component, h*', will turn out generally to be different from 1. Before any drawing can take place, homogeneity must be restored.

(3) Divide each row of the matrix produced in the previous step by the value of its fourth element, the h*'. This will produce an m x 4 matrix with each row having the form [u*, v*, n*, 1].

(4) Draw the image using the (u*, v*) coordinates for each point generated in the previous step.

The formulas given above have a number of important features and consequences. Before listing some of these, we illustrate the steps in the calculation to produce a perspective view of the house used in the earlier examples.

<u>Example</u>
Take the uvn-coordinate system used in producing Figure 4 above (Figure 4 is really an orthographic projection of the house onto the uv-plane), and produce a perspective image assuming that the observer is on the n-axis (so $e_u = 0$ and $e_v = 0$) at position $e_n = -20$ (that it, the observer is 20 units away from the origin along the n-axis on the opposite side of the uv-plane from the house). Since $e_u = 0$ and $e_v = 0$, the matrix $\mathbf{M}_s$ above is an identity matrix and so the matrix **P** is identical to the matrix $\mathbf{M}_p$, both differing from an identity matrix only in that the fourth element of the third row is $-1/e_n = -1/(-20) = +0.05$. Thus, step(2) implementing formula (11) gives:

$$\begin{array}{c|cccc|}
 & u & v & n & h \\ \hline
1 & 34.4 & 2.5 & 73.0 & 1 \\
2 & 0.0 & 0.0 & 49.1 & 1 \\
3 & 11.5 & -1.7 & 32.8 & 1 \\
4 & 45.9 & 0.8 & 56.7 & 1 \\
5 & 34.4 & 22.3 & 71.0 & 1 \\
6 & 0.0 & 19.9 & 47.0 & 1 \\
7 & 11.5 & 18.2 & 30.7 & 1 \\
8 & 45.9 & 20.7 & 54.7 & 1 \\
9 & 40.1 & 31.5 & 61.8 & 1 \\
10 & 5.7 & 29.0 & 37.9 & 1 \\
11 & 2.3 & -0.3 & 45.8 & 1 \\
12 & 2.3 & 8.6 & 44.9 & 1 \\
13 & 5.2 & 8.2 & 40.8 & 1 \\
14 & 5.2 & -0.8 & 41.7 & 1 \\ \hline
\end{array}
\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0.05 \\ 0 & 0 & 0 & 1 \end{bmatrix}
=
\begin{array}{|cccc|c}
u^{*\prime} & v^{*\prime} & n^{*\prime} & h^{*\prime} & \\ \hline
34.41 & 2.45 & 73.04 & 4.65 & 1 \\
0.00 & 0.00 & 49.08 & 3.45 & 2 \\
11.47 & -1.67 & 32.78 & 2.64 & 3 \\
45.88 & 0.78 & 56.74 & 3.84 & 4 \\
34.41 & 22.35 & 71.00 & 4.55 & 5 \\
0.00 & 19.90 & 47.04 & 3.35 & 6 \\
11.47 & 18.23 & 30.74 & 2.54 & 7 \\
45.88 & 20.68 & 54.70 & 3.74 & 8 \\
40.14 & 31.46 & 61.84 & 4.09 & 9 \\
5.73 & 29.01 & 37.88 & 2.89 & 10 \\
2.29 & -0.33 & 45.82 & 3.29 & 11 \\
2.29 & 8.62 & 44.90 & 3.25 & 12 \\
5.16 & 8.20 & 40.83 & 3.04 & 13 \\
5.16 & -0.75 & 41.75 & 3.09 & 14 \\ \hline
\end{array}$$

The 14 x 4 matrix on the left is the matrix of uvn-coordinates used to produce Figure 4. The 4 x 4 matrix in the center is the matrix **P** in this instance, and the matrix on the right is the product of these two. Notice that none of the values in the h*' column are 1's, as they need to be before we can plot points. So, following step 3 in the procedure outlined above, we divide each row by the value in the fourth column. Thus, the first row needs to be divided by 4.65, the second by 3.45, and so on, giving in the end tje results in the table below to the left.

