

Some Details on Matlab Assignment

Chris Brown

30 Nov 2008, Edits 23 Nov 2010

Here I pretty much work straight through the concepts in their order of appearance in the assignment. You should read the assignment too... This is a first draft to please tell me about mistakes, confusions, etc. (I know the Figs are not in order).

1 Coordinates

Fig. A shows a familiar 2-D coordinate system. Typical 3-D axis labels are as in Fig. B: useful for robots and visualizing functions of X and Y as height. For cameras, it's customary (with me anyway) to think of Z as the direction of gaze, out along the camera's central line of sight, so let's use labels as in Fig. C.

A special coordinate is nailed down and constant throughout our calculations: luckily for us we can do this exercise using only coordinates with respect to that coordinate system, which we call LAB.

To keep signs straight, we shall use *right-handed coordinates*. If you stick your right thumb in the direction of the Z coordinate: for a rotation around this axis by a positive angle, the fingers curl in the direction that the rotation moves points on the X axis (i.e. towards Y.) Similarly (positive) rotation around Y rotates points on the Z axis towards X and around X carries points on Y toward Z. Note: for this exercise, the axes do not move, only points move!

Our project is to implement the situation of Fig. D, with the (pinhole, if you will) camera modeled as *point projection*. 3-D scene points a and b being projected onto the image through the *viewpoint*, or camera's *focal point* P, to yield 2-D image points p and q. One can leave the focal point on the Z

axis, as in Fig. E, but I like the gaze direction to yield an image point in the center of the image, so I make a little adjustment in the image points (subtracting half the image size from each one's X and Y coordinates) to accomplish that, as in Fig. F.

Anyway, our camera model is just going to be a set of rigidly-connected 3-D points (they all move together coherently), which we'll move around in space to implement camera motion. Then we need to model the picture-taking process and the properties of light interacting with surfaces, and we're done.

2 Movement I: Translation

Generally we'll think of vectors as *column vectors*, usually in 3 dimensions. Columns are hard to write, so without warning I can also use row notation, as in (u, v, w) , or I might note that their column-hood is important by using a Matlab-Transpose operator as in $(u, v, w)'$. Often, however, the ' is just a prime on a single vector variable, as in $x' = Rx$, meaning x has been transformed to a new x' . Worse, I'm terrible about using notation like x , x , and even P all to mean 3-d column vectors. I hope context disambiguates it all.

Fig. G shows a point at coordinates (p, q, r) in LAB (the fixed 'laboratory' coordinate frame). It defines a vector, or really IS a vector whose elements are (p, q, r) as measured from LAB's origin, as shown. v is a vector with elements (a, b, c) : you see it over at the origin, but then also moved to illustrate the familiar vector addition operation, whereby we derive a vector-sum point $(p+a, q+b, r+c)$. Thus we model the translation of points like $(p, q, r)'$ by vector addition with a *translation vector* like $(a, b, c)'$: $(p, q, r)' + (a, b, c)' = (p + a, q + b, r + c)'$. All coordinates in this treatment and in the assignment are expressed in LAB. (This is dandy: we don't have to express points in any other coordinate system).

So: to model *camera translation* (in movie-speak, that's *tracking*), we're going to add a translation vector to every point in our camera model.

3 Directions

Directions as Unit Vectors. A 3-D direction is described by a 3-vector: first imagine its base at the origin of LAB with its head at coordinates (a,b,c). Its head is at a single point but clearly the vector defines a direction: the one you head in as you traverse from its tail to its head.

