COMP30019 Graphics and Computation
Sample exam question solutions

Department of Computing and Information Systems,
The University of Melbourne

1 Conversion of image coordinates

$$\begin{aligned} x &= j \\ y &= (R-1)-i \end{aligned}$$

Though this last one is sometimes seen as

$$y = R-(i+1)$$

I think the first form is preferable. Derivation and reverse conversion should be obvious.

2 Image offset

This works out the same as regular array-indexing address calculation, mapping a 2D array position into linear addresses. The pixel at row i, column j has Ci pixels in front of it from previous rows, and j pixels in front of it on the same row. So it's $Ci+j$ pixels from the start of the pixel data, that's $k(Ci+j)$ or $Cki+kj$ bytes. Counting the H bytes of the header, it's therefore $H+Cki+kj$ bytes from the beginning of the file. This should be pretty straightforward—just make sure that you understand what's involved.

3 Image rescale

For simplicity, assume that the source and destination images are stored respectively in explicit 2D arrays `a` and `b`. Normally, these pixel arrays would be accessed through pointers.

We basically need to for-loop over all the pixels of `b`, rescale the coordinates back to where they would have come from in `a`, and interpolate a value at this position to put into `b`.

```
int a[WIDTH][HEIGHT], b[WIDTH][HEIGHT];    int x2, y2;    double x1, y1;
for( y2 = r2-1; y2 >= 0; y2-- ) {
    for( x2 = c2-1; x2 >= 0; x2-- ) {
        x1 = (double) x2 * c1 / c2;    y1 = (double) y2 * r1 / r2;
        b[x2][y2] = interpolate( a, x1, y1 );
    }
}
```

I've made the loops run backwards (just for fun, to make the point that for most operations the order of visiting pixels is irrelevant—though remember on most machines a loop termination test against zero is faster than a test against some other value). Also, the three statements inside the loop could all have been combined into one, dispensing with `x1` and `x2`.

4 Colour choice

There are a number of problems with this:

focus Edges between regions of different colour cannot be brought to sharp focus, because of chromatic aberration in the eye. This can cause eye strain.

edges The visual subsystem that detects edges between regions responds only to intensity differences, not to differences purely of colour. So such edges would not be well perceived.

motion Similarly, our motion detection subsystem also responds only to intensity, so the motion of the soccer players would not be well perceived either.

colour blindness A significant fraction of the population is red-green colour blind, and hence may not be able to see the difference between the players and the background.

The solution is to make the difference one of intensity as well as of colour—say by making the background dark green instead, and perhaps also by drawing a black outline around the players.

5 Video images

The basic problem is that we don't have a enough lines, and we have too many frames per second. The simplest solution is to replicate every 5th line coming through in each frame. That is, if the lines coming in are ABCDE, we output the lines ABCDEE. This gives us 105 extra lines, a total of 630, and we can just drop the extra 5 lines to leave 625. We also just drop every 6th frame coming in, to reduce the number of frames from 30 per second to 25 per second. The main problem with this is that the picture and motion is distorted on a small scale, but it comes out about right over all. On the input side, the computation just requires a single line of storage. However, additional lines of buffers may be needed on the input side, and additional whole images of buffers on the output side, in order to make timing work out properly.

Going back the other way, we have too many lines, and too few frames. We can do a similar trick, but this time we drop every 6th incoming line. This drops us down to 521 lines, then we can tack on an extra 4 black lines. We also replicate every 5th frame, to bring up the frame rate. The computation needs at least an entire image stored on the input side (so the entire frame can be replicated when needed), but again may need additional whole image buffers on the input and output sides to make all the timing work out properly. Again, shape and motion will be distorted a little on the small scale but overall will be about right.

The issue of how much input and output buffering is needed for the sake of timing is one I haven't given enough thought to. You might like to think about it more yourself.

More complex (and more accurate) methods for converting between the two T.V. standards can be based on spatial and temporal interpolation.

6 The robot's camera

(a) From the position of the point's projection in the image, all we can infer is the line of sight it lies along. However, since it's a point on the ground plane, we know it must lie at the unique point where this line of sight intersects the ground plane.

A reasonable coordinate system puts the 3D origin on the ground directly below the camera's optical centre, with the Y axis up, the X to the robot's right, and the Z axis going backwards. The camera-cantered system has the same X and Z, but we have to subtract h (the height of the robot's camera) from camera Y coordinates in order to get world Y. So, in terms of world coordinates, our projection equations are:

$$\frac{x}{f} = \frac{X}{Z} \tag{1}$$

$$\frac{y}{f} = \frac{Y-h}{Z} \tag{2}$$

In this system, the ground plane is given by $Y = 0$. Substituting this into Equation 2 gives

$$\frac{y}{f} = \frac{-h}{Z}, \text{ or}$$

$$Z = \frac{-hf}{y} \tag{3}$$

Knowing Z from Equation 3, we can compute X by substitution into Equation 1.

$$X = \frac{Zx}{f} = -\frac{hx}{y} \tag{4}$$

And of course, $Y = 0$, so we have all three coordinates of this point in space.