	u*	v*	n*	h*
1	7.4	0.5	15.7	1.0
2	0.0	0.0	14.2	1.0
3	4.3	-0.6	12.4	1.0
4	12.0	0.2	14.8	1.0
5	7.6	4.9	15.6	1.0
6	0.0	5.9	14.0	1.0
7	4.5	7.2	12.1	1.0
8	12.3	5.5	14.6	1.0
9	9.8	7.7	15.1	1.0
10	2.0	10.0	13.1	1.0
11	0.7	-0.1	13.9	1.0
12	0.7	2.7	13.8	1.0
13	1.7	2.7	13.4	1.0
14	1.7	-0.2	13.5	1.0

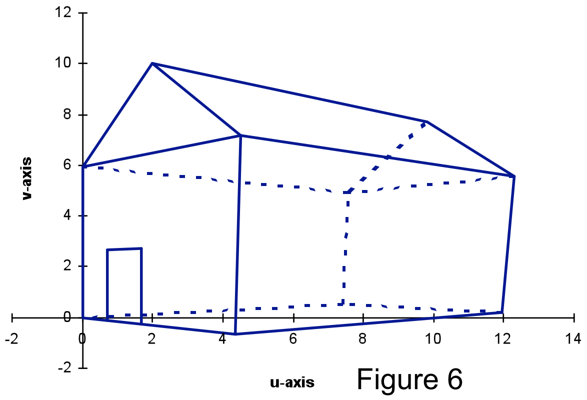


Figure 6

Notice that the h* coordinates are now all equal to 1.0. To produce Figure 6, just plot the (u*, v*) values from the table above for each point, and join the points using the information specifying edges in the object. As with Figure 4 earlier, we have manually dotted edges which would be hidden. Notice that this image does have the "foreshortening" effect we expect to see in perspective views, in that parallel lines seem to be converging to a point in the far distance. More about that shortly.

■■■

There is a great deal more that could be said about perspective views and transformations, and the books by Hill and by Rogers and Adams in the course reference text list go into considerably more detail than these notes do on the topic. Before leaving this issue and moving on to the third matter covered in this particular document, we mention just a few important features and tie up some loose ends.

(1) When you look at formulas (VIEW-9) or (VIEW-10) you see that there will be trouble if either $e_n = 0$ or if $e_n = p_n$, since then division by zero results. $e_n = 0$ means that the observer is sitting right in the

view plane, and so amounts to something like trying to watch a movie with your eyeball against the screen — not an arrangement one would normally need to employ. $e_n = p_n$ arises when the point being viewed and the observer are in the same plane parallel to the viewplane. In this case, the line of sight is parallel to the uv-plane and never intersects it, and so the point in question is not in the image. This situation can arise if the observer is considered to be inside some extensive object (for example, programs which simulate walkthroughs of buildings). If this is likely to occur in a particular application, the program would have to explicitly guard against attempts to include such points in the view.

(2) This viewing transformation really only makes sense for points for which e_n and p_n have different signs (above, the eye was located at a negative value of e_n and the vertices of the object viewed were all located at positive n coordinates). This means that p_n/e_n will be negative and so the denominator is formulas (VIEW-10), for example, will be numbers bigger than 1, since we are subtracting a *negative* value from 1. In fact, these denominators will be larger values for larger values of p_n, indicating points further away from the observer. Thus u* will be smaller than p_u and v* will be smaller than p_v, so that points further from the observer will tend to appear closer to the line of sight. This is the familiar perspective narrowing effect.