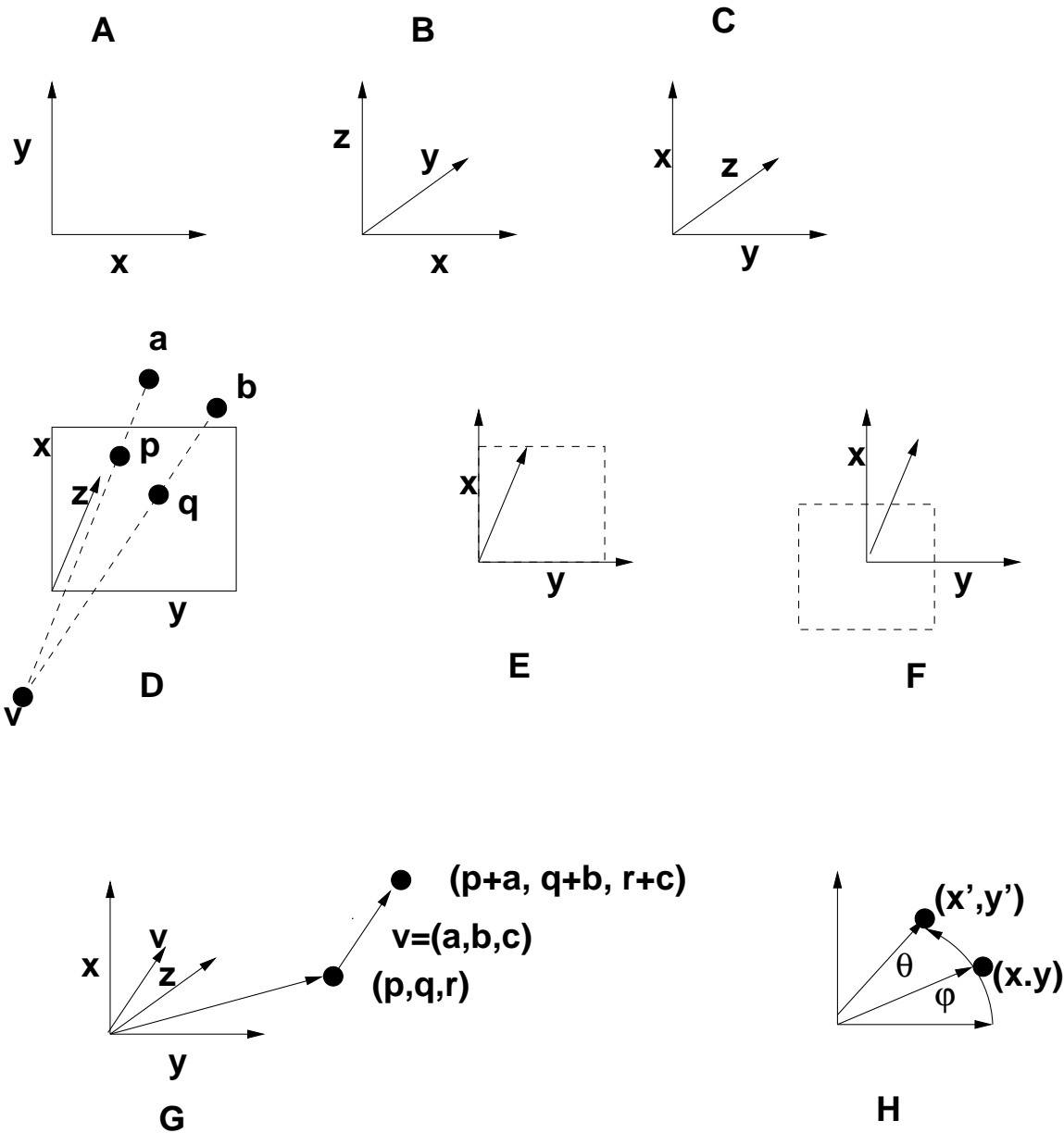
Let's always make *direction* vectors *unit vectors*, with unit length. If we need to convert a vector to a unit vector, we *normalize* it: divide each of its elements by the square root of the sum of its squared elements. If you think about it, its elements are then the projections of the unit direction vector onto the X,Y,Z axes (here, LAB's X,Y,Z axes). These projections are sometimes called *direction cosines*: if D is a direction vector and X is a unit direction vector along an axis, then $D \cdot X$, the dot product, projects D onto X , and the length of this projection is $|D| |X| \cos(\theta)$, so is equal to the cosine of the angle between D and X .

Now we can see Fig. G in a new light: we can also read it that v is also a direction vector, and that it has been translated by (p,q,r). The thing to notice is that it still points in the same direction: translation leaves it invariant. Thus for a vector anywhere in space defined by the location of its tail and head points, we can figure out which direction it represents by moving its tail to the origin (by subtracting the location of its tail from that of its head) and normalizing.

Light source direction. A *point source at infinity* is a very convenient mathematical abstraction. The idea is that there is a very small light source (like a star as seen from earth) that is very far away (ditto). In the ideal case the light comes from only one direction throughout all 3-D space. First, there's only one point generating the light. Second, you can imagine the point in the room with you where its light shines out in all directions. Now imagine it running off into the sky: the light rays passing through the room come from a decreasing set of angles, getting more and more parallel. With the source at infinity, all its rays are parallel: from the same direction. What direction? The direction of (pointing to) the light source. Or: if two observers anywhere in the universe saw different directions to the source, they could project those directions into space, and they'd have to meet at the source, somewhere in 3-D space. But the source isn't in space, it's off at infinity, so the observers can't see different directions.

You can easily model another "lighting effect:" *ambient light* is light from all directions, and its effects (for the reflectance functions we consider here,

at least) are modeled by simply declaring a constant minimum brightness, or by adding a constant value to all computed brightnesses to account for the ambient light.