A minor variation is to make the robot-centred coordinates the same as the camera-centred coordinates. In this case the ground plane has instead the equation $Y = -h$, but the derivation is similar.

(b) We need to be able to assume that the object is resting on the ground (not flying through the air), and that its point of contact with the ground is detectable (for example, no extensive overhangs that block view of the object's base).

(c) It's the *horizon* line. Look at Equation 3 and consider what happens as $y \to 0$. Notice that nothing above this line could possibly come from something on the ground. This is the horizon for an infinite flat earth—not quite the same as the horizon for our spherical earth, but close.

(d) Then this method breaks down, but the question is, How much? If we had an accurate topographical map of the terrain, we could instead compute the intersection of the line of sight with the ground surface. Harder to do, but the same principle. Failing that, if we could put upper and lower bounds in the ground surface, we could feed these bounds through the equations to derive bounds on the 3D position of a seen ground point. And failing even that, for points that appear in the image vertically lined up, we can at least infer that the higher-up one is qualitatively further away than the lower-down one. (This may not hold for extremely pathological ground surfaces, but then I wouldn't send my robot to such a place.)

7 Composition of a scaling transformation

We do this by translating everything so that the point at $\begin{pmatrix} c_x & c_y \end{pmatrix}$ arrives at the origin. This is a translation by $\begin{pmatrix} -c_x & -c_y \end{pmatrix}$. Then we scale about the origin, and translate back again, that is by $\begin{pmatrix} c_x & c_y \end{pmatrix}$. Expressed in matrices, this combined transformation is

$$\mathbf{M} = \mathbf{T}_2\mathbf{S}\mathbf{T}_1$$

where

$$\begin{aligned}
\mathbf{T}_1 &= \begin{pmatrix} 1 & 0 & -c_x \\ 0 & 1 & -c_y \\ 0 & 0 & 1 \end{pmatrix} \\
\mathbf{S} &= \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \\
\mathbf{T}_2 &= \begin{pmatrix} 1 & 0 & c_x \\ 0 & 1 & c_y \\ 0 & 0 & 1 \end{pmatrix}
\end{aligned}$$

So

$$\begin{aligned}
\mathbf{S}\mathbf{T}_1 &= \begin{pmatrix} s_x & 0 & -c_x s_x \\ 0 & s_y & -c_y s_y \\ 0 & 0 & 1 \end{pmatrix} \\
\mathbf{T}_2\mathbf{S}\mathbf{T}_1 &= \begin{pmatrix} s_x & 0 & c_x(1-s_x) \\ 0 & s_y & c_y(1-s_y) \\ 0 & 0 & 1 \end{pmatrix}
\end{aligned}$$

In homogeneous coordinates, the corner points are respectively

$$\begin{pmatrix} 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & 1 & 1 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 1 \end{pmatrix}$$

or any multiples of these. The centre of the square is the point $\begin{pmatrix} 1/2 & 1/2 & 1 \end{pmatrix}$, and we're scaling by 2 both horizontally and vertically (s_x and s_y). Plugging this into our symbolic matrix gives

$$\mathbf{M} = \begin{pmatrix} 2 & 0 & -1/2 \\ 0 & 2 & -1/2 \\ 0 & 0 & 1 \end{pmatrix}$$

Multiplying this onto our four corner points (on the right) gives respectively

$$\begin{pmatrix} -1/2 & -1/2 & 1 \end{pmatrix} \quad \begin{pmatrix} 3/2 & -1/2 & 1 \end{pmatrix} \quad \begin{pmatrix} 3/2 & 3/2 & 1 \end{pmatrix} \quad \begin{pmatrix} -1/2 & 3/2 & 1 \end{pmatrix}$$

With, in this simple case, the obvious conversion back into regular 2D coordinates (just drop the extra 1—but it isn't always this simple).

Basically, all these transformations are a linear change in coordinates. When fed through the equation of a line, we still get the equation of a line.

8 Transformation by composition

Steps:

1. Translate A to origin

$$\mathbf{T_1} = \begin{bmatrix} 1 & & -2 \\ & 1 & -2 \\ & & 1 \end{bmatrix}$$

2. Scale. Reflect about the x axis.

$$S_x = 1$$

$$S_y = -1$$

3. Scale. Use AB and $A^{'}B^{'}$ to determine scale in x and AG and $A^{'}G^{'}$ to determine scale in y.

$$S_x = \frac{\sqrt{2^2+2^2}}{2^2+0^2} = \sqrt{2}$$
$$S_y = \frac{\sqrt{1^2+1^2}}{0^2+1^2} = \sqrt{2}$$

