A tutorial for 2d and 3d vector and texture mapped graphics

Version 0.50ß

If you want to reach me, see the end of this file.

---

1 An introduction to 3d transformations

First, let's introduce the idea of projections. We have to realize that we are trying to view a 3d world through a 2d «window». Thus, one full dimension is lost and it would be absurd ot think that we can reconstitute it wholly and recognizably with one less dimension. However, there are many forms of depth-cueing that we introduce in a 2d image to help us recognize that 3d dimension. Examples include but are not limited to reduced lighting or visibility at a distance, reduced size at a distance, out-of-focus blur, etc...

Our media is a screen, ideally flat, and the 3d world being represented is supposed to look as real as possible. Let's look at how the eyes see things:

µ §
Figure 1

The projection rays are the beams of light that reflect off the objects and head to your eye. Macroscopically, they are straight lines. If you're projecting these beams on a plane, you have to find the intersection of these beams with the plane.

µ §
Figure 2

The projection points are thus the intersection of the projection rays with the projection plane. Of course, projection points coming from projection lines originating from behind an object are obviously hidden, and thus comes the loss of detail from 3d to 2d. (i.e. closer objects can hide objects that lie further away from the eye).

The objects can be defined as a set of (x,y,z) points in 3d space (or 3space). Thus, a filled sphere can be defined as x**2+y**2+z**2<=1. But we need to simplify the problem right away because of machine limitations. We can safely assume the eye is always outside any object, thus we do not need do define the interior. [Note: a**b stands for «a raised to the bth power», thus, x**2 is x squared]

Now let us define a projection onto a plane. Many different projections could be defined, but the one that interests us is the following for it's simplicity. Projecting takes any given point (x,y,z) and localizes it on an (u,v) coordinate system. I.e. space (x,y,z) is mapped onto a plane (u,v). Let's transform (x,y,z) first. Multiplying (x,y,z) by a constant keeps it on a line that passes through (0,0,0) (the origin). We will define that line as the projection ray for our projection. Since all of these lines pass through the origin, we thus define the origin as our eye. Let the constant be k. Thus our transformation as of now is k(x,y,z). Now we want to set everything into a plane. If we select k as being 1/z, assuming z is nonzero, we get a projected point of (1/z)(x,y,z). This becomes (x/z, y/z, 1). Thus all points of any (x,y,z) value except z=0 will be projected with a value z'=z/z of one. Thus all points lie in the plane z=1. This projection thus fits our need for a projection with linear projector through the eye on a plane surface.

If you didn't get all that right away, you can always look it up later. The important thing to remember now is this: to project from (x,y,z) onto (u,v) we decide to use

u=x/z
v=y/z

But, I hear you say, this means the eye is facing parallel to the z axis, centered on the origin and is not banked. What if that is not what I desire?

First, it is obvious that if the eye is at location (a,b,c) it will not affect what one would see to move EVERYTING including the eye by (-a,-b,-c). Then the eye will be centered.

Second, and almost as obvious, if the eye is rotated around the x,y and z axis by agles of rx, ry and rz respectively, you can rotate EVERYTHING by the exact opposite angles, which will orient the eye correctly without changing what the eye would see.

Translation (moving) is a simple addition. I.e. (x,y,z)+(a,b,c) becomes (x+a,y+b,z+c). Additions are fairly quick. But there is the problem of the rotations.

1.1 Trigonometry

We will do it in two dimensions first.

µ §
Figure 3

We have the point defined by (x,y) and we want to rotate it with a counterclockwise angle of b to (x',y'). See figure 3. The radius r stays the same throughout the rotation, as per definition, rotation changes the angle but not distance to origin. Let r be the radius (not shown on figure 3). We need to express x' and y' as functions of what we know, that is x, y and b. The following can be said:

y'=sin(a+b)r
x'=cos(a+b)r

With the identities sin(a+b)=sin(a)cos(b)+sin(b)cos(a) and cos(a+b)=cos(a)cos(b)-sin(a)sin(b), we substitute.

y'=rsin(a)cos(b)+rcos(a)sin(b)
x'=rcos(a)cos(b)-rsin(a)sin(b)

But from figure 3 we know that

rsin(a)=y and
rcos(a)=x

We now substitute:

y'=ycos(b)+xsin(b)
x'=xcos(b)-ysin(b)

A special note: you do not normally calculate sinuses and cosinuses in real time. Normally, you build a lookup table, that is, an array. This array contains the precalculated value of sin(x) for all values of x within a certain range. That is, if the smallest possible angle in your model is on half of a degree, then you need a sinus table containing 720 entries, of which each entry is the value of the sinus of the corresponding angle. You can even note that a sinus is highly symmetric, that is, you need the values of sin(x) only for x between 0 and 90 degrees. Furthermore, you do not need a cosine table, as cos(x)=sin(x+90 degrees). This and fixed point mathematics may become obsolete as floating point processors become faster and faster. A good algorithm for calculating a sinus can be very quick today on a reasonably high performance platform, and it shouldn't use powers.

With that said, we will approach briefly 3d transformations.

In 3d, all rotations can be performed as a combination of 3 rotations, one about the x axis, one about the y axis and one about the z axis. If you are rotating about the z axis, use the exact same equation as above, and leave the z coordinate untouched. You can extrapolate the two other transformations from that.

Note that rotating first about the x axis and then about the y axis is not the same as rotating about the y axis and then the x axis. Consider this: point (0,0,1) rotated 90° about the x axis becomes (0,1,0). Then, rotated 90° about the y axis it remains there. Now, rotating first about the y axis yields (1,0,0), which rotated then about the x axis stays at (1,0,0), thus the order of rotations is important [(0,1,0) and (1,0,0) are not the same].

Now, if you are facing west (x axis, towards negative infinity). The user pulls the joystick up. You want the ship to dive down as a result of this. You will have to rotate the ship in the xy plane, which is the same as rotating about the z axis. Things get more complicated when the ship is facing an arbitrary angle.

Here I include a document authored by Kevin Hunter. Since he did a very nice job of it, I didn't see a reason to re-do it, especially when considering the fact that it would probably not have been as tidy. :-) So, from here to chapter two has been entirely written by Kevin Hunter.

Unit Vector Coordinate Translations

By Kevin Hunter

One of the more common means of doing “world to eye” coordinate transformations involves maintaining a system of “eye” unit vectors, and using a property of the vector dot product. The advantage of this approach, if your “eye” frame of reference is rotated with respect to the “world” frame of reference, is that you can do all the rotational calculation as simple projections to the unit vectors.

The dot product of two vectors yields a scalar quantity:

R dot **S** = |**R**| |**S**| cosa

where |**R**| and |**S**| are the lengths of the vectors **R** and **S**, respectively, and a is the angle between the two vectors. If **S** is a unit vector (i.e. a vector whose length is 1.0), this reduces to

R dot **S** = |**R**| cosa

However, a bit of geometry shows that |**R**| cosa is the portion of the **R** vector that is parallel to the **S** vector. (|**R**| sina is the part perpendicular). This means that if **R** is a vector with its origin at the eye position, then **R** can be converted from “world” space into “eye” space by calculating the dot product of **R** with the eye X, Y, and Z unit vectors. Getting a “eye origin” vector is trivial - all you need to do is subtract the “world” coordinates for the eye point from the “world” coordinates for the point to be transformed. If all this sounds a bit foggy, here’s a more concrete example, along with a picture. To

make the diagram easier, we'll consider the two-dimensional case first.

µ §
Figure 3a

In this figure, the "world" coordinates have been shown in their typical orientation. The "eye" is displaced up and left, and rotated to the right. Thus, the EyeX vector would have world-space coordinates something like (0.866, -0.500) [I've picked a 30 degree rotation just for the numbers, although my less-than-perfect drawing talents don't make it look like it). The EyeY vector, which is perpendicular to the EyeX vector, has world-space coordinates (0.500, 0.866). The eye origin point is at something like (1, 2).
Based on these numbers, any point in world space can be turned into eye-relative coordinates by the following calculations:

EyeSpace.x = (Px - 1.0, Py - 2.0) dot (0.866, -0.500)
= (0.866Px - 0.866) + (-0.500Py + 1.00)
= 0.866Px - 0.500Py + 0.134
EyeSpace.y = (Px - 1.0, Py - 2.0) dot (0.500, 0.866)
= (0.500Px - 0.500) + (0.866Py - 1.732)
= 0.500Px + 0.866Py - 2.232

The simple way of implementing this transformation is a two-step approach. First, translate the point by subtracting the eye origin. This gives you the eye-relative vector. Then, perform each of the three dot products to calculate the resulting X, Y, and Z coordinates in the eye space. Doing this takes the following operations for each point:

1. Three subtractions to translate the point
2. Three multiplications and two additions for the X coordinate
3. Three multiplications and two additions for the Y coordinate
4. Three multiplications and two additions for the Z coordinate

Since additions and subtractions cost the same in virtually any system, this reduces to a total of nine multiplications and nine additions to translate a point.