4 Movement II: Rotation

Cameras *pan* (rotate left and right, as in shaking your head “no”), and *tilt* (rotate up and down, as in nodding “yes” — in the USA anyway). They don’t usually twist along the gaze direction (the psychedelic movies of the 70’s had some exceptions).

In 2-D, consider the situation in Fig. H: A unit vector (x,y) is rotated by an angle θ : what are its new coordinates? Well, originally it was at some angle ψ , and that means its coordinates were $x = \cos(\psi)$, $y = \sin(\psi)$. After rotation by θ , its coordinates clearly have to be $x' = \cos(\psi + \theta)$, $y' = \sin(\psi + \theta)$. By the trig identities for the cosine and sine of a sum of angles,

$$\begin{aligned}x' &= \cos(\psi + \theta) = (\cos \theta \cos \psi - \sin \theta \sin \psi), \\y' &= \sin(\psi + \theta) = (\sin \theta \cos \psi + \cos \theta \sin \psi).\end{aligned}$$

We can express the rotation transformation by a multiplication by a *rotation matrix* (they have lots of nice properties we won’t get into). Let’s abbreviate cosine as c and sine as s . Recall $(x, y) = (\cos \psi, \sin \psi)$, and you’ll see that the above equations for x', y' are given also by this matrix formulation:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} c\theta & -s\theta \\ s\theta & c\theta \end{bmatrix} \begin{bmatrix} c\psi \\ s\psi \end{bmatrix} = \begin{bmatrix} c\theta & -s\theta \\ s\theta & c\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Now in 3-D things turn out to look similar. Remember how to interpret rotations in a right-handed system. Clearly rotating around X doesn’t change the X coordinate, ditto Y and Z. If you think of the 2-D case as being rotation around Z, you sort of get the idea that all the rotation matrices should look similar, which is almost true. So respectively RotX to rotate (points on) the Y axis toward Z and RotY to rotate (points on) the Z axis toward X are:

$$RotX(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c\theta & -s\theta \\ 0 & s\theta & c\theta \end{bmatrix}$$

$$RotY(\theta) = \begin{bmatrix} c\theta & 0 & s\theta \\ 0 & 1 & 0 \\ -s\theta & 0 & c\theta \end{bmatrix}$$

Please Note the Sign Change in RotY!

If you want to twist your camera, RotZ rotates the X axis toward Y,

$$RotZ(\theta) = \begin{bmatrix} c\theta & -s\theta & 0 \\ s\theta & c\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

So to rotate a point (column 3-vector) v , you just make the matrix product $v' = Rv$, where R is a 3×3 rotation matrix. The resulting column vector v' is your rotated point.

Matrix multiplication is associative but not commutative. So rotating around X and then around Y is not the same as doing the same rotations in the other order.

For an example, let A, B, C, D be any four 3-D rotation matrices, each about any axis, and let v be some 3-D point. Consider the multiplication $v' = ABCDv$: v' is the point v rotated in LAB by D , and then the resulting point rotated in LAB by C , and then in LAB by B , and then in LAB by A . Note that $ABCD$ or generally any product of rotation matrices is a rotation matrix, so if we compute $E = ABCD$ we have in E the effect of doing all the component rotations in right-to-left order, all packed into a single 3×3 matrix.

If you want to think about “doing one thing at a time”, then you get to the same place. First you decide to rotate with D , and then later you want to rotate the resulting point by C , etc. Gives the same result but you never really think about a product of rotations, you just think of applying rotations through time: $x' = Dx, x'' = C(x')$, etc.

There is another way to think about that $ABCD$ product that is useful for robotics, but I won't get into it here: the one above is natural and intuitive, since it's based on the idea of rotation actions moving points in one coordinate system defined by LAB.

5 Rotations about Other Centers

It's natural to think of rotating a camera about its own center, not LAB's origin. But if you take a translated camera and rotate it, our mathematics means you swing it around LAB's origin, which is hard to think about and probably not what you want (Fig. K).

SO: to rotate a camera about its own focal point P: subtract P (i.e. put the camera at the origin of LAB), rotate, add P (stick P back where it came

from, with all the rest of the camera being rotated around that point).

6 Pinhole Camera Model

There are lots of ways to model a camera, even as simple one as our pinhole camera (point projection) model. Often a pinhole camera is abstracted to be a 4×4 matrix, thought of as containing a rotation and a translation and numbers to define its focal length. We're going to use all those concepts but keep them separate.