4. Scale. Reflect about the x axis.

$$S_x = 1$$
$$S_y = -1$$

$$\mathbf{T_4} = \begin{bmatrix} 1 & & \\ & -1 & \\ & & 1 \end{bmatrix}$$

5. Rotate by $\theta = 45\,\text{deg}$

$$\begin{aligned} \mathbf{T_2} &= \begin{bmatrix} cos(\theta) & -sin(\theta) & \\ sin(\theta) & cos(\theta) & \\ & & 1 \end{bmatrix} \\ &= \begin{bmatrix} 1/\sqrt{2} & -1/\sqrt{2} & \\ 1/\sqrt{2} & 1/\sqrt{2} & \\ & & 1 \end{bmatrix} \end{aligned}$$

6. Translate to A'.

$$\mathbf{T_5} = \begin{bmatrix} 1 & & 2 \\ & 1 & 4 \\ & & 1 \end{bmatrix}$$

The combined transformation matrix (CTM) is determined by multiplying the matrices together:

$$\begin{aligned} \mathbf{CTM} &= T_5T_4T_3T_2T_1 \\ &= \begin{bmatrix} 1 & & 2 \\ & 1 & 4 \\ & & 1 \end{bmatrix} \begin{bmatrix} 1/\sqrt{2} & -1/\sqrt{2} & \\ 1/\sqrt{2} & 1/\sqrt{2} & \\ & & 1 \end{bmatrix} \begin{bmatrix} \sqrt{2} & & \\ & \sqrt{2} & \\ & & 1 \end{bmatrix} \begin{bmatrix} 1 & & \\ & -1 & \\ & & 1 \end{bmatrix} \begin{bmatrix} 1 & & -2 \\ & 1 & -2 \\ & & 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & & 2 \\ & 1 & 4 \\ & & 1 \end{bmatrix} \begin{bmatrix} 1 & -1 & \\ 1 & 1 & \\ & & 1 \end{bmatrix} \begin{bmatrix} 1 & & \\ & -1 & \\ & & 1 \end{bmatrix} \begin{bmatrix} 1 & & -2 \\ & 1 & -2 \\ & & 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & & 2 \\ & 1 & 4 \\ & & 1 \end{bmatrix} \begin{bmatrix} 1 & -1 & \\ 1 & 1 & \\ & & 1 \end{bmatrix} \begin{bmatrix} 1 & & -2 \\ & -1 & 2 \\ & & 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & & 2 \\ & 1 & 4 \\ & & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & -4 \\ 1 & -1 & \\ & & 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 1 & -2 \\ 1 & -1 & 4 \\ & & 1 \end{bmatrix} \end{aligned}$$

9 Computational properties of matrices

This should be obvious from multiplying out:

$$\begin{pmatrix} . & . & . \\ . & . & . \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} . & . & 0 \\ . & . & 0 \\ . & . & 1 \end{pmatrix}$$

Essentially, the last column of the second matrix "selects out" the last column of the first matrix as the last column of the product, and this is $\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$.

Basically, all the 2D transformations we've seen—translation, rotation, scaling, reflection—have this characteristic. This means that to represent such transformations (or any product of such transformations) as homogeneous matrices, we need only store and compute 6 numbers rather than 9.

10 Convex hulls and Bézie curves and 20 joining Bézier curves

Additional part to question: ray tracing curved surfaces. Assume you're doing ray-tracing. Compare Bézier surface patches and quadric surfaces in regard to the computational cost of computing intersections of rays with the surfaces, and cost of computing surface normals at the intersection point. Take into account the use of bounding volumes to speed intersection tests. What about for rendering methods other than ray-tracing?

These questions are all inter-related. First, about convexity: For a Bézier curve segment we can use this convex-hull property to speed intersection tests, say with straight line segments (or other curves). If the line segment doesn't intersect the convex hull of the control points, then we know it can't possibly intersect the curve segment. This is generally a win, because computing the intersection of a line with a Bézier cubic parametric curve requires solving cubic equations, which is doable but non-trivial. This convex-hull property (in 3D) also holds for Bézier surface patches in 3D. If you think about the surface patch $Q(s,t)$ as being a family of curves in t, one for each value of s, then these t-curves sweep out the surface as s changes. Any point on a t-curve is a convex linear combination of that curve's control points. These control points themselves move along curves defined by the control points in the other direction, and are therefore convex linear combinations of these other control points. This means that any point on the surface is ultimately a convex linear combination of the 16 control points, and therefore must lie within their convex hull. This property can be useful for much the same reason, intersection tests, like intersection with a line of sight for ray-tracing.

For two successive Bézier curve segments, to get C^1 continuity, first have to have C, and to achieve this, you have to have the end point of the first curve segment be identical to the starting point of the second curve segment. This is already the case in this formulation, since the control point P_4 is common to both curve segments. Beyond this, to obtain C^1 continuity, the tangent vector at the end of the first segment has to be the same as the tangent vector at the start of the second segment. This is achieved by having the vector P_3P_4 be the same as the vector P_4P_5. In other words, P_3, P_4, P_5 must all be collinear, with P_4 midway between P_3 and P_5.