If you're striving for performance, however, you can make this run a little better. Note, in the two-dimensional example above, that we were able to combine two constants that fell out of the second line into a single constant in the third line. That last constant turns out to be the dot product of the particular eye coordinate vector and the negative of the eye origin point. If you're smart, you can evaluate this constant once at the beginning of a set of point transformations and use it over and over again.

This doesn't change the total number of operations - the three subtractions you take away at the beginning you put back later. The advantage, however, is that the constant is known ahead of time, which may let you keep it in a register or cache area. Basically, you've reduced the number of constants needed to transform the coordinates and the number of intermediate values, which may let you keep things in registers or the cache area. For example, this approach lets you use a stacked-based FPU in the following manner:

```
push    P.x
push    UnitVector.x
mult
push    P.y
push    UnitVector.y
mult
add
push    P.z
push    UnitVector.z
mult
add
push    CombinedConstant
add
pop     result
```

This algorithm never has more than three floating point constants on the FPU stack at a time, and doesn't need any of the intermediate values to be removed from the FPU. This type of approach can cut down on FPU-to-memory transfers, which take time.

Of course, if you don't have a stack-based FPU, this second approach may or may not help. The best thing to do is to try both algorithms and see which works better for you.

Maintaining Systems of Unit Vectors

Spinning In Space

To effectively use the system discussed in the previous section, you need to be able to generate and maintain a set of eye-space unit vectors. Generating the vectors is almost never a problem. If nothing else, you can use the world X, Y and Z vectors to start. Life gets more interesting, however, when you try to maintain these vectors over time.

Translations of the eye point are never a problem. Translations are implemented by adding or subtracting values to or from the eye origin point. Translations don't affect the unit vectors. Rotations, however, are another matter. This kind of makes sense, since handling rotations is what maintaining this set of vectors is all about.

To be effective, the set of three unit vectors have to maintain two basic properties. First, they have to stay unit vectors. This means that we can't let their lengths be anything other than 1.0. Second, the three vectors have to stay mutually perpendicular.

Ensuring that a vector is a unit vector is very straight-forward conceptually. All you need to do is to calculate the length of the vector and then divide each of its individual coordinates by that length. The length, in turn, can be easily calculated as SQRT(x^2 + y^2 + z^2). The only interesting part of this is the SQRT() function. We'll come back to that one later.

Keeping the set of vectors perpendicular takes a bit more work. If you restrict rotations of the vector set to sequences of rotations about the individual vectors, it's not too hard to do a decent job of this.

For example, if you assume that, in eye space, Z is to the front, X is to the right and Y is down (a common convention since it lets you use X and Y for screen coordinates in a 0,0 == top left system), rotation of the eye to the right involves rotating around the Y axis, with Z moving toward X, and X toward -Z. The calculations here are very simple:

$$\mathbf{Z}_{new} = \mathbf{Z}_{old}\cos a + \mathbf{X}_{old}\sin a$$
$$\mathbf{X}_{new} = \mathbf{X}_{old}\cos a + (-\mathbf{Z}_{old})\sin a$$

Of course, while you're doing this you need to save at least the original **Z** vector so that you don't mess up the **X** calculation by overwriting the **Z** vector before you get a chance to use it in the second equation, but this is a detail.

In a perfect world, when you're done with this rotation, **X**, **Y**, and **Z** are still all unit vectors and still mutually perpendicular. Unfortunately, there's this thing called round-off error that creeps into calculations eventually. With good floating point implementations, this won't show up quickly. If you're having to do fixed-point approximations (for speed, I assume) it will show up a lot faster, since you're typically storing many fewer mantissa bits than standard FP formats.

To get things back square again there are a couple of things you can do. The most straight forward is to use the cross product. The cross product of two vectors yields a third vector which is guaranteed (within your round-off error) to be perpendicular to both of the input vectors. The cross product is typically represented in determinant form:

$$\mathbf{A} \text{ cross } \mathbf{B} = \begin{vmatrix} \mathbf{x} & \mathbf{y} & \mathbf{z} \\ Ax & Ay & Az \\ Bx & By & Bz \end{vmatrix}$$

In more conventional format, the vector resulting from A cross B is

(AyBz - AzBy, AzBx - AxBz, AxBy - AyBx)

One thing about the cross product is that you have to watch out for the sign of things and the right-handedness or left-handedness of your coordinate system. **A** cross **B** is not equal to **B** cross **A** - it's equal to a vector in exactly the opposite direction. In a right-handed coordinate system, **X** cross **Y** is **Z**, **Y** cross **Z** is **X**, and **Z** cross **X** is **Y**. In a left-handed coordinate system, the arguments of the crosses have to be exchanged.

One tactic, then, is to rotate one of the vectors, and then, instead of rotating the other, to recalculate the other using the cross product. That guarantees that the third vector is perpendicular to the first two. It does not, of course, guarantee that the vector that we rotated stayed perpendicular to the stationary vector. To handle this, we could then adjust either the stationary vector or the one we originally rotated by another cross product if we wanted to. In practice, this frequently isn't necessary, because we rarely are rotating only about one axis. If, for example, we pitch and then roll, we can combine

rotations and crosses so that no vector has undergone more than one or two transformations since it was last readjusted via a cross product. This keeps us from building up too much error as we go along.

One other thing to remember about the cross product. The length of the generated vector is the product of the lengths of the two input vectors. This means that, over time, your calculated "cross" vectors will tend to drift away from "unitness" and will need to be readjusted in length.

Faster!

It is unusual in interactive graphics for absolute and unwavering precision to be our goal. More often (much more often) we're willing to give up a tiny bit of perfection for speed. This is particularly the case with relatively low-resolution screens, where a slight error in scale will be lost in pixel rounding. As a result, we'd like a measure of how often we need to readjust the perpendicularity of our unit vectors. As it happens, our friend the dot product can help us out.

Remember that we said that the dot product of two vectors involves the cosine of the angle between them. If our unit vectors are actually perpendicular, the dot product of any pair of them should then be zero, since the cosine of a right angle is zero. This lets us take the approach of measuring just how badly things have gotten out of alignment, and correcting it only if a tolerable threshold has been exceeded. If we take this approach, we "spend" three multiplications and two additions, hoping to save six multiplications and three additions. If we're within our tolerance more often than not, we're in good shape.

Foley et al point out that we can make this tradeoff work even better, however. Remember that the dot product represents the projection of one vector along another. Thus, if we find that X dot Y is not zero, the resulting value is the projection of the X vector along the Y vector (or the Y vector along the X vector, if you want). But if we subtract from the X vector a vector parallel to the Y vector whose length is this projection, we end up with X being perpendicular to Y again. The figure may help to make this clearer

µ §

Figure 3b

This lets use the following algorithm:

```
A = X dot Y
if (abs(A) > threshold)
{
        X -= Y times A
}
```

This approach uses three multiples and two adds if we don't need to adjust, and six multiplies and five adds if we do. This represents a savings of three multiplications over the "test and then cross product" approach. Note that, like the cross product approach, this technique will tend to change the length of the vector being adjusted, so vectors will need to be "reunitized" periodically.

Perfection?

If we want to use our unit vector system, but we need to model rotations that aren't around one of our unit axes, life gets more difficult. Basically, except in a very few circumstances (such as if the rotation is around one of the world axes) the calculations are much harder. In this case, we have at least two options. The first is to approximate the rotation by incremental rotations that we can handle. This has

to be done carefully, however, because rotations are not interchangeable - rotating about X then Y doesn't give the same answer as rotating around Y then X unless the angles are infinitely small.

The second option is to use quaternions to do our rotations. Quaternions are a very different way of representing rotations and rotational orientations in space. They use four coordinates rather than three, in order to avoid singularities, and can model arbitrary rotations very well. Unfortunately, I'm not completely up on them at this point, so we'll have to defer discussion of them until some future issue when I've had a chance to do my studying.

Efficient Computation of Square Roots

We mentioned earlier that we'd need to look at square roots to "unitize" vectors.

Let's assume for the moment that we need to calculate the square root of a number, and we don't have the hardware to do it for us. Let us assume further than multiplications and divisions are cheap, comparitively speaking. How can we go about calculating the square root of a number?