(3) The perspective transformation destroys information. Every point on the line of sight (such as the one illustrated in Figure 5) gives the same values of (u*, v*), and so plots to the same point in the perspective view. Thus, unlike the geometrical transformations studied earlier in the course (rotations, translations, etc.), we cannot "undo" a perspective transformation to obtain the (u, v, n) coordinates of a point given just the corresponding (u*, v*) coordinates of its image. This means that we need to retain some information in addition to values of (u*, v*) to be able to distinguish between points on the same line of sight in a particular view.

(4) Notice that in Figure 5 and formulas (VIEW-9) and (VIEW-10), we give meaning only to u* and v*, coordinates of points on the viewplane itself, suddenly in formula (VIEW-11) and the example, a quantity n* shows up, which (you can confirm) is given by

$$n^* = \frac{p_n}{1 - \frac{p_n}{e_n}} \qquad \text{(VIEW-14)}$$

when $e_u = 0$ and $e_v = 0$. This quantity is often called the **pseudodepth** of the point. It has the useful property that it increases in value with the value of p_n, and so these numbers give us relative positions of the points along the n-axis. This information is needed in some procedures for determining visibility.

What parts of the object are actually visible to the observer?

This document is getting rather long, but we need to at least briefly address one further issue in a very introductory way. In the various "views" of the house above, we've used dotted lines to indicate edges and faces of the object that aren't really visible to the observer. Nothing in the calculations leading to these wireframe images indicates which edges really are hidden by parts of the object closer to the observer. We just used our mental image of the situation to decide which edges to show as dotted, and then manually changed the line style in the drawing program used to produce the actual graphic images for this document.

Such an approach is unsatisfactory in general. As our viewpoint of an object changes some edges/faces which were visible will become partially or fully obliterated, which others previously not in sight, will become visible. Since changes of viewpoints may happen automatically, or at least at a program user's choice, it is necessary to have computational procedures that will allow the program to decide for itself which edges/faces are visible and which are not.

This is the start of the so-called *"hidden surface removal"* or *HSR* problem (some people also refer to it as the hidden line removal problem for obvious reasons) — one of the most complex in computer graphics. Over the past more than thirty years, several very sophisticated approaches have been developed to deal with aspects of the problem, most of them beyond the scope of this course. What we will do here is briefly explain a very simple approach that works with so-called convex shapes.

A convex shape is one in which any line segment between any two points on the surface of the object lies entirely inside the object. This has the consequence that any given face on the object is either totally visible or totally invisible. No face can be partially occluded by other parts of the object. The house we've been using as an example in this document is a convex object. If we added a chimney to some position on the roof, it would cease to be convex (since a line joining a point on the surface of the chimney with another point on the roof would pass through some points outside of the house), and there would be points of view from which the chimney would block our view of parts of the image of the roof say, but not all. There are relatively simple methods for handling such a situation, but we will not consider them here.

The principle we use here is simply this: for a convex object, faces which are “facing” towards the observer are visible; faces which are “facing” away from the observer are not visible. Thus, visibility can be determined if we have a way of determining the direction in which a face of the object is facing. Again, there are several general ways to approach this problem and we will just describe one of them here which is relatively easy to set up for simple objects and doesn’t require use of more than some simple vector calculations. Note that simply looking at, say, Figure 1, and recording that the front of the house faces outwards in the +x direction, the right side faces out in the +y direction, etc., won’t do the job in general, because any transformation of the object (rotation in particular) could invalidate such specific information. Further, we really need to make decisions regarding visibility from within the viewing (uvn) coordinate system and so it would be convenient to have information allowing us to distinguish between inside and outside of faces which doesn’t depend too closely on using a specific coordinate system (though such an approach is feasible, since we always have ways of relating quantities in different coordinate systems).