For intersecting a ray with a surface: For a Bézier cubic surface, this requires solving a cubic equation. There is a (more or less) a closed-form solution for this, but it's relatively expensive. For a quadric surface, it's just a matter of solving a quadratic equation, which is much simpler.

Computing the normal at a point (s,t) on a Bézier surface patch requires computing the tangent vectors to the surface in each of the s and t directions, and then computing their vector cross product (since the normal must be perpendicular to both tangent vectors). This involves evaluating six quadratic expressions (for the three components of the two tangents) plus the cross product. For a quadric surface, the normal is essentially the gradient of the quadratic form, whose three components are just linear expressions, and so much easier to compute.

Note, this doesn't take into account shortcuts from use of bounding volumes, such as convex hulls or bounding boxes, but I expect the speed up from these would be much the same for both cases. Note also, that quite often for rendering a Bézier patch will be "diced" up into quadrilaterals (by stepping through s and t in sufficiently small steps). Then this mesh of quadrilaterals can be rendered by techniques similar to those used for polygonal meshes. (The quadrilaterals are not necessarily planar and so won't be polygons, but would usually be pretty close.) Or the quadrilaterals can be subdivided into triangles (by running an arbitrary fake diagonal edge across a quadrilateral) and then it can be rendered as a triangular mesh (a special case of a polygonal mesh), often with some sort of interpolated shading. To ray-trace a diced Bézier patch, we have to compute intersections with many triangular (quadrilateral) pieces.

Sometimes dicing is done even for quadric surfaces. Because quadric surfaces are easier to handle, this probably isn't worthwhile on stock hardware, but may be an advantage say if you have special graphics hardware for rendering triangles.

12 Piece-wise curves

So long as you have some machinery and conventions for finding a seed point inside the enclosed region, simplest would be just to draw the entire boundary (assuming you have code for tracing the curves), and then flood-fill the inside. (You'd probably also need to assume that the boundary didn't cross itself.)

Some kind of scan-line approach would be possible, but much more involved than in the polygon case. For any scan-line, you could compute intersections with all the curve pieces and sort the intersections by their x coordinate. But doing this would require being able to solve the equations for all the intersections of the scan line with each curve piece. This would be fairly easy and reliable if the curve pieces were quadratics (conic sections) but harder if they were higher-order curves.

The sorting of pieces for orderly progression wouldn't work directly, because a curve piece could wiggle up and down arbitrarily between its endpoints. However, it would be possible to compute (as additional information) the y extremes of each curve piece, and use these instead. (Obviously, the endpoints and extremes are the same for straight line segments.)

Also, if the scan lines progress row by row down the image then the intersections on one row would normally differ only a little from those on the previous row. So they could profitably be computed by using the previous intersections as starting approximations in some iterative root-finding procedure, rather than finding the roots from scratch on each row. (For circular arcs, the circle version of Bresenham's method could be used.)

13 Rotation by shear

Note: the following solution assumes the following matrix form:

$$\begin{aligned}
\mathbf{R}_\theta &= \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} \\
&= \mathbf{H}_a \mathbf{V}_b \mathbf{H}_a \\
&= \begin{pmatrix} 1 & 0 \\ a & 1 \end{pmatrix} \begin{pmatrix} 1 & b \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ a & 1 \end{pmatrix}
\end{aligned}$$

(According to the convention used in lectures, this should really be:

$$\begin{aligned}
\mathbf{R}_\theta &= \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} \\
&= \mathbf{H}_a \mathbf{V}_b \mathbf{H}_a \\
&= \begin{pmatrix} 1 & a \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ b & 1 \end{pmatrix} \begin{pmatrix} 1 & a \\ 0 & 1 \end{pmatrix}
\end{aligned}$$

)

Solution:

$$\begin{aligned}
\mathbf{H}_a \mathbf{V}_b &= \begin{pmatrix} 1 & 0 \\ a & 1 \end{pmatrix} \begin{pmatrix} 1 & b \\ 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} 1 & b \\ a & 1+ab \end{pmatrix}
\end{aligned}$$

Therefore

$$\begin{aligned}
\mathbf{H}_a \mathbf{V}_b \mathbf{H}_a &= \begin{pmatrix} 1 & b \\ a & 1+ab \end{pmatrix} \begin{pmatrix} 1 & 0 \\ a & 1 \end{pmatrix} \\
&= \begin{pmatrix} 1+ab & b \\ 2a+a^2b & 1+ab \end{pmatrix}
\end{aligned}$$

Equating this element by element with $\mathbf{R}_\theta$, we'd have to have:

$$\begin{aligned}
b &= \sin\theta, \qquad \text{and} \\
1+ab &= \cos\theta, \qquad \text{therefore} \\
a &= \frac{\cos\theta - 1}{\sin\theta} = \frac{-\sin\theta}{1+\cos\theta} \\
&= -\tan\frac{\theta}{2}
\end{aligned}$$