[Note: Intel-heads like me think this is an unnatural situation, but it does, in fact, occur out there. For example, the Power PC can do FP add/subtract/multiply/divide in a single cycle, but lacks sqrt() or trig function support.]

Isaac Strikes Again

Perhaps the simplest approach is to use Newton's method. Newton's method is a technique for successively approximating roots to an equation based on the derivative of that equation. To calculate the square root of N, we use the polynomial equation $x^2 - N = 0$. Assuming that we have a guess at the square root, which we'll call x_i, then next approximation can be calculated by the formula

$$x_{i+1} = x_i + (N - x_i^2) / (2x_i)$$

This begs two questions. First, how do we get started? Second, when do we get there? To answer the first part, Newton's method works really well for a well-behaved function like the square root function. In fact, it will eventually converge to the correct answer no matter what initial value we pick. To answer the second part, "Well, it all depends on where you start."

To get an idea of how this iteration performs, let's expand it a bit. Let's assume that our current guess (x_i) is off by a value "e" (in other words, the square root of N is actually $x_i + e$.) In this case,

$$\begin{aligned} x_{i+1} &= x_i + ((x_i + e)^2 - x_i^2) / (2x_i) \\ &= x_i + (x_i^2 + 2ex_i + e^2 - x_i^2) / (2x_i) \\ &= x_i + (2ex_i + e^2) / (2x_i) \\ &= x_i + e + (e^2 / 2x_i) \end{aligned}$$

But since "$x_i + e$" is the answer we were looking for, this says that the error of the next iteration is "$e^2/2x_i$". If the error is small, the operation converges very quickly, because the number of accurate bits doubles with each iteration. If the error is not small, however, it's not so obvious what happens.

I'll spare you the math, and simply tell you that if the error is large, the error typically halves each iteration (except for the first iteration, in which the error can actually increase). So, while a bad initial guess won't prevent us from finding an answer, it can slow us down a lot.

So how do we find a good initial guess? Well, in many cases, we have a good idea what the value ought to be. If we're re-unitizing a vector, we suspect that the vector we're working with is already somewhere near a length of 1.0, so an initial guess of 1.0 will typically work very well. In this situation, four iterations or so usually gets you a value as accurate as you need.

What if we're trying to convert a random vector into a unit vector, or just taking the square root of a number out of the blue? In this case, we can't be sure that 1.0 will be a very good guess. Here are a few things you can do:

1. If you have easy access to the floating point number's exponent, halve it. The square root of A x 2^M is SQRT(M) x $2^{M/2}$. Using A x $2^{M/2}$ is a really good first approximation.
2. If you <u>are</u> taking the length of a vector (x,y,z), abs(x)+abs(y)+abs(z) won't be off by more than a factor of 2 - also a good start.
3. It turns out that the operation will converge faster if the initial guess is higher than the real answer. If all else fails, using N/2 will converge pretty quickly.

Don't go crazy trying to get a good first guess. Things will converge fast enough even if you're somewhat off that a fast guess and an extra iteration may be faster than taking extra time to refine your guess.

If You Don't Like Newton...

If you're looking for a really accurate answer, Newton's method may not be for you. First, you can't, in general, just iterate a fixed number of times. Rather, you have to check to see if the last two iterations differ by less than a threshold, which adds calculation time to each iteration. If you're in fixed-point land, the assumption we made earlier that multiplications were cheap may not be the case. In this case, there is an alternate closed-form algorithm you can use that will result in the "exact" answer in a predefined amount of time. This algorithm also uses an incremental approach to the calculation.

Since $(x + a)^2 = x^2 + 2xa + a^2$, we can iterate from x^2 to $(x+a)^2$ by adding $2xa + a^2$ to our working sum. If a is2^N, however, this translates into adding 2^{N+1} x + 2^{2N}. Since we can multiply by 2^N by shifting to the left by N bits, we can take a square root by use of shifts and subtractions. To take the square root of a 32-bit number, for example, this results in an algorithm that looks something like this:

```
remainder = valueSquared;
answer = 0;
for (iteration = N; iteration >= 0; iteration--)
{
        if (remainder >= (answer << iteration+1) + (1 << iteration * 2))
        {
                remainder -= (answer << iteration+1) + (1 << iteration * 2);
                answer += 1 << iteration;
        }
```

}

A “real” algorithm won’t repeatedly perform the shifts on the 5th line. Rather, an incremental approach can be taken in which the answer and “1 bit” is kept in pre-shifted form and shifted to the right as the algorithm proceeds. With appropriate temporary variable space, this algorithm shifts two values and does two subtractions and either one or zero add or “or” per iteration, and uses one iteration for each two bits in the original value.

This algorithm can be used very effectively on integer values, and can also be adapted to take the square root of the mantissa portion of a floating-point number. In the latter case, care has to be taken to adjust the number into a form in which the exponent is even before the mantissa square root is taken, or the answer won’t be correct.

Table It

If you don’t need absolute precision, there is a table-driven square root algorithm that you can use as well. The code for the 32-bit FP format (“float”) is in Graphics Gems, and the 64-bit FP format (“double”) is in Graphics Gems III. The source code is online at ftp:://Princeton.edu/pub/Graphics/GraphicsGems. (Note the capital ‘G’. There is also a /pub/graphics (small ‘g’) which is not what you want...)

The way this algorithm works is to reach inside the IEEE floating point format. It checks the exponent to see if it is odd or even, modifies the mantissa appropriately for odd exponents, and then halves the exponent. The mantissa is then converted to a table lookup with a shift and mask operation, and the new mantissa found in a table. The version in Graphics Gems III is set up to allow you to adjust the size of the lookup table, and thus the accuracy of the operation.

Whether or not this method is appropriate for you depends on your tolerance for accuracy vs. table size tradeoffs. The algorithm runs in constant time, like the previous one, which is an advantage in many situations. It’s also possible to combine them, using a table lookup to get the first N bit guess, and then finishing it out using the bit shifting approach above.

Trig Functions

Unfortunately, trig functions like sine and cosine don’t lend themselves nicely to methods like the square root algorithms above. One of the reasons for this is that sine and cosine are interrelated. If you try to use Newton’s method to find a cosine, you need to know the sine for your guess. But to figure out the sine, you need cosine. Department of Redundancy Department.

One alternate approach you can take is to use a Taylor series. A Taylor series is a polynomial that will approximate a sine or cosine to any desired accuracy. The problem with this is that the Taylor series for sine and cosine don’t converge really quickly. They don’t do too bad, but you still have to do a bunch of computation.

A better approach in this case is probably to use a combination of computation and a look-up table, like the last square root algorithm. Since sine and cosine are cyclic, and interrelated, you only really need to be able to calculate one of them over the range 0 - p/2. Given this, the rest of the values can be determined. Depending on how much calculation you want to do, how much table space you want to spend, and how accurate your answer needs to be, you can break up the region above into a fixed number of regions, and use some form of interpolation between points. If you’re in Floating Point

land, over a relatively short region, you can fit a quadratic or cubic function to either the sine or cosine function with pretty good accuracy. This gives you a second or third order polynomial to evaluate and a little bit of table lookup overhead. For all but the lowest accuracy cases, this will probably net out faster than calculating the number of Taylor terms you'd need.

If you're in Fixed Point Land, you may be better off with a table lookup and linear interpolation, rather than using a higher-order polynomial. For comparable accuracy, this approach will probably need more table space, but if you're avoiding floating point, you'll probably be avoiding integer multiplications as well. (Particularly because you have the decimal point adjustments to worry about). The linear interpolation can be reduced to a single multiply by storing delta information in the table, so all you have to do is multiply the bits you stripped off while building the table index (the "fractional part" by this delta value and add it to the table entry.

One trick you can use in either case to make your table lookups easier is to measure your angles in 2^{16} units rather than radians or degrees. 2^{16} units basically consist of the following mapping:

Degrees	Radians	2^{16} units
0	0	0
45	p/4	0x2000
90	p/2	0x4000
180	p	0x8000
270	3p/2	0xC000
359.999	2p-e	0xFFFF

The neat thing about this set of units is that they wrap just the way that radians do (and degrees are supposed to). They give you resolution of about 0.005 degrees, and they make table lookups very easy, since you can shift-and-mask them down into table indices very easily.

Interestingly enough, this is something I'd devised for myself late last year (feeling very proud of myself) only to see the technique published in Embedded Systems a month or so later. Just goes to show you...