Fig. I shows the one-dimensional point-projection situation. The view-point, or *focal point* of the camera, is at P, which we treat here as the origin of coordinates. Its *focal length* is f . A point to be imaged is out at (x,z) . The image location x' of the scene point is defined as the intersection of the line from P to the point and the *image plane* (here an image line), which is located at $z = f$. If the point gets closer (z gets smaller) but stays the same "size" (x value), clearly its image (x') will get bigger, as shown with the $(x, z-d)$ point in Fig. I. So as we expect, closer things look bigger.

The smaller the focal length the bigger these size-variation effects, or *perspective distortion*. Wide-angle (small f) lenses aren't used for portraits because they make your nose look big. A telephoto lens has a longer focal length and minimizes perspective effects: in fact, if the focal length goes to infinity as in Fig. J, we have *orthographic projection* or parallel projection, in which the imaged x is the same as the scene x : the z component is simply 'projected away'.

Things in a 2-D camera work just like this, with the Y dimension treated similarly.

7 A Brute Force Camera Model

The camera model I have in mind for this assignment consists of:

1. a column 3-vector representing the location of P (the camera's focal point) in LAB.
2. a 2-D *image array* of *image points*, or pixels. Each pixel is just a location in space, or another column 3-vector. All the pixels lie in a

plane, the *image plane* (corresponding to the vertical 'image line' in Figs. I and J).

So what are these points? Think of them as the locations of individual CCD pixels on the sensor chip in the camera: they are in a 2-D grid and that grid is somewhere in space inside the camera. Thus they are actual 3-D points, not indices or anything abstract. I think of the units of their coordinates as locations measured in millimeters: the coordinates are (X,Y,Z) locations expressed in LAB. The size of a CCD image array is defined in millimeters (in CCD sensor arrays used in cameras, the advertized size is often the diagonal measurement of the array, as in TV screen specifications). I like to think of an initial situation with the camera at the origin of Lab ($P = (0, 0, 0)$). Then if you have pixels equally-spaced in X and Y, and know the focal length f , the 3-D location of pixel I(i,j) is (i*k, j*k, f), where k is the constant that says how many millimeters the pixels are separated by. The important thing is to remember the camera is "in the world", in the sense that all its points are represented in LAB.

To translate the camera, we translate all its points. To rotate it, we rotate all its points. For simplicity let's assume that the image array is $N \times N$ pixels (you can easily change this). Then there are N^2 pixels.

Remember expensive vs. efficient ways to represent state in state-space search? Here, the most obvious (to humans) way to represent the camera is a 2-D pixel array (pixels are 3-D objects, recall): represent the Image Array with an $N \times N$ array of 3-D pixels and to rotate and translate it you just use two for-loops and for every element of the array you do

```
ImageArray(i,j) = R * ImageArray(i,j);
```

```
ImageArray(i,j) = ImageArray(i,j) + Tran;
```

or

```
ImageArray(i,j) = (R * ImageArray(i,j)) + Tran;
```

OK, but there's a more efficient way: You could represent the $N^2 + 1$ points, including P, in a (basically) one-dimensional array indexed by pixel number from 1 to N^2 , with P appended as a special $N^2 + 1^{st}$ pixel). Actually it's a $3 \times N^2 + 1$ matrix: row 1 the X coordinate, row 2 the Y, row 3 the Z. You could make it by generating pixel locations in a left-right (column) within top-to bottom (row) scan, generating the x,y,z coordinates and putting each set in as a col. of the pixel array. Call this array Pixels. This representation has the nice property that you can just take a rotation matrix

R and rotate all the pixel-locations (and P's) at once. In Matlab:

```
RotatedCamera = R * Camera;
```

Of course remember that if you want to rotate around P, you must translate the camera so that P goes to the origin, then rotate, then translate back.

To translate this representation you need to add the translation column vector $(t_x, t_y, t_z)'$ to every column of Pixels, since every column represents a 3-D point. You could do that with a for loop, or you could vectorize it by burning space (here `Tran` is a column 3-vector of X,Y,Z translations, and `kron()` is Matlab's Kronecker Product function)

```
TranslatedCamera = Camera + kron(ones(1,N*N+1),Tran);
```

So then you'd have to be a little clever at figuring out, from the `i` in `Pixels(i,:)`, just what `j,k` pair in the image `I(j,k)` corresponds to `i`. (I hope this array index computation is familiar from data structures.)

Now you could of course use a 2-D array of pixels and a separate vector for P, but you'd then need a nested for-loop to do the camera rotation and translation.

8 Ray Casting

Ray casting is a basic graphics technique. The idea is rather like one of the earlier theories of vision (before you were born, maybe 1600 or so): your eye shoots out a (heck of a lot of) ray(s) and what happens to it (them) tells you "what's out there".