If we have this, then

$$\begin{aligned}
2a + a^2 b &= \frac{2(\cos\theta - 1)}{\sin\theta} + \frac{(\cos\theta-1)^2 \sin\theta}{\sin^2\theta} \\
&= \frac{2\cos\theta - 2 + \cos^2\theta - 2\cos\theta + 1}{\sin\theta} \\
&= \frac{\cos^2\theta - 1}{\sin\theta} = \frac{-\sin^2\theta}{\sin\theta} = -\sin\theta
\end{aligned}$$

which confirms that it all comes out right.

Calculating a rotation by direct multiplication by a rotation matrix (once you know $\sin\theta$ and $\cos\theta$) takes four multiplies and two additions. A single shear takes just one multiply and one addition, so three shears take three multiplies and three additions to do the rotation. Since on most machines, addition is faster than multiplication, this gives a slight edge to rotation by shears. This analysis ignores the cost of data movement, but that should be fairly small and about the same for both methods—besides, it depends a lot on the instruction set of the target machine. However, for most machines, the triple-shear method would also have a slight edge in data movement and register usage.

We'll see later how this triple-shear method is a bigger winner for rotation of digital images.

14 Computational efficiency of homogeneous coordinates

Hint: consider the number of multiplications and additions required. This is discussed in the Foley text in Section 5.6.

15 Perspective in homogeneous coordinates

If we think of perspective projection as a mapping from 3D to 2D, then it can be expressed as a multiplication (on the right) by a 4×3 matrix (that is, 4 rows, 3 columns).

We want the homogeneous point $(X, Y, Z, 1)$ in 3D to "go to" the homogeneous point $(fX/Z, fY/Z, 1)$ in 2D—this comes straight from the equations of perspective projection. This latter point is the same as $(X, Y, Z/f)$, since multiplication throughout by the constant Z/f gives the same point in homogeneous coordinates, and

$$\begin{pmatrix} X & Y & Z & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1/f \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} X & Y & Z/f \end{pmatrix}$$

Perspective projection can also be thought of as a mapping from 3D to 3D, given by the 4×4 matrix:

$$\begin{pmatrix} X & Y & Z & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & * & 0 \\ 0 & 1 & * & 0 \\ 0 & 0 & * & 1/f \\ 0 & 0 & * & 0 \end{pmatrix} = \begin{pmatrix} X & Y & * & Z/f \end{pmatrix}$$

where the starred matrix column can be filled in in various ways to provide a Z coordinate for the result. One way is to just map onto the plane $Z = f$, another (popular in 3D graphics) computes a so-called *pseudo-depth*, which, while not the same as true depth, is a monotonic function of depth, and therefore can be used conveniently for ordering points in hidden-surface elimination, and similar processes. Pseudo-depth can also make perspective projection into an invertible affine transformation.

Treatments of perspective projection in most graphics textbooks tend to have the viewing/image surface pass through the origin, and have the centre of projection away from the origin. This gives slightly more complicated equations than our setup. Conversion from one to the other requires a 3D translation. This, and the use of pseudo-depth, explains the slightly different formulation of perspective projection as a matrix operation seen in many graphics texts. But the underlying idea is the same.

16 Perspective transformations using matrices

Here we have a slightly constrained version of the fully general "synthetic camera": in this case the camera can move anywhere, but its orientation is determined by its 3D position.

The main problem is to transform from world coordinates into camera coordinates—once there, the perspective projection (or orthographic projection) is quite simple. This coordinate-change transformation is the same transformation that (applied to the camera) brings the camera coordinate frame to match the world coordinate frame.

To take a simple 2D example, if the camera were at coordinates (U, V) in the plane, then the transformation that would convert from world coordinates to camera coordinates is a translation by $(-U, -V)$. This is the same translation that would bring the camera to the origin.

But first we need to specify what the pose (position and orientation) of the camera is. Obviously, the position of the camera (optical centre) is at (U, V, W). The camera is pointing towards the origin, so the direction of the camera-Z axis is along the direction of the vector $\begin{pmatrix} U & V & W \end{pmatrix}$. (Remember that, with our conventions, the positive direction of the camera-Z axis is *backwards* from the direction the camera is looking.) This direction can conveniently be expressed as a unit vector $\begin{pmatrix} u & v & w \end{pmatrix}$ where

$$u = \frac{U}{\sqrt{U^2 + V^2 + W^2}}, v = \frac{V}{\sqrt{U^2 + V^2 + W^2}}, w = \frac{W}{\sqrt{U^2 + V^2 + W^2}}$$