Here, proceed as follows. Every face of the object is a polygon — a closed sequence of straight lines joining a sequence of points. For example, the polygon in Figure 7 consists of five lines joining five points, each distinguished by identifying numbers (denoted p_1, p_2, etc. in the figure). When this polygon is a face on a solid object, we can distinguish the “inside” side of the face from its “outside” side by recording the order of the vertices as we move around the polygon. In the method explained here, we will specify the orientation of a face by ***listing the vertices forming the polygonal face in order counterclockwise from a viewpoint outside of the object***. Thus, we start by generating an additional collection of information, one record for each face of the object, giving this list of vertices.

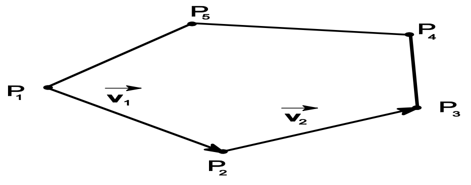


Figure 11

Example
Refer to the house sketched in Figure 1. It is made up of seven faces. Number the front face 1, right face 2, back face 3, left face 4, bottom face 5, right roof 6, and left roof 7. For this simple example, the face orientation data is easily created by visual inspection to be:

Face	vertices				
1 (front)	2	3	7	10	6
2 (right)	3	4	8	7	
3 (back)	1	5	9	8	4
4 (left)	1	2	6	5	
5 (bottom)	1	4	3	2	
6 (right roof)	7	8	9	10	
7 (left roof)	5	6	10	9	

Thus, standing outside the house and looking at its front face (the one with the door), you see a five-sided polygon shape with vertices being the points numbered 2, 3, 7, 10, 6, going counterclockwise around the face. Your vertex list for each face can start with any of the its vertices, though in the table we’ve begun with the lowest-numbered vertex in each case.

◆◆◆

Note that this information is unaffected by normal geometric transformations of the object or by changes in viewpoint. If you are outside of the object and you can see a particular face, the counterclockwise order of vertices on that face will be the same.

Now, do the following calculation for each face to determine its visibility. The table of the sort shown in the example just above gives the identity of points p_1, p_2, p_3, etc. for each face. Compute the edge vectors $\mathbf{v}_1$ (from p_1 to p_2) and $\mathbf{v}_2$ (from p_2 to p_3) in the usual way by subtracting the coordinates of the tail point from the coordinates of the head point of each vector:

$$(\mathbf{v}_1)_u = p_{2u} - p_{1u}; \qquad (\mathbf{v}_2)_u = p_{3u} - p_{2u}$$

$(\mathbf{v}_1)_v = p_{2v} - p_{1v}$; $(\mathbf{v}_2)_v = p_{3v} - p_{2v}$

$(\mathbf{v}_1)_n = p_{2n} - p_{1n}$; $(\mathbf{v}_2)_n = p_{3n} - p_{2n}$

Notice that we are using uvn-coordinates here: not xyz-coordinates, since the view is being constructed in the uvn-system, and specifically the uv-plane; not u*v*n*-coordinates, since they don't really represent geometric locations in space, but places where points have to be plotted in the uv-plane to get a realistic image. If you use perspective coordinates for this calculation, you will find odd faces appear in your picture mysteriously in your picture when they obviously are behind the object, or disappear leaving a gap in the picture when they obviously should be in sight. The uvn-coordinates are the ones which contain precise geometric information about the location of the points determining each face.

Then, a basic result of vector analysis says that the vector cross product $\mathbf{v} = \mathbf{v}_1 \times \mathbf{v}_2$ will produce a vector $\mathbf{v}$ which is perpendicular to both $\mathbf{v}_1$ and $\mathbf{v}_2$, oriented in the same sense as the coordinate system in use. Since our uvn-system here is a left-handed system, and since $\mathbf{v}_1$ and $\mathbf{v}_2$ have been constructed to be in counterclockwise order, this means $\mathbf{v}$ will be perpendicular to the face, and pointing ***inwards*** (try it with your left hand - imagine $\mathbf{v}_1$ being shifted so its tail coincides with the tail of $\mathbf{v}_2$ and curl your fingers from $\mathbf{v}_1$ to $\mathbf{v}_2$ — notice that your left thumb will be pointing in the direction you know to be inwards from the face into the object.). We will say that **v** is the ***inward normal*** vector for the face. However, since vectors indicate only direction and not position, what the inward normal vector really represents is the direction you need to be sighting in order to see the outside of the face (it is the direction that "looks" directly away from the inside of the face, so it must be the direction that "looks" directly at the outside of the face).