In a computer graphics context we make a computer-held model of a 3-D world, light sources, and camera, and for every pixel in the camera's image we figure out what part of the 3-D world (as illuminated by our light sources) contributes to what we "see" at that pixel location.

If you look at CB's home web page www.cs.rochester.edu/u/brown/home_orig.html You'll see a scene rendered by the POVRay raycaster, which I think is still available. *Rendering* is turning an internal 3-D representation that describes object shapes like cylinders, spheres, etc, and their reflectivity and light transmission properties to a 2-D image: basically what we're doing. Ray tracing made the crystal sphere turn the upside-down and backwards letters around; the "cities of Oz" are actually the same, but rotated, the clouds are really painted on a big sphere above, etc.

The idea behind ray tracing is: I look out into the world in some direction: what do I see? I need to figure out what the light coming back at me from that direction originates from and how it's modified. Now for the minimal version of our exercise it's simple; we can only see our sphere(s) out there, and there's only one simple light source, no mirrors, no lenses – we just need to figure how bright the sphere looks at any point (of course it may be painted). See Fig. L.

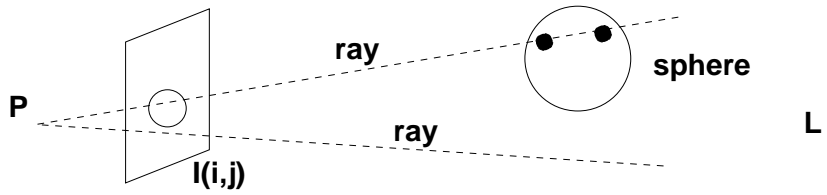
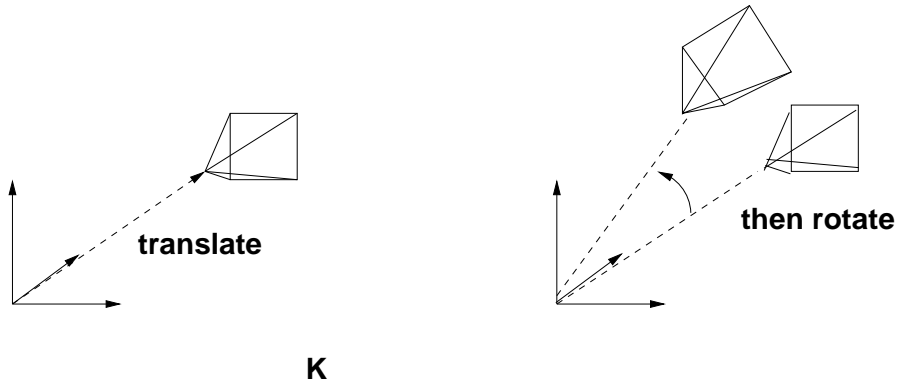
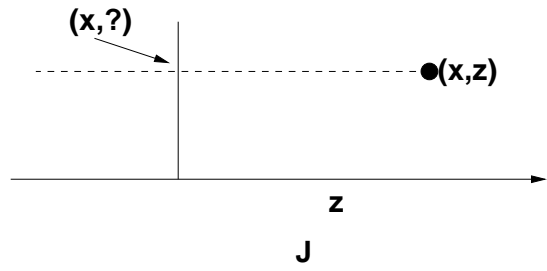
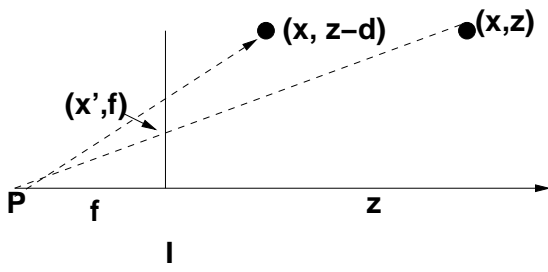
We need to know where on the image array any point on the sphere can project: or (easier), if a point on the image array represents a point on the sphere, what sphere point is it? Well, the line from the camera's P through a pixel I(j,k) (remember these are both 3-D points, which have been subjected to who-knows-what camera movements.) either intersects the sphere or it doesn't, and if it does it happens at a pair of (x,y,z) points. If you graze the sphere, the points are the same, if you miss the sphere the points are imaginary.