---

2 Polygon drawing on a raster display in two dimensions

The second main problem is to display a filled polygon on a screen. We will discuss that problem here. First, let us approach the 2d polygon.

We will define a polygon as a set of points linked by segments, with a defined inner region and outer region. The inner region need not be in a single piece.

Our rule for determining if a point is in a given region is to draw a line from infinity to that point. If that line crosses an odd number of segments (edges), the point is within the polygon. Otherwise it is not.

µ §
Figure 4

Drawing a line from infinity to any of the gray areas of the star will cross an odd number of edges. In the white areas, it will cross an even number of edges. Try it.

µ §
Figure 5

The polygon shown in a, b and c are all the same. In a, the inner region has been shaded, but this has been omitted in b and c for clarity.

The polygon shown in a can be listed as vectors, as in b or c. Let us agree that both versions are equally satisfying. In b we enumerated the edges in a counterclockwise continuous fashion, meaning that the head of each vector points to the tail of another. In c, we made each vector point generally downward, or more strictly, each vector's topmost point is its tail. These vector are noted (a,b)-(c,d) where (a,b) is the location of the tail of the vector, and (c,d) is the location of the head (arrow) of the vector. In diagram c, for all vectors d<b (or b>d, either way). (Note, we will see what to do with b=d later. That being a special case, we will not discuss it now).

As we are drawing on a screen (supposedly), it would be interesting to be able to color each pixel once and only once, going in horizontal or vertical lines (as these are easy to compute). We will choose horizontal lines for hardware reasons. Thus, we start scanning from infinitely upwards to infinitely downwards using horizontal scanlines (e.g., the «first» scanline is way up, and the last is way down). Now, our screen being finite, we need only to draw those lines that are on screen. At each pixel, we will draw an imaginary line from the infinite to the left (horizontal line) to the current pixel. If it crosses an odd number of edges, we color it. Otherwise, we don't.

This means that from infinite left to the first edge, we do not color. Then, from first edge to second edge, we color. Then, from second edge to third edge, we do not color. From third edge to fourth edge, we color. And so on. If you want, you can always draw a background bitmap left from the first edge, from second edge to third edge, etc.... Thus, you could put the picture of a beautiful blue sky behind at the same time you're drawing a polygon.

Now, let's start that over again. We are coming from infinitely upward and drawing. We do not need to consider every edge, only those that are on the current scanline (horizontal line). We will thus build an inactive edge list (those edges that have not yet been used in the drawing process) and an active edge list (those edges that are currently in use to draw the polygon, i.e. that intercept the current scanline). Thus, at first, all edges (named m,n,o,p in figure 5) would be in the inactive edge.

As the algorithm reaches the topmost line of the polygon, any relevant edge will be transfered from the inactive edge list to the active edge list. When an edge becomes unused (we're lower than the lowermost scanline), we remove it from the active list and throw it to the dogs. (# grin #)

Now you will realize that at each scanline we have to check every edge in the inactive edge list to see if we need to transfer them to the active edge list. To accelerate that, we can sort the inactive edge list in decreasing order. Thus, the ordered list would become {m,p,n,o}. Then, we need only to check the first vector at every scanline. Since they point downward, we need only to check the first point of it.

If you get any horizontal lines in the edge list, remove them from the list. Horizontal lines can be identified easily as b=d. Proving that it works is left as an exercise (# muhaha! #).

The polygon drawing algorithm (scan-line algorithm).

```
I=inactive edges list, initially all edges
A=active edges list, initally empty

sort I with highest first endpoint first

for each scanline i starting from the top of the screen
        does the first edge in I start at scan line i?
                Yes, transfer all edges that should start now from I to A
        find x value of all edges in A for current scanline (see the line algorithm below)
        sort all edges in A with increasing x-intercept value (the thing we jest calculated)

        previous_x_intercept<-leftmost pixel on screen
        for all edges n in A, counting by two, do
                draw background between previous_x_intercept and edge n in A
                fill polygon between edge n and edge n+1 in A
                previous_x_intercept<-x intercept of edge n+1
        end for

        draw background for the rest of the scan line (from previous_x_intercept to end of line)
        for all edges for which we have reached the bottom
                remove the edge from A
        end for
end for
```

2.1 A line algorithm for the polygon drawing algorithm

This section discusses a somewhat shortened version of Bresenham's line drawing algorithm. It is optimized for drawing filled polygons.

You assuredly realize you need to calculate the x coordinate of the intersection of a line with an

horizontal line. Well consider the line equation that follows:

x=dy/dx x y + k

If the line is defined with the vector (a,b)-(c,d), then we define dy as d-b, dx as c-a. Say x0 is the x coordinate of the topmost point of the segment.

x0=a

Say xn is the x intercept value of the segment for the nth line from the top. Then, xn can be defined as follows:

xn=dy/dx x n + a

Say we know xn-1 and want to know xn based on that knowledge. Well it is obvious that:

xn-x(n-1)=dy/dx x n + a - [dy/dx x (n-1) + a]
xn-x(n-1)=dy/dx
xn=dy/dx+x(n-1)

Thus, knowing xn-1 and adding dy/dx yields xn. We can thus do an addition instead of a multiply and an add to calculate the new x intercept value of a line at each scanline. An addition been at least 5 times as fast as an add, usually more in the 15 times range, this will speed things up considerably. However, it might annoy us to use floating point or fixed point, even more so with roundoff error.

To this end, we have found a solution. dy/dx can be expressed as a+b/d, b<d. a is the integer part of dy/dx. Substituting, we find that

xn=a+b/d+xn-1

Thus, we always add a at each scanline. Furthermore, when we add a sufficient amount of b/d, we will add one more to xn. We will keep a running total of the fraction part. We will keep the denominator in mind, and set the initial numerator (when n=0) to 0. Then, at each line we increase the numerator by b. If it becomes greater than d, we add one to xn and decrease the numerator by d. In short, in pseudocode, we draw a segment from (x0,y0) to (x1,y1) this way:

```
dx=x1-x0
dy=y1-y0
denominator=dy
numerator=dx modulo dy
add=dx divided (integer) dy
running=0
x=x0
for each line (0 to n) do
        x=x+add
        running=running+numerator
        if running>=denominator then
                running=numerator-denominator
```

```
            x=x+1
        end if
        do whatever you want with line here, such as drawing a pixel at location (x,line)
next line
```

---

3 3d polygon drawing

The first step is to realize that a projected 3d line with our given projection becomes a 2d line, that is it stays straight and does not bend. That can be easily proved. The simplest proof is the geometric one: the intersection of two planes in 3d is a line in the general case. The projector all lie in the plane formed by the origin ant the line to be projected, and the projection plane is, well, a plane. Thus, the projection is contained in a line.

This means that all the boundaries of a polygon can be projected as simple lines. Only their endpoints need be projected. Then, you can simply draw a line between the two endpoints of each projected edge (or more precisely, draw a 2d polygon with the given endpoints).

However, there comes the problem of overlapping polygons. Obviously, the color of a pixel must be that of the closest polygon in that point (well for our objective, that suffices). But this means that we must determine the distance of each polygon that has the possibility to be drawn in the current pixel, and that for the whole image. That can be time consuming.

We will make a few assumptions. First, we will assume no polygons are interpenetrating; this means that to come in front of another polygon, a polygon must go around it. This means that you only need to determine what polygon is topmost when you cross an edge.

Less obviously, but equally useful, if from one scanline to the next, you encounter the same edges in the same order when considering left to right, you do not need to recalculate the priorities as it has not changed.

Inevitably, though, you will need to determine the depth of the polygon (z value) in a given pixel. That can be done rather easily if you carry with the polygon its plane equation in the form $Ax+By+Cz+D=0$. Remember that the projection ray equation for pixel (u,v) is $x=zu$ and $y=zv$. Feed that x and y in the plane equation and you find that

$Azu+Bzv+Cz+D=0$
$z(Au+Bv+C)=-D$
$z=-D/(Au+Bv+C)$

Thus you can easily calculate the z coordinate for any plane in a given (u,v) pixel.

The cautious reader will have noticed that it is useful to do some of these multiplications and additions incrementally from scanline to scanline. Also note that if Au+Bv+C equals zero, you cannot find z unless D is also zero. These (u,v) coordinates correspond to the escape line of the plane of the polygon (you can picture it as the line of «horizon» for the given plane). The projection ray is parallel to the plane, thus you never cross it.