Now, form a line-of-sight vector, **Lofs**, from the eye to any point on the face, say point p_1:

$(\mathbf{Lofs})_u = p_{1u} - e_u$

$(\mathbf{Lofs})_v = p_{1v} - e_v$

$(\mathbf{Lofs})_n = p_{1n} - e_n$

where (e_u, e_v, e_n) are the uvn-coordinates of the observer's eye, as used earlier in forming the perspective view. If this line-of-sight vector is pointing in the same general direction as the inward normal (in fact, within 90 degrees of the inward normal) then the face is visible to the observer. To determine whether **Lofs** is within 90^o of **v**, we just need to determine whether the cosine of the angle between the two vectors is positive or negative (from basic trigonometry, cos θ is positive for θ between 0^o and 90^o, and negative for θ between 90^o and 180^o. But the sign of the cosine of the angle between two vectors is the same as the sign of the dot product of the two vectors. So here, just form

$$\gamma = \mathbf{Lofs} \bullet \mathbf{v} = (\mathbf{Lofs})_u\, \mathbf{v}_u + (\mathbf{Lofs})_v\, \mathbf{v}_v + (\mathbf{Lofs})_n\, \mathbf{v}_n$$

If γ is positive in value, the face is visible, and if γ is negative in value, the face is not visible. If $\gamma = 0$ exactly, it means the line-of-sight to the face is exactly perpendicular to the inward normal, meaning that the face is visible exactly edge on (for practical purposes, you can probably consider this to be equivalent to the face being invisible).

This completes the analysis except for a few remarks to tie up some loose ends. However, first, we illustrate how to apply this procedure to several faces on our exemplary house.

Example

To illustrate the above calculations, we consider the two halves of the roof of the house, faces 6 and 7. We use the uvn-system set up in earlier steps of this example, and locate the eye as before at $\mathbf{e} = (e_u, e_v, e_n) = (0, 0, -20)$. In reference to the diagram in Figure 1, you can calculate that this puts the eye at (x, y, z) ≅ (81.4, 56.3, 7.0), or to the right front of the house, about midway up its height. Thus, the right side of the roof (face 6 in the present numbering system) is expected to be visible, whereas the left side of the roof (face 7) should not be visible.

The first three vertices forming face 6 are vertices 7, 8, and 9, from the table just above. Thus, $\mathbf{v}_1$ is the vector pointing from vertex 7 to vertex 8, and $\mathbf{v}_2$ is the vector pointing from vertex 8 to vertex 9. Then, tabulating the calculations, we get:

	u	v	n
point 7:	11.47	18.23	30.74
point 8:	45.88	20.68	54.70
point 9:	40.14	31.46	61.84
$\mathbf{v}_1$	45.88 - 11.47 = 34.41	20.68 - 18.23 = 2.45	54.70 - 30.74 = 23.96
$\mathbf{v}_2$	40.14 - 45.88 = -5.73	31.46 - 20.68 = 10.78	61.84 - 54.70 = 7.13
$\mathbf{v} = \mathbf{v}_1 \times \mathbf{v}_2$	-240.85	-382.76	385.07

Thus,

$$\gamma = \mathbf{Lofs} \bullet \mathbf{v} \cong (11.47)(-240.85) + (18.23)(-382.76) + (50.74)(385.07) \cong 9801.47.$$

Since this is positive, we conclude face 6 is visible from the selected viewpoint (as we know to be the case).