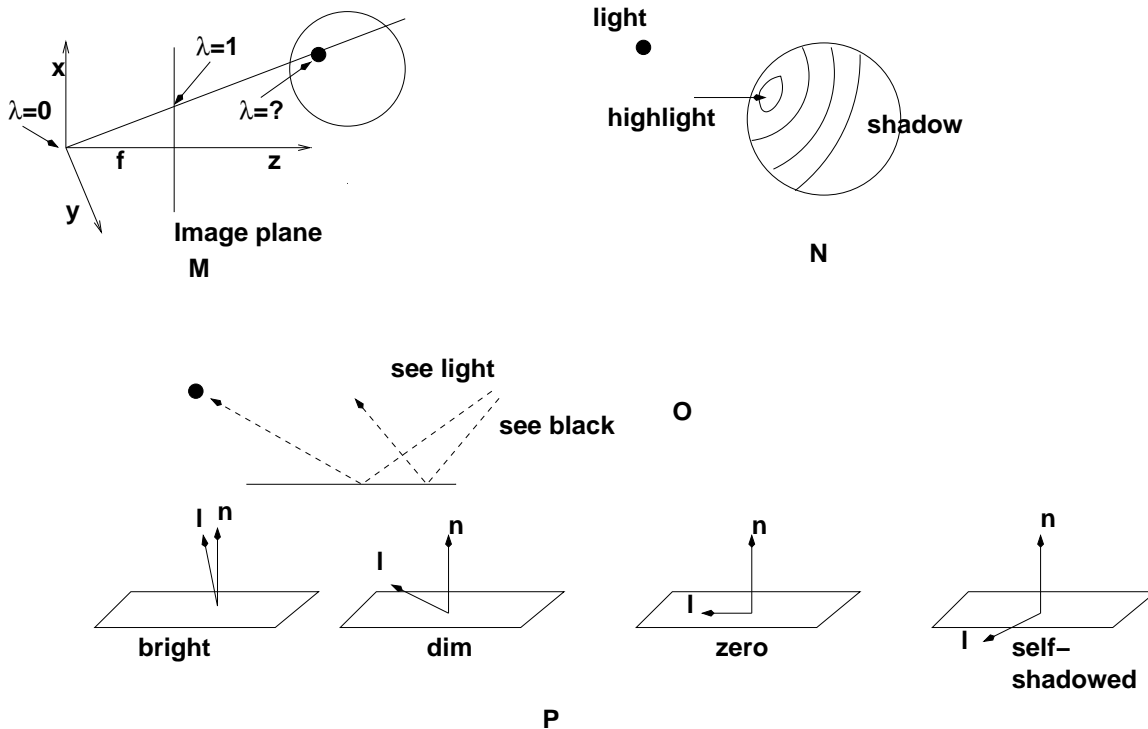
Figuring out the intersection (or not) point is a simple exercise in solving quadratic equations. The web page I found (in the assignment) tells (almost) all, or you may just want to do it yourself. One good way to approach the problem is to represent the 3-D ray (there are lots of ways to represent 3-D lines) by the set of all points $u = P + \lambda I(j, k)$, which in a really ugly notation means: if $\lambda = 0$, one point is P (on the line for sure), and if $\lambda = 1$ then that's the point at the pixel $I(i, j)$ in the image array. As λ gets larger the point continues moving out in that direction. Sooner or later it hits something (or doesn't) and so we know (with our simple model of solids out there in space) what world point causes the image at that pixel.

So the strategy is to find what λ makes the ray intersect with the scene, which for us is a sphere whose surface points are such that

$$(x - S_x)^2 + (y - S_y)^2 + (z - S_z)^2 = r^2$$

for a sphere with radius r and center $(S_x, S_y, S_z)'$. So set the equation describing the ray points equal to that describing the sphere points, solve for λ , use that to get the $(x, y, z)'$ points of intersection. It's pretty much all in the web page I point to in the assignment (Fig. M). As stated in the assignment, using higher-powered Matlab functionality at any point in this assignment is wrong.





9 Surface Reflectance

Given a viewpoint or focal point, a light source direction, and a surface (for us a sphere or maybe a plane), how bright does a particular pixel appear in the image?

Let's think about planes first, in particular mirrors and sheets of white paper. The ideal mirror is a *specular* surface and the ideal sheet of paper with no glossy component, or a “flat” white wall, is a *matte* surface. The ideal matte surface is *Lambertian* (Johann Heinrich Lambert's a physicist (1700's) with his name also on the Lambert Unit). If we look (from a point at infinity) into a mirror reflecting a point source at infinity, we see one of two things: in exactly one direction, the entire mirror reflects light from the source and looks uniformly bright. From any other direction it's dark (Fig. O). This is the “angle of incidence equals the angle of reflection” thing from Physics 101.