This brings me to another very important thing about planes. Another way to define a plane is to give a normal vector. A vector and a point of reference generate one and only one plane perpendicular to the vector and passing through the point of reference. Think of a plane. Think of a vector sticking up perpendicular to it. That vector is called normal to the plane. It can be easily proved that this vector is (A,B,C) with A,B,C being the coefficients of the plane equation Ax+By+Cz=D. We might want to normalize the vector (A,B,C) (make it length 1), or divide it by SQRT(A2+B2+C2). But to keep the equation the same, we must divide all member by that value. The equation thus becomes A'x+B'y+C'z=D'. If we want to measure the distance perpendicularly to the plane of a point (a,b,c), it can be shown that this (signed) distance is A'a+B'b+C'c+D' (you can always take the absolute value of it if you want an unsigned distance). That might be useful for different reasons.

It is also notable that since the A, B and C coefficients are also the coords of the normal vector, and that rotating a vector can be done as we saw previously, it is easy to rotate the plane. If the vector is normalized in the first place, it will stay normalized after rotation. After rotation, you will ignore the D component of the plane, but you do know the coordinates of a point in the rotated plane. Any point in the polygon will fit the bill. Thus, you can deduce D. In fact, when performing back-face culling (see chapter 4), you will calculate D even as you are performing some optimization.

Normally, you will build an inactive edge table (IET) in which you will store all edges of all polygons to be rendered. Also initialize an empty list called the active edge table (AET). In the IET, make sure that the edges are sorted in increasing values of y for the topmost endpoint of any edge. Also, as per the 2d polygon drawing algorithm, discard any horizontal edge.

As you scan from the top of the screen to the bottom of the screen, move the edges from the IET to the AET when you encounter them. You do not need to check every edge in the IET as they are sorted in increasing y order, thus you need only to check the first edge on the IET.

As you are scanning from left to right, each time you cross an edge, you toggle the associated polys on or off. That is, if poly X is currently "off", it means you are not inside poly X. Then, as you cross an edge that happens to be one of poly X's edge, turn it "on". Of all polys that are "on", draw the topmost only. If polys are not self-intersecting, you only need to recalculate which one's on top when you cross edges. Your edge lists must contain pointers to thir parent polygons. Here's a pseudo-code for the algorithm:

```
Let polylist be the list of all poly
Put all edges of all polygon in IET
Empty AET
for Y varying from the topmost scanline to the last
        while the edge on top of the IET's topmost endpoint's Y coodrinate equals Y do
                take the edge off the IET and insert it in the AET with an insertion sort
```

```
            so that all edges in the AET are in increasing X values.
        end_while

        discard any edge in AET for which the second endpoint's Y value is Y

        previous_X=minus infinity
        current_poly=none
        for i=first edge through last edge in AET
            for X varying from previous_X to i's X coordinate
                draw a pixel at position (X,Y), attribute is taken from current_poly
            end_for
            toggle polygons that are parent to i
            current_poly=find_topmost_poly()
        end_for

        update intercept values of all edges in AET and sort them by increasing X intercept
end_for
```

A recent uproar about "s-buffers" made me explain this algorithm a little more carefully. Paul Nettle wrote a FAQ explaining something that he calls "s-buffering". It is essentially an algorithm where, instead of calculating the spans on the fly as above, you buffer them, that is, you precalculate all spans and send them to a span buffer. Then you give the span buffer to a very simple rasterizing function. It's a bit harder to take advantage of coherence in that case (an example of coherence is when we note that if the order of the edges do not change from one scanline to the next, we don't need to recalculate which spans are topmost, assuming polygons aren't interpenetrating).

Another very nice algorithm is the z-buffer algorithm, discussed later. It allows interpenetrating polygon and permits mixing rendering techniques in a rather efficient way (for real-time rendering purposes).

---

4 Data Structures

A very important aspect of 3d graphics is data structures. It can speed up calculations by a factor of 6 and up, only if we put some restriction on what can be represented as 3d entities.

Suppose that everything we have in the 3d world are closed polyhedras (no self-crossing(s)!). Each polyhedra is constituted of several polygons. We will need each polygon's normal and vertexes. But how we list these is where we can improve speed.

It can be proved that each edge is shared by exactly two polygons in a closed polyhedra. Thus, if we list the edges twice, once in each polygon, we will do the transformations twice, and it will take twice as long.

To that, add that each vertex is part of at least 3 points, without maximum. If you list each point in every vertex it is part of, you will do the transformations at least thrice, which means it will take thrice the time. Three times two is six. You're doing the same transformations at least six times over.

Thus, you need to list the vertexes in every polygon as pointers to vertexes. This way, two polygons can share the same data for a shared vertex, and will allow you to do the transformations only once.

The same goes for edges, list the endpoints as pointers to endpoints, thus you will spare yourself much runtime work.

Another interesting thing. All polygons have an interior and exterior face in that model. Make it so that the normal to all polygons point towards the exterior face. It is clear that, if you are standing outside all objects, and all objects (polyhedra) are closed, you cannot see an inside face. If, when transforming polygons, you realize a polygon is facing away from the eye, you do not to transform it, just skip it. To know if it's pointing away from you do the following:

This is called back-face culling. Take the (a,b,c) vector from the eye to any one endpoint of the polygon. (E.g., if the eye is at (m,n,o) and a vertex of the poly is at (i,j,k), then the said vector would be (i-m,j-n,k-o).) Take also (u,v,w) the normal vector to the polygon. No matter what direction you are looking, do the following operation:

au+bv+cw

If the result is positive, then it is facing away from you. If it is negative, it is facing towards you. If it is null, it is parallel to you. The operation is called scalar multiply of two vectors. It is very notable that the above scalar multiplication of two vectors is interestingly enough the D part of the plane equation Ax+By+Cz=D.

You can also illuminate objects with a very distant light source (e.g. the sun) very easily. Take the vector pointing away from the light source (a,b,c) and the normal of the plane (u,v,w). Do a scalar multiplication of both as above. The result can be interpreted as the intensity of the lighting or shadow for the polygon. The approximation is good enough to give interesting results. You might also want to affect the shading of the polygon according to its distance to origin, to yield a fog result.

---

5 Texture mapping

What we are about to do would probably be more aptly named pixmap mapping onto a plane. We want to take a pixmap (or bitmap, although less appropriate a word, it is very popular) and «stick» it to the surface of a polygon, cutting away any excess. The result can be quite impressive. This, however, can be a painfully slow operation. We will discuss here of the mathematics involved, and after that, we will try to optimize them.

Let's give a polygon its own coordinate system, (i,j). Thus, all points on the said polygon can be localized in (i,j), or (x,y,z), or, once projected, in (u,v).

µ §
Figure 6

Figure 6 illustrates the problem. The blue plane is the projection plane. The green triangle is a polygon we are projecting on that plane and then scan-converting in (u,v) coordinates. What we want to do is find, what (i,j) point corresponds to any given (u,v) point when projected this way. Then, the point (i,j) can be indexed in a pixmap to see what color this pixel in (u,v) space should be.

Let the i and j vectors be expressed in (x,y,z) coordinates, which they should normally be. We want a general solution.

i=(a,b,c)
j=(d,e,f)

Let also the origin of (i,j) space be pointed to in (x,y,z) by the vector k=(m,n,o). Thus, a point at location (p,q) in (i,j) space is the same as a point at location pi+qj+k in (x,y,z) space. Furthermore, we have the projection ray defined as follow: it shoots through point (r,s) in (u,v) space. This corresponds to point (r,s,1) in (x,y,z) space. The equation of that ray is thus t(r,s,1) where t can take any real value. Simplifying, we find that the ray is (tr,ts,t), or

```
      x=tr
R:    y=ts
      z=t
```

or

```
      x=zr
R:    y=zs
      z=z
```

The point in the plane is

```
        x=pa+qd+m   (1)
P:      y=pb+qe+n   (2)
        z=pc+qf+o   (3)
```

We want to find (p,q) as a function of (r,s). Here is what i did in stupid MathCAD. [MathCAD flame deleted] I'm sorry I didn't use the same variable names in MathCAD...

#######Begin picture of mathcad screen#########

########End picture of mathcad screen#########

These equations for p and q can be optimized slightly for computers. Since it is usually faster on a computer to do a(b+c) than ab+ac, and that both are equivalent, we will try to factorize a little.

p=(Mv+Nu+O)/(Pv+Qu+R)
where M,N,O,P,Q,R are as follows:
M=CXq-AZq
N=BZq-CYq
O=AYq-BXq
P=ZqXp-ZpXq
Q=YqZp-YpZq
R=YpXq-YqXp

They are all constants, so they need only to be calculated once to draw the whole image.