However, this gives only the pitch and yaw of the camera (or pan and tilt)—it does not give the camera roll. To do this, we need to specify one of the other camera axes, camera-X or camera-Y. Let's take camera-Y (which is the same direction as the y axis of the image surface coordinates). This points towards the positive world-Z axis. There are probably more succinct ways of deriving this, but here's a way that's fairly direct and intuitive. Directions are invariant to scale, so we can imagine scaling the whole set-up down so the camera actually is positioned at $\begin{pmatrix} u & v & w \end{pmatrix}$. Camera-$X$ and in particular camera-Y must lie in the plane

$$uX + vY + wZ = 1$$

which passes through the camera centre, normal to camera-Z. This plane intersects the world-Z axis $(X = Y = 0)$ at the point $(0, 0, 1/w)$. The vector from the camera centre to this point, by vector subtraction, is $(-u, -v, (1/w) - w)$, or $\left(-u, -v, (w - 1)/w^2\right)$. This vector defines the camera-Y axis.

Having established the camera pose, we can now proceed to the transformation that brings the camera frame to the world frame. First we have to translate by $\begin{pmatrix} -U & -V & -W \end{pmatrix}$, to move the camera to the world origin. Call this translation T.

Next we have to rotate to line up the camera-Z axis with the world-Z axis. For example, this could be achieved by a rotation about the world-X axis to bring the camera-Z axis into the world XOZ plane, followed by a rotation about the world-Y axis (within the world XOZ plane) to bring camera-Z into exact alignment with world-Z.

Alternatively we could have rotated first about world-Y and then about world-X.

We have to remember this composite rotation, call it R_1.

We have to apply R_1 to the camera-Y axis—this tells us where camera-Y has ended up after all this. The final thing to do is to rotate around world-Z to bring camera-Y into alignment with world-Y. Call this rotation R_2.

So the composition of all these three TR_1R_2 is the transformation that will bring the camera to line up with the world coordinate frame. It is therefore also the transformation that will convert world coordinates into camera-cantered coordinates. Once a world point (X, Y, Z) is converted into camera-centred coordinates, the simple projection equations can be applied to obtain image x and y.

I haven't provided the details of these transformations, but note that that u, v, and w are actually the cosines of the angles the unit vector $\begin{pmatrix} u & v & w \end{pmatrix}$ makes with the respective coordinate axes. (They are, after all, also known as "direction cosines".) So the sines and cosines needed for the rotation matrices for R_1 and R_2 can be computed algebraically from u, v and w—there is no need to invoke trigonometric functions.

This derivation has behind it the mental picture of an object sitting on a table, with world plane XOY being the table top, world-Z upwards, the origin somewhere under the object, and the camera looking down on the object from somewhere above the table. The camera-orientation constraints get problematic if the camera is allowed to go below the table. (This might make sense if the table top were glass and we wanted to get an underside view of the object.) Under the table, you just can't get positive image-y to point to positive world-Z. The derivation given here would force the camera to flip over as it went below the table, so that positive image-y would point towards *negative* world-Z.

This is not unrelated to an anecdote about the automatic flight-control systems for a certain American jet fighter. The first time it was flown south of the equator, the plane flipped over and tried to fly upside-down. There is similar geometry involved.

Alternatively, the camera could keep the same general sense of "up" as it sank below the table top, so that instead negative image-y changed over to point to negative world-Z. But this would require a slightly different characterisation of camera-Y.

17 Multiple light sources

Since the light source is "distant", all points on a face will be effectively at the same orientation with respect to the light-source direction. This means that we'll get the same light intensity reflected to the viewer from all the points on the face. We can store in our frame buffer some kind of index number for the face each pixel belongs to. We can then compute the intensity of that face, depending on the intensity and direction of the light source, the intensity of ambient light, and the albedo (reflectivity) of each face, and store that into the lookup table entry for that face's index. This ensures that each pixel will be displayed with the correct intensity.

If we change the light-source direction (or intensity for that matter) all we have to do is recompute the lookup-table entries for each face index, and the image pixels will be displayed with the correct, new intensities. This could be done quite quickly, say in response to mouse input, mapping mouse positions on screen to directions in space.

For the Lambertian case, there's no problem with multiple (distant) light sources—it's just a matter of adding up their individual contributions. The image intensity would still be constant across each face, so the same trick of storing a face index in the lookup table would still work.

Specular reflection (like a mirror) also depends on the viewer direction. For a nearby viewing position, the viewer direction would vary across the face, and this technique wouldn't work. It would work fine, however, for a distant viewer (or for parallel projection, which amounts to about the same thing). Remember, though, that that the specular reflection (highlight) would be the same all the way across the face, and this might not look very realistic for an object composed of just a few large faces. (It does happen, say when the late sun catches the side of one of those mirror-faced skyscrapers.) It would probably look more realistic for an object made up of many small polygonal faces approximating a curved surface. Then the highlight would just come off one small patch (face or few adjacent faces).