For a set of two interacting reasons, if you look from any direction at a lambertian plane illuminated by a point source at infinity, it *always looks*

the same brightness(!). Your viewing position does not affect its brightness at all. What *does* affect its brightness is the direction of the light source: e.g. illumination from directly overhead for a plane on the table (in general, from a direction normal to the surface) means it looks the brightest (from any viewing direction, remember). As the angle of illumination drops down to be parallel with the plane the perceived brightness also drops to zero.

(Optional) What’s going on is cute: think of the surface as “glowing” as a result of its illumination by the light source. Because a square foot of the surface gets less light (fewer photons) if it’s slanted with respect to the direction of the light, it glows brightest if it’s facing the source and the glow drops to zero as it tilts to be parallel with the light direction. So for any “lighting geometry” (relation of surface to the light source) it’s glowing more or less. Now what makes it a lambertian surface is that it emits (glows) differently in different directions. It emits most light in the normal direction of the surface, dropping like the cosine to zero parallel to the surface (Fig. P). BUT when you look at the surface, the more it’s slanted with respect to your viewing direction the more of it you see in a given “solid angle” (e.g. the amount of surface that contributes photons to a given pixel of your camera or retinal receptor in your eyeball). In fact, you observe just enough increased area (which goes as $1/\cosine$ of your observation angle) to cancel the lambertian fall-off in glow strength in that direction (which goes as the cosine of the surface’s normal to the viewing angle). SO (phew!) the perceived brightness varies because of the light source angle but *not* because of the viewing angle.

(End optional). We’re going to start with a Lambertian surface and work toward a model of a partly-specular reflectance mixed with lambertian...sort of a ‘semi-gloss paint’ idea. Forgive a new vector notation so I can emphasize dot products. The brightness of a Lambertian surface (as seen from anywhere) is

$$B = a * \vec{l} \cdot \vec{n}$$

Where a is a scalar giving the *albedo* of the surface (high for white, low for black) describing the fraction of incident light reflected at that point. \vec{l} and \vec{n} are unit vectors. \vec{l} is the direction to the light source (the same everywhere, recall), \vec{n} is the surface normal. So the brightness you see on a lambertian surface goes as the cosine of its inclination to the light-source direction (that is, as $\vec{l} \cdot \vec{n}$: Fig. P). Remember the geometric interpretation of dot product:

$$a \cdot b = |a| |b| \cos \theta$$

with θ the angle between the vectors a, b .

With Lambertian spheres nothing changes except that a plane has one normal everywhere and a sphere has all the possible normals, one per surface point on the sphere, all different (Fig. N). These normals are the ones you get if you bring the sphere's center to the origin of LAB and consider that for a point x on the sphere, its surface normal is in the same direction as the vector x (or \vec{x} if you like) itself. SO the sphere's surface points now tell you the normals to those points themselves. This interpretation of the sphere is sometimes called the *Gaussian Sphere*. If the origin of the sphere is at the origin of LAB, a point (a,b,c) ' on the sphere has surface normal $\text{Normalize}((a,b,c)')$, where Normalize turns $(a,b,c)'$ into a unit vector.

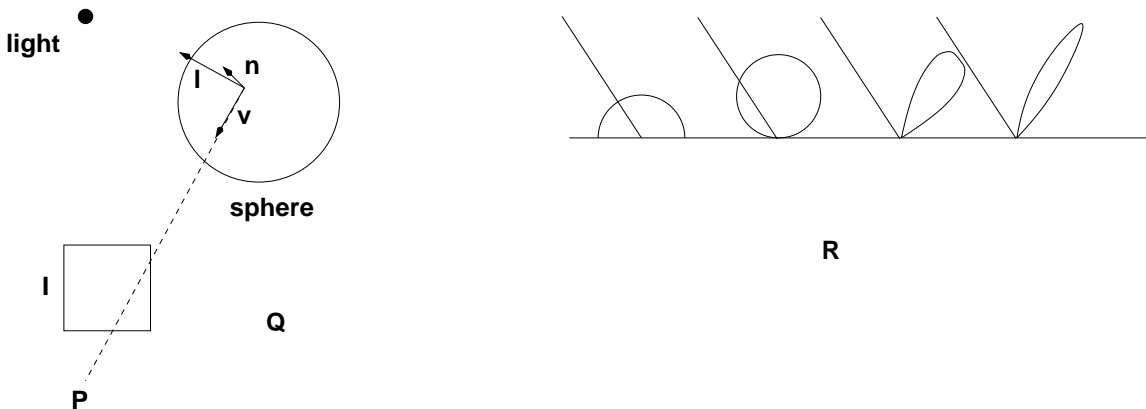
Notice that we can use the “sheet of paper” argument to infinitesimal facets of sphere surface, and argue that the point on the sphere whose surface normal is in the direction of the light source will appear brightest (from any viewpoint), and the sphere will appear less brightly illuminated as its surface bends away from the illumination direction, until at the terminator the sphere's surface is perpendicular to the incoming light rays and at that angle and beyond the sphere is self-shadowed, it is not illuminated at all. (Fig. N)

You should test for this *self-shadowing*: a sphere point with negative dot-product is “looking away” from the light source; it is shadowed by some sphere material. Its brightness is thus dictated by any other light sources, including the *ambient light* we mentioned in the Section on Directions – it may be zero, but it's not negative.