Similarly for q, we have:
q=(Sv+Tu+U)/(Vv+Wu+X)
and
S=AZp-CXp
T=CYp-BZp
U=BXp-AYp
V=ZqXp-ZpXq
W=YQZP-YPZQ
X=YqXp-YpXq

All these constants should be calculated only once. Now we can simplify a little with, knowing that P=V, Q=W, R=X.

p=(Mv+Nu+O)/(Pv+Qu+R)
q=(Sv+Tu+U)/(Pv+Qu+R)

This is noteworthy: (Q,P,R)(u,v,1)=uQ+vP+R, the denominator. But, (Q,P,R)=(Xq,Yq,Zq)x(Xp,Yp,Zp). This cross-product of two vectors will yield a vector that is perpendicular to the first two (i.e. a normal to the plane). If these two vectors are of length 1, then the cross product will also be of length 1. But we might already have the normal from previous transformations (e.g. if we're doing back-face culling). Then, we won't need to calculate (Q,P,R) as it is the normal to the plane. Or we can deduce the normal from these equations if we don't have it.

The denominator needs to be calculated only once per pixel, not for both p and q. The numerator for p and q is different, though. Furthermore, It is obvious that Mv+O are constant throughout a constant-v line (horizontal line). Thus, it needs to be calculated only once per line. We can spare ourselves the multiply per pixel doing it incrementally. (We're still stuck with a divide, though.) Example, let V=Mv+O, for a given v. Then, when calculating Mv+Nu+O, we can use V+Nu. If we know V+Nua, then we can calculate V+N(ua+1) as follow: V+Nua+N

Let W=V+Nua, then we get

W+N

That's a simple add instead of a multiply. But we can not get rid of the divide, so we're stuck with a divide per pixel (and at least two adds) per coordinate.

There is a way to solve this, however, and here's how to do it.

First, it is interesting to note the following. The plane equation is Ax+By+Cz=D, and x=zu, y=zv, where (u,v) is the screen coordinates of a point, and (x,y,z) is the corresponding world coordinate of the unprojected point. Thus,

Auz+Bvz+Cz=D

What happens when z is a constant? Let us express v as a function of u.

Bvz=-Auz-Cz+D
v=(-A/B)u+(-C/B)+D/(Bz)

Let M=-A/B, and N=d/(Bz)-C/B then we have

u=Mv+N

a linear equation. Furthermore, M is independant of z. Thus, a slice of constant z in the plane is projected as a line on (u,v). All of these slices project in lines that are all parallel to each other (e.g. the slope, M, is independant of the slice taken).

Now, a slice of constant z in the original Ax+By+Cz=D plane is a line. Since z is constant throughout that line, all points of that line are divided by the same number, independantly of (x,y). Thus, the projected line is merely the original line scaled up or down.

Up to now, we have been rendering polygons with a scanline algorithm where the scanlines are horizontal. First, we have to conceive that the scanlines need not be horizontal, merely parallel. If we use a slope of M for the scanlines (M=-A/B, as above), then we will be performing scans of constant z value. Two advantages are very obvious. First, it is easy to calculate the z value for a large number of pixel (it is constant throughout a scanline) and second, texture mapping the scanline becomes a mere scaling operation, which can be done incrementally. Furthermore, M need be calculated only once for the whole polygon, while N requires at most a divide and an add per scanline.

Note that if abs(A)>abs(B), you might want to express v as a function of u instead of the above, so that

the slope ends up as being -B/A instead. (E.g., make certain that the slope is less than or equal to 1 in absolute value). There is a degenerate case: A=B=0. This is the case where the whole plane is of constant z value. In that case, you can use horizontal scan lines and just scale the texture by a factor of 1/z.

Thus, we're going to be rasterizing the polygon with nonhorizontal scanlines of slope M. The equation of a scanline can be expressed as:

v=Mu+Vi

Where Vi is the v coordinate when u is 0. All scanlines being parallel, M never changes, and Vi identifies uniquely each and every scanline. Say i=0 corresponds to the topmost scanline. Say we want to find the intersection of the scanline with a polygon edge. What we're really interested in is the u value of the intersection of the two lines. The edge can be represented as being:

v=Nu+K

Where K is the v value for the edge when u=0. Now, intersecting the two yields

Nu+K=Mu+Vi
(N-M)u=(Vi-K)
u=(Vi-K)/(N-M)

That is, assuming N-M is not zero, which shouldn't be because we're not including edges parallel to the scanlines in our scan line algorithm, thus N-M is never zero.

Now, V(i+1) is (Vi)+1. When we go from scanline i to scanline i+1, V is incremented by one. We thus get u', the intersection of the two lines for the next scanline, as being:

u'=(Vi+1-K)/(N-M)
u'=(Vi-K)/(N-M)+1/(N-M)
u'=u+1/(N-M)

It is thus possible to calculate u from scanline to scanline incrementally, bresenham stlye. To clarify things up a bit: if P1 and P2 delimit the edge, then let (p,q) be P2 minus P1 (that is, p is the delta u and q is the delta v). Therefore, N is q/p (by definition of slope). Now, we know M to be of the form -B/A (from the plane equation, above). We need to calculate N-M for our incremental calculations, which is relatively easy. We have

N-M=q/p-(-A/B)
N-M=q/p+A/B
N-M=(qB+Ap)/(Bp)

Thus, u' is really:
u'=u+Bp/(qB+Ap)

At one point in the algorithm, we will need to calculate which scanline contains a particular point. As an example, when sorting from topmost edge downwards, we need to use a definition of top that is

perpendicular to the scanline used. Then, we have the slope of the scanline, which is, say, m. If the equation of the line that interests us is:

y=mx+b

Then, we can sort the edges according to increasing b values in the above equation. If we want to calculate b for point (i,j), simply isolate b in the above equation.

This last bit is called constant-z texture mapping. It is not particularly useful in the general case, for a number of reasons, mainly added aliasing, and span generation problems. However, it has a very well known and very used application. If we have vertical walls and horizontal floors exclusively, and that the eye is never banked, then the following is always true: the constant-z lines are always either horizontal or vertical. For example, the walls' constant-z lines are vertical while the floors' and ceilings' are horizontal. So basically, you can render floors and walls using the techniques described in this chapter in a very straightforward manner, if you use both horizontal and vertical scanlines. Just use the last bit about constant-z texture mapping and optimize it for horizontal and vertical scanlines. It is to note that looking up or down will not affect the fact that walls will have vertical scan-lines and floors and ceilings will have horizontal scan-lines.

---

6 Of logarithms

If we have many multiplies or divides to make, little time and lots of memory, and that absolute precision is not utterly necessary, we may use the following tricks.

Assuming x and y are positive or null real numbers, and b a base for a logarithm, we can do the following: (note: x**y means x raised to the yth power, logby denotes base b logarithm of y).

logb(xy)=logbx+logby
b**logb(xy)=b**(logbx+logby)
xy=b**(logbx+logby)

Now, you might say that taking the log or the power of a value is very time consuming, and it usually is. However, we will reuse the trick that we used to do not so long ago, when pocket computers were not available, we will make a logarithm table. It is easy to store the values of all possible logaritms within a finite, discrete range. As an example, if we are using 16 bit fixed points or integer, we can store the result of a log of that number as a table with 65536 entries. If each entry is 32 bits wide, which should be sufficient, the resulting table will use 256kb of memory. As for the power table, you

can build another lookup table using only the 16 most significant bits of the logarithm, yielding a 16 or 32 bit wide integer or fixed point. This will yield a 128 to 256kb table, for a grand total of under 512kb. With today's machines, that should not be a problem. The only problem is that you must then limit your scope to 16 bits numbers. You realize of course that you can not multiply if one or both of the numbers are negative. Well, you can do it this way. Say x<0, then let u=-x, thus u>0. Then, xy=-uy, and uy you can calculate with this method. Similarly,

x/y=b**(logbx-logby)

Thus divides are also a simple matter of looking it up in the table and subtracting.