If there is more than one object, or if the object is non-convex, then for certain light-source directions parts of the object may be in shadow (either cast by other objects, or other parts of the same object). The shadowing calculations are quite complex (roughly the same as hidden-surface elimination), and most important for us, will vary across each face. So our lookup-table approach couldn't be made to work.

18 Difference between Phong and Gouraud

Since Phong handles specular reflection (highlights) better than Gouraud, a good guess would be an example involving such. Take the extreme case (what better!) of a perfect mirror face, and a point light source. Take it that the normals at the face vertices are the same as the face normal.

On viewing such a face, what you'd see (assuming things are lined up right) would be a mirror image of the point light source located somewhere inside the face. With no ambient lighting, you see just a single point of light—everything else would be black. In particular, the vertices of the face would be black, so any scheme, like Gouraud's, that interpolates between vertices would colour the face all black.

Phong shading would interpolate the normal direction across the face (in this case constant) and catch the point image of the light source (so long as it was cooperative enough to fall exactly on a pixel).

More reasonably, the face would not be so perfectly specular, so the point source would give an extended highlight. But the same reasoning applies: along a scan line, Phong shading would start out at zero, rise to a maximum passing through the highlight, then fall back to zero. Gouraud shading would linearly stay at zero all the way along.

19 Shading highlights

You can use Gouraud shading with specular reflection (instead of Lambertian reflection) to create lighting highlights on object surfaces by simply calculating specular properties at vertices and interpolating.

In terms of comparison of Phong shading, as a function of number of polygons and amount of computation required,

- you would require more polygons using Gouraud than Phong because don't interpolate surface normal.
- you would require more computation in Phong (to calculate shading equation at each interpolated point) but would require less polygons and visa-versa.

20 Computational cost of shading

- Phong shading is also known as normal-vector interpolation shading,
- as you interpolate the surface normal vector $\bar{N}$, rather than the intensity.
- At each pixel a new intensity calculation is performed using the illumination model (such as the Phong illumination model).
- The computational cost at each pixel includes,
  - re-calculation of the illumination model at each pixel, and
  - the interpolated surface normal must be normalised.
- DUFF79 : combination of difference equations and table lookup can be used to speed up these calculations.
- Bishop and Werner: Approximation using Taylor series expansion to get greater speedup.

21 Intensity and RGB colour

Normalised colour can be very useful, within limits, because to a large extent it does cancel out the incidental effects of illumination and shading. But it does throw some of the baby out with the bath water, in that the intrinsic *albedo* of surfaces is also lost. For example, all grays, from (almost) black through dark gray through light gray to white, which are characterised by $R = G = B$, are normalised to $r = g = b = 1/3$.

And saying "almost" black gives a hint of the other problem: Normalised colour is obviously undefined for "perfect" black, $R = G = B = 0$, but also runs into severe error problems in any dark area, characterised by low values of R, G, and B. Intensity quantisation (and added noise if any) introduce a certain absolute error into the RGB values. For small RGB values, this absolute error can be a quite large *relative* error, which is what counts for doing division (as in computing normalised colour). As an extreme example, RGB value $(1, 1, 0)$ normalises as $(1/2, 1/2, 0)$, the same as "pure" yellow, while only a one-step change (which could easily come about from an accident of quantisation or noise) could give $(1, 0, 0)$, which normalises as $(1, 0, 0)$, the same as "pure" red. This means that normalised-colour values can be extremely unreliable in dark areas of an image, and that normalised colour values computed in bright areas of an image cannot be computed at all in the darker areas.

This is a general problem that shows up whenever we're computing ratios of quantised values, like the ratio between two images, or edge directions from gradient components. If the values are small (just a few quantisation steps), the large relative errors can make the ratios almost meaningless.

22 Different shading approaches

No, the suggested shortcut doesn't really work, and is yet another example of the old adage "you can't average averages". Consider the case of incoming light that is pure red (say $P_R = 1$, $P_G = 0$, $P_B = 0$, where these are the respective RGB light-source components) and an object that reflects only blue light (say $r_R = r_G = 0$, $r_B = 1$, where these are the respective RGB reflectivities). Then the object will reflect none of the incoming red light ($r_R = 0$), and won't get any blue light that it could reflect $P_B = 0$. So overall, the object will reflect no light in any RGB component, and therefore will appear black.

Now if we add together the incoming light to get $P = 1$, and average together the reflectivities to get $r = 1/3$, then we'll get a monochrome shade of 1/3 (on a scale of black 0, white 1). Wrong answer!

The situations in which this will work are when the objects are all intrinsically gray (having equal reflectivities in RGB), or when the incoming light is white (all RGB components equal). In both cases, there is a constant that can meaningfully be factored out of the calculations. The first situation is not very interesting—an all-gray world—, but the second corresponds closely to a common situation of coloured objects illuminated by (presumably white) natural light. Not a perfect fit, since natural light may not be perfectly white, but pretty close for most purposes.