10 More on Albedo

You can paint your sphere with darker or lighter paint by making the albedo (the constant a in the reflectance formulae in Section 9) a function of where a point is on the sphere. This is how I put the dark segments on my spheres. I just used a function of a couple of the components of the surface normal of the sphere (since my sphere doesn't rotate). You can see that my albedo is one constant if these points (or angles) are in a set of ranges and is another if they are in another set. Or make up a completely different and more exciting paint job.

11 Semi-Gloss Reflectance



The idea is to have a Lambertian component and a mirror-like component to the reflectance function: then we'll have several parameters to play with: albedo, glossiness, and the amount of glossy vs. the amount of matte in the reflectance. We use a “phenomenological model”, or unprincipled hack, that produces plastic-looking results. (Part of the success story of modern computer graphics is coming up with more physically-principled and thus better looking reflectance functions).

Our new reflectance function looks like this:

$$F(i, e, g) = a(sC^n + (1 - s) \cos(i)).$$

Here i, e, g are all angles: i is the *incidence* angle of light upon the surface, and so we can write $\cos(i)$ as $\vec{l} \cdot \vec{n}$, which is familiar to us from the formula for Lambertian reflectance as it should be. We need three angles now to take into account the direction to the light source, the surface normal, and the direction to the viewpoint from a surface point. You clearly need this much information if you're doing specular highlights. Angles e and g are angles of emittance and phase, but we don't need to worry about them since we'll only need their cosines and those we can compute, as we did for $\cos(i)$, from the angles we do know about: $\vec{n}, \vec{l}, \vec{v}$ (see computation of C just below). So we could rewrite our reflectance function as

$$G(\vec{n}, \vec{l}, \vec{v}) = a(sC^n + (1 - s)(\vec{l} \cdot \vec{n})).$$

In $G(., ., .)$, a is the albedo scalar, and s is a scalar between 0 and 1 that gives the proportion of specular effect: $(1 - s)$ is the rest of the effect (due to Lambertian component).

C is the scalar cosine of an important angle: it is the angle between the ideal reflection direction and the viewpoint. So if the surface were a perfect mirror, if the angle isn't zero you don't see the light source at all. There is a whole cone of directions in which the angle is any other constant (up to 90 degrees, after which we don't care since the point is self-shadowed).

OK, so C is the cosine of this angle. If it's 1 you're looking in the direction of maximum reflectance (if the surface acts like a mirror). It goes to 0 as your angle increases to 90 degrees. So exponentiating it "sharpens" the cosine function. If its exponent $n = 0$ it just adds a constant value to all reflected light: if $n = 1$ it adds another lambertian component. As n gets bigger, the function sharpens and it becomes more selective: it adds less and less to directions that are off the "ideal reflecting" direction. It's an increasingly glossy semi-gloss function that changes from spherical to spike-like (Fig Ra). If n goes to infinity, the specular component of the sphere acts like a mirror sphere out in the garden, with one infinitely small highlight spot reflecting the light source (Fig. R).

So what is C ? Calculating it takes some magic vector manipulation. Let \vec{l} be the direction from a point on the sphere to the light source (same old direction, recall it never varies unless the light at infinity moves). Also \vec{v} is the direction to the viewpoint from the sphere point (or the negative of the direction from P to the sphere point), and \vec{n} is the surface normal at the point on the sphere (Fig. Q).

C , being a cosine, may be interpreted as a dot product: in fact, the dot of the direction of perfect reflectance and the direction \vec{v} from that point on the sphere to P . This dot product turns out (by the magic vector calculation) to be defined as a simple function of three more dot products of vectors we know (in fact by the cosines of i, e, g):

$$C = 2(\vec{n} \cdot \vec{l})(\vec{n} \cdot \vec{v}) - \vec{v} \cdot \vec{l}.$$

That's it. You do need to be careful not to add in either this or the Lambertian component when the surface normal's dot product with the light direction is less than 0 (again, self-shadowing rules).