Powers are made much more simple also. I won't say anything long winded about powers, but here is, in short:

logb(x**y)=y logbx
x**y=b**(y logbx) (*)
And from there, even better yet:
x**y=b** (b** [logby+logb(logbx)])

For y=constant, some shortcuts are usually possible. Since ax can sometimes be very quickly calculated with left shifts, you can use (*) instead of the last equation. Very common example:
320y=(256+64)y
320y=256y+64y
320y=(y<<8)+(y<<6)

Where x<<y denotes "x, left shifted y times". A left shift of y is equivalent of multiplying by 2**y. The ancient egyptians only knew how to multiply this way (at least, as far as I know, for a while, like for the pyramids and stuff - I think they even calculated an approximation of pi during that time, and they couldn't even multiply efficiently! :-)

---

7 More on data structures and clipping

Usually, the space viewed by the eye can be defined by a very high square-based pyramid. Normally, it has no bottom, but in this case it might have one for practical purposes. Let us say that the eye has a width of vision (horizontally and vertically) of 90 degrees. (Since it is square, that angle is slightly larger when taken from one corner to the opposite.) Thus, we will see 45° to both sides, up and down. This means a slope of one. The planes bounding that volume can be defined as x<z, x>-z, y<z and y>-z. To a programmer with an eye for this, it means that it easy to «clip», i.e. remove obviously hidden polygons, with these bounds. All polygons that do not satisfy either of these inequalities can be

discarded. Note that these polygons must fall totally outside the bounding volume. This is called trivial rejection.

For polygons that are partially inside, partially outside that region, you may or may not want to clip (I am not sure yet of which is more efficient - I suspect it depends on the situation, i.e. number of polygons, number of edges in each polygon...)

There is also a slight problem with polygons that are partially behind, partially in front of the eye. Let us give a little proof. All of our polygon-drawing schemes are based on the assumption that a projected line is also a line. Here is the proof of that.

Let us consider the arbitrary line L, not parallel to the plane z=0, defined parametrically as:

$$L: \quad x = at+b, \quad y = ct+d, \quad z = t$$

Now, let us define the projection P(x,y,z) as follows:

$$P(x,y,z): \quad u=x/z, \quad v=y/z$$

Then, P(L) becomes

$$P(L): \quad u=(at+b)/t, \quad v=(ct+d)/t$$

We are using a segment here, not an infinite line. If no point with t=0 needs to be projected, we can simplify:

$$P(L): \quad u=a+b/t, \quad v=c+d/t$$

Let us measure the slope of that projection:

$du/dv=(du/dt) / (dv/dt) = (-b/t2)/(-d/t2)$

Since t is not 0

$du/dv=b/d$, a constant, thus the parametric curve defined by P(L) is a straight line if t is not zero.

However, if t is null, the result is indeterminate. We can try to find the limits, which are pretty obvious. When t tends towards zero+, then u will tend towards positive (or negative if b is negative) infinity. If t tends towards zero-, then u will tend towards negative (or positive, if b is negative) infinity. Thus, the line escapes to one infinity and comes back from the other if t can get a zero value. This means that if the segment crosses the plane z=0, the projection is not a line per se.

However, if the line is parallel to z=0, L can be defined as follows:

$$x=at+b$$

L: y=ct+d
z=k

Where k is the plane that contains L. P(L) becomes

P(L): u=(at+b)/k
v=(ct+d)/k

The equations do not admit a null k. For a non-null k, we can find the slope:

du/dv = (du/dt)/(dv/dt) = (a/k)/(c/k)
du/dv=a/c

Still a constant, thus P(L) is a line. However, if k is null, we do not get a projection. As k gets close to zero, we find that the projection lies entirely at infinity, except for the special case b=d=0, which is an indetermination.

This will have a practical impact on our 3d models. If a cube is centered on the eye, it will be distorted unless we somehow take that into consideration. The best way to do it, with little compromise and efficient coding, is to remove all parts of polygons that lie in the volume defined by z<=0. Of course, if an edge spans both sides of the plane z=0, you will have to cut it at z=0, excluding everything defined by z<=0. Thus, you will have to make sure the point of the edge with the smaller z coordinate is strictly greater than 0. Normally, you would have to take the next higher real number (whatever that might be!) but due to hardware contraints, we will use any arbitrarily small number, such as z=.001 or even z=.1 if you are building a real-time something in which infinite precision is not required.

Lastly, you might not want to render objects that are very far away, as they may have very little impact on the global scene, or because you are using a lookup table for logarithms and you want to make sure that all values are within range. Thus, you will set an arbitrary maximum z for all objects. It might even be a fuzzy maximum z, such as «I don't wand any object to be centered past z0». Thus, some objects may have some part farther than z0, but that may be more efficient than the actual clipping of the objects.

Moreover, you might want to make several polygon representations of the objects, with different levels of details as measured by the number of polygons, edges and vertices, and show the less detailed ones at greater distances. Obviously, you would not normally need to rasterize 10 polygons for an object that will occupy a single pixel, unless you are doing precision rendering. (But then, in that case, you'll have to change the whole algorithm...)

You may want to perform your trivial rejection clipping quickly by first rotating the handle coordinates of each polyehdra (the handle coordinate is the coordinate of its local coordinates' origin), and performing trivial rejection on them. As an example, if you determine that all of a polyhedra is contained within a sphere of radius r, and that this polyhedra is «centered» at z=-3r, then it is obvious that it is entirely outside the view volume.

Normally, the view volume is exactly one sixth of the total viewable volume (if you do not take into account the bottom of the pyramid). Thus, you will statistically eliminate five sixths of all displayable polyhedras with such a clipping algorithm.

Let's make a brief review of our speed improvements over the brute-force-transform-then-see-no-think algorithm. We multiplied the speed by six with our structure of pointers to edges that point to vertexes. Then, we multiply again by two (on average) with back-face culling, and now by six again. The overall improvement is a factor of 72! This shows you how a little thinking pays. Note however that that last number is a bit skewed because some operations do not take as long as some others, and also by the fact that most polyhedra do not contain a nearly infinite amount of vertices.

We have not taken into account several optimizations we mentioned, such as the lessening of detail, a bottom to the view volume and we did not compare with other kinds of algorithm. But another appeal of this algorithm is that it draws once, and only once to each pixel, thus allowing us to superimpose several polygons with minimal performance loss, or make complicated calculations for each pixel knowing that we will not do it twice (or more!) for some pixels. Thus, this algorithm lends itself very well to texture mapping (or more properly, pixmap mapping), shading or others.

However, we have to recognize that this approach is at best an approximation. Normally, we would have to fire projection rays over the entire area covered by a pixel (an infinite number of rays), which will usually amount to an integral, and we did not perform any kind of anti-aliasing (more on that later), nor did we filter our signal, nor did we take curved or interpenetrating surfaces into account, etc... But it lends itself very well to real-time applications.

8 A few more goodies...

Anti-aliasing is something we invented to let us decrease the contrast between very different pixels. More properly, it should be considered an approximation of high frequencies in a continuous image through intensity variations in the discrete view space. Thus, to reduce the stairway effect in a diagonal line, we might want to put some gray pixels.

µ §
Figure 7

Figure 7 demonstrates this. This diagonal line a was «anti-aliased» manually (I hear the scientists scream) in b, but it should let you understand the intention behind anti-aliasing.

Scientists use filters, and so will we, but I will not discuss fully the theory of filters. Let us see it that way. We first generate a normal image, without any kind of anti-aliasing. Then we want to anti-aliase it. What we do is take each pixel, modify it so it resembles its non-anti-aliased neighbors a little more and show that on screen.

Let us further imagine that how we want to do that specifically is to take one eight of each of the four surrounding pixel (north, south, east and west) and one half of the «real» pixel, and use the resulting

color for the output pixel.

If our image is generated using a 256-color palette, there comes a handy little trick. Let us imagine that we want to take one half of color a and one half of color b and add them together. Well, there are 256 possible values for a. And for each single value of a, there are 256 values of b, for a total number of 256x256 pairs, or 64k pairs. We can precalculate the result of mixing one half a with one half b and store it in a matrix 64kb large.

Now let us call the four surrounding pixels n,s,e,w and the central point o. If we use the above matrix to mix n and s, we get a color A that is one half n plus one half s, that is A=n/2+s/2. Then, mix e and w as B=e/2+w/2. Then, mix A and B. C=A/2+B/2=n/4+s/4+e/4+w/4. Lastly, mix C and o, and we get D=o/2+n/8+s/8+e/8+w/8, the exact desired result. Now, this was all done through a lookup table, thus was very fast. It can actually be done in real time.

Thus, if you want to do this, you will first create the original, un-anti-aliased image in a memory buffer, and then copy that buffer as you anti-aliase it on screen. Obviously, it will be slower a little, but you might want to consider the increased picture quality. However, you will probably want to do this in assembler, because it can be highly optimized that way (most compilers get completely confused...)

Another very nice technique is this. For pixel (u,v) on screen, mix equal parts of pixels (u,v), (u+1,v), (u,v+1), (u+1,v+1) in your rendering buffer. It's as if the screen was half a pixel of center in relation to the rendering buffer. You'll need to render an extra row and column though.