For face 7, the first three vertices are points 5, 6, and 10. Repeating the same calculations in this case gives the result:

	u	v	n
point 5:	34.41	22.35	71.00
point 6:	0.00	19.90	47.04
point 10:	5.73	29.01	37.88
	0.00 - 34.41 =	19.90 - 22.35	47.04 - 71.00
$\mathbf{v}_1$	-34.41	= -2.45	= -23.96
	5.73 - 0.00 =	29.01 - 19.90	37.88 - 47.04
$\mathbf{v}_2$	5.73	= 9.11	= -9.17
$\mathbf{v} = \mathbf{v}_1 \times \mathbf{v}_2$	240.85	-452.87	-299.50
	34.41 - 0.00 =	22.35 - 0.00 =	71.00 - (-20.00) =
Lofs	34.41	22.35	91.00

In this case,

$$\gamma = \mathbf{Lofs} \bullet \mathbf{v} \cong (34.41)(240..85) + (22.35)(-452.87) + (91.00)(-299.50) \cong -29090$$

Being negative, this indicates that face 7 is not visible to the observer, confirming the fact which we already knew in this simple example.

■■■

Finally, some miscellaneous remarks about this calculation:

(1) This is a calculation involving directions, and the correct interpretation of the final result requires that all steps involving direction in some sense be carried out precisely as detailed above — no directional detail can be changed or assumed to be irrelevant. Thus, concluding that γ positive indicates a face is visible is based on: the uvn-system if left-handed, the three vertices listed are in order counterclockwise when the face is viewed from outside the object, the vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ have the orientation indicated in Figure 7, and the normal to the face, $\mathbf{v}$ is calculated as $\mathbf{v}_1 \times \mathbf{v}_2$ and not the other way around. For every single change you make in any of these directional details in the calculation, the γ criterion changes sign.

(2) Compared to algorithms that can handle non-convex objects and deal with partially hidden faces, the calculations shown above are very short and simple. Although this method is completely adequate only for a relatively rarely encountered type of object, it is used as a first step in more sophisticated methods that can handle partially hidden faces to eliminate from consideration faces which are totally out of sight of the observer.

(3) The method described and illustrated above tells you which faces of the object are visible. Once a face is determined to be visible, its image can be drawn by constructing a polyline (or just a sequence of line segments) from one vertex to another around the face, using whatever viewing coordinates are appropriate (either the actual uvn-coordinates for an orthographic projection or perspective coordinates for a perspective view). This is possible because you have access to a list of points forming the vertices around the face. This method doesn't work if you wish to draw hidden edges as, say, dotted lines, or if there are lines to be drawn which aren't edges of visible faces (for example, the outline of the door on the front face of our house). The first problem may be most easily dealt with by drawing the entire figure in dotted lines, and then redrawing visible edges in solid lines as the visible faces are detected. Another way to deal with both of these problems would be to expand the information stored specifying the lines in the figure to include not just the sequence numbers of the two points forming endpoints of the line segment (the information, for example, in the table at the top of page 2 above), but two more integers for each line, specifying the sequence numbers of the faces for which each line segment acts as an edge. For lines embedded within a face , these two numbers would both be the sequence number of that face. Then, when a face was determined to be visible, mark any line forming an edge of that face as visible (say, via an array of flags). After scanning all of the faces for visibility, all of the visible lines will be marked, and those which aren't visible will be unmarked, and both kinds can be represented in the picture as desired.

(4) The most difficult part of this approach is getting the ordered list of vertices around each face. Any mistakes in developing that list will result in erroneous pictures being produced. For simple examples such as the seven-faced house used in these notes, the face-vertex list can be developed by inspection in just a few minutes, and errors become obvious and are easily corrected. However, for objects involving large numbers of intricately shaped faces, developing this face-vertex list correctly may be a very frustrating step. Other approaches are available which develop and track directional information about faces in an object based on specifying initial inward or outward normals (or derived information) for each face, and then transforming this information as the object itself is transformed.