Really even the "full" RGB approach is only a convenient approximation. Light has a continuous distribution across all possible wavelengths, and object surfaces can have different reflectivities at all different wavelengths. In a sense, the RGB approach is a kind of projection of this infinite-dimensional wavelength space onto just a three-dimensional RGB colour space. Since our human colour perception is essentially three-dimensional, this works pretty well for most reasonable situations. However, there are unusual cases in which it can break down. Imagine looking at an object that reflects spectral red and green (but not yellow) under a pure spectral yellow sodium lamp. The object should appear black. But if we treat the incoming light as a yellow-looking mixture of red R and green G components, then the object will appear yellow.

23 Refraction

If we label the angles as in the diagram, Figure 1, then obviously $\theta_2 = \theta_3$ because of all the parallels. Taking the refractive index of the plate (glass) as n_2, and the refractive index of the surrounding medium (air) as n_1 (which would be 1.0 for air), two applications of Snell's Law (one at each interface) give us:

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{n_2}{n_1}$$

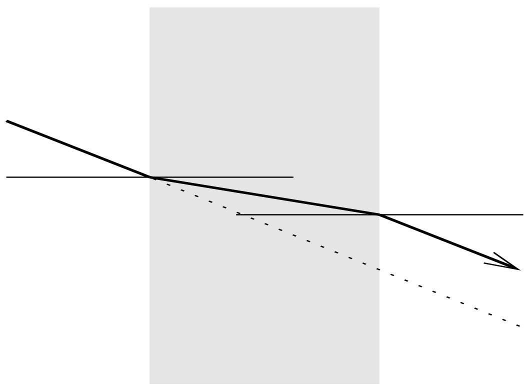
Figure 1: Double refraction through a glass plate.

and

$$\frac{\sin\theta_4}{\sin\theta_3} = \frac{\sin\theta_4}{\sin\theta_2} = \frac{n_2}{n_1}$$

From this, it's obvious that $\sin\theta_4 = \sin\theta_1$, so $\theta_4 = \theta_1$. Since both the incoming and outgoing rays make the same angle with the (parallel) surfaces, they must be parallel. Looked at another way, it's obvious from the symmetry of the situation.

Suppose the plate had thickness d, then the path of the light through the plate has length $x = d/(\cos\theta_2)$. The angle between this path and the original path of the incoming ray is $\theta_1 - \theta_2$. This angle forms a right triangle of which the hypotenuse is length x, and the offset we seek Δ is the opposite side. Therefore

$$\Delta = x\sin(\theta_1 - \theta_2) = d\frac{\sin(\theta_1 - \theta_2)}{\cos\theta_2}$$

Ultimately, this should be reduceable to a formula just in terms of θ_1, n_1 and n_2 (actually just their ratio), but I haven't done this yet.

Remember that the sines and cosines that appear in many formulas are actually ratios than can be computed from the direction components of suitable vectors. (That's why the components of a unit vector are called *direction cosines*—that's what they are.) So it's usually not necessary to deal with explicit angles and call trig functions to get the ratios, the needed ratios are already there in the geometry.

24 Hidden surface removal

Linear interpolation of depth means we can use an adaptation of Bresenham's algorithm to do the interpolation quickly using only integer arithmetic. Think of a polygon that slants away from the viewer in depth. A step in x at the near end will cover a small step in depth; the same-size step in x at the far end will cover a bigger step in depth. The greater the slant, the worse the effect. So linear interpolation doesn't really work right. If we're happy to recompute respective polygon depths at the intersection points, and dispense with interpolation, then it doesn't matter. If we used interpolation we'd get (slightly) wrong answers. The advantage of pseudo-depth is that it preserves the ordering of true depth, and (being obtained by a linear transformation) preserves linearity. So linear relationships (like planarity) are preserved even after perspective transformation into x, y, and pseudo-depth (in homogeneous coordinates). This is a nice property that has its uses, one of them being that it is possible to do linear interpolation of pseudo-depth after perspective transformation, when x and y are in image coordinates. Since pseudo-depth preserves the ordering of true depth, hidden surfaces work the same. Pseudo-depth has a couple of other nice properties: it is bounded, so we can handle infinite depth conveniently. And (related to this) for a quantised pseudo-depth, a greater part of its range is devoted to nearby objects than to far away objects, in contrast to what we'd get with a uniform quantisation or true depth. In a sense, it requantizes depth to give finer quantisation where it most counts.

25 Animation and kinematics

For this question refer to lecture slides on animation and kinematics in the lectures slides. The theme of the question relies on the balance between forward kinematics (when you have at least as many equations as unknowns) and inverse kinematics (when you have more unknowns than equations).

One of the downside of forward kinematics, however, is that you typically need to specify the parameters of transformations ahead of time, whereas in inverse kinematics less needs to be specified. Of course, this is overlaid with the problem of how many joints each limb or component being animated has.