It is also interesting to note that the silhouette of a projected sphere is a circle. The centre of the circle is the projection of the centre of the sphere. However, the radius is NOT the projection of the radius of the sphere. However, when the distance is large enough when compared to the radius of the sphere, this approximation is good enough.

To determine if a point is contained inside a polyhedra, we extend our test from the polygons like this: if a line drawn from the point to infinity crosses an odd number of faces, the it is contained, otherwise it is not.

If you want to approximate a continuous spectrum with 256 colors (it will not look perfect though), use base 6. That is, using the RGB model (we could use HSV or any other), you will say that red, green and blue can all take 6 different values, for a total number of combinations of 216. If you add an extra value to green or red, you get 252 colors, close to the maximum. However, 216 is not bad and it leaves you 40 colors for system purposes, e.g. palette animations, specific color requirements for, let's say, fonts and stuff. Under Microsoft Windows, 20 colors in the palette are already reserved by the system, but you still have enough room for your 216 colors. By the way, I think it's stupid that MS Windows

does not allow you to do that as a standard palette allocation. You have to define your own video driver to enable programs that limit themselves to dithering to use those colors. And that's living hell.

In addition, if the colors are arranged in a logical way, it will be easier to find the closest match of the mix of two colors as per anti-aliasing, above. (A bit like cheaper 24 bit color.)

Some people give 8 values to red and green, and 4 values to blue, for a total of 256 colors, arguing that shades of blue are perceived more difficultly. Personnally, I think the result is not as good as the base six model I previously described.

Some other nice techniques: divide your palette in spans. Example, allocate palette registers 16-48 to shades of blue. Use only registers 20-44 in your drawings, but arrange so that it goes from white-blue at color #16 through black-blue at color #48. Then, shading is a simple matter of adding something to make it darker, or subtracting to make it lighter. Make a few spans for your essential colors, and maybe keep 30 for anti-aliasing colors. These should not be used in your pixmaps. They could be used only when performing final anti-aliasing of the rendered picture. Pick these colors so to minimize visual artifacts. (E.g. you have a blue span at 16-48, and a red span at 64-100. But you don't have any purple, so if blue happens to sit next to red, anti-aliasing will be hard to achieve. Then, just a bunch of your special 30 colors for shades of purple.) Another particular trick to increase the apparent number of colors (or rather, decrease the artifacts created by too few colors), you could restrict your intensity to lower values. E.g. make your darkest black/gray at 25% luminosity, and your lightest white at 75%. This will make your graphics look washed out, but your 256 colors will be spread over a narrower spectrum.

---

9 Sorting

Sorting becomes an important issue because we do it a lot in our 3d rendering process. We will discuss here briefly two general sorting algoritmthms.

The first sort we will discuss is the bubble sort. It is called this way because the elements seem to «float» into position as the sort progresses, with «lighter» elements floating up and «heavier» ones sinking down. Given the set S, composed of elements s0, s1, ..., sn-1. This set has n elements. Let us also suppose that we can compare two elements to know which of the two is «lighter», i.e. which one goes first. Let us also suppose that we can exchange two consecutive elements in the set S. To sort the set S, we will proceed as follow. Starting at element 0 up to element n-2, we will compare each element to its successor and switch if necessary. If no switches are made, the set is sorted. Otherwise, repeat the process. In pseudocode, we have:

repeat the following:
    for i varying from 0 to n-2, repeat the following
        if si should go after si+1, then exchange si and si+1
    end of the loop
until no elements are switched in the for loop (i.e the set is sorted)

This algorithm, while very inefficient, is very easy to implement, and may be sufficient for relatively small sets (n of about 15 or less). But we need a better algoritm for greater values of n.

However, we have to keep in mind that we are sorting values that are bounded; that is, these values have a finite range, and it is indeed small, probably 16 bits or 32 bits. This will be of use later. But let us introduce first the sort. It is called the radix sort.

Assume we are give a set S of numbers, all ranging from 0 to 9999 inclusively (finite range). If we can sort all si according to the number in the ones' place, then in the tens' place, then in the hundreds' place, then thousands' space, we will get a sorted list. To do this, we will make a stack for each of the 10 possible values a digit can take, then take each number in the S set and insert them in the appropriate stack, first for the rightmost digit. Then, we will regroup the 10 stacks and start over again for the second rightmost digit, etc... This can be done in pseudocode as follows:

divisor=1
while divisor<10000, do the following:
    for i varying from 0 to n-1, do the following:
        digit=(si divided (integer version) by divisor) modulo 10
        put si on stack # digit
    end of for loop
    empty set S
    for i varying from 0 to 9, do the following:
        take stack # i and add it to set S
    end of for loop
    divisor=divisor x 10
end of while loop

Now, let us assume that we want to work with hexadecimal numbers. Thus, we will need 16 stacks. In real life, what we want to sort is edges or something like that. If we limit ourselves to 16384 edges (16k, 12 bits), then our stacks need to be able to fit 16384 objects each. We could implement them as arrays and use lots of memory. A better way is to use lists.

One could also use the quicksort algorithm, which has a complexity of O(nxln(n)), which seems worse than n for this particular problem. It is also a bit more tedious to code (in my humble opinion). In our problem, we may not even need to sort for 16 bits. The only place that we need a powerful sorting algorithm is when sorting all edges from top to bottom and from left to right. If we are certain that all coordinates are bound between, say, 0 and 200 (e.g. for y coordinate), then we can sort on 8 bits. For the x coordinate, though, depending if we already clipped or not, we may have to sort on the full 16

bits. When using the z-buffer algorithm (decribed scantily elsewhere), you want to do a z-sort, then drawing the polygons that are on top FIRST so that you can avoid these nasty pixel writes.

---

10 Depth-field rendering and the Z-buffer algorithm

You can approximate any 3d function z=f(x,y) with a depth field, that is, a 2d matrix containing the z-values of f(x,y). Thus, z=M[x][y]. Of course, if you want any precision at all, it has somewhat large memory requirements. However, rendering this depth-field is fairly efficient and can give very interesting results for smooth, round surfaces (like a hill range or something).

To render a depth field, find the vector V from our eye to the pixel in the projection plane in 3space. Shoot a ray, as before, using that vector. Start at the eye and move in direction V a little. If you find that you are still above the surface (depth field), then keep going (add another V and so on) until you hit or fall under the surface. When that happens, color that pixel with the color of the surface. The color of the surface can be stored in another matrix, C[x][y]. In pseudocode, we get:

```
M[x][y]=depth field
C[x][y]=color field
for each pixel to be rendered, do:
        V=projection ray vector
        P=Eye coordinates
        while (M[Px][Py]<Pz)
                P=P+V
        end while
        color the current pixel with C[Px][Py]
        if we are using z-buffer algorithm, then set Z[Px][Py] to Pz (see below)
end for
```

Of course, you will want to break the loop when P is too far from origin (i.e. when you see that nothing is close through that projection ray, stop checking).

For vector graphics, we were forced to place our eye centered on origin and looking towards positive z values. That forced us to rotate everything else so the eye would be positioned correctly. Now, obviously, we cannot rotate the whole depth field as it would be very time- and memory-consuming. Instead, the eye now can be centered anywhere (the initial value for P need not be (0,0,0)) and the vector V can be rotated into place. Even better, if we can find the vector W from the eye to the center of the projection plane, and the two base vectors u and v in 3d for the projection plane, all of the other vectors Vu,v wil be a linear combination of the form: V=W+au+bv where (a,b) is the coordinate of the pixel in the (u,v) coodinates system. If we are doing this rendering with scan lines (as we most probably are doing), then the above equation can be done incrementally, e.g. say Va,y=A, then Va+1,y=A+u. No multiplications whatsoever.

The z coordinate of each pixel is readily available in the above algorithm and it has a use. Let us store all the z coordinate values in a matrix Z[a][b]. If we then want to add a flat image with constant z value (say z=k) to the already rendered scene, we can do it very easily by writing only to pixels for which Z[a][b]>k. Thus, we can use another rendering technique along with depth-field rendering. This is called the z-buffer algorithm. Accelerating calculations for the z-buffer with polygons, you might want to remove the divide-per-pixel. That is fully discussed towards the end of the texture-mapping chapter.

A few optimizations are easily implemented. Let us assume we are drawing from the bottom of the screen to the top, going in columns instead of rows. First, if you are not banked (or banked less than the steepest slope, e.g. no «overhangs» are seen in the picture) and you have rendered pixel a. Then you know how far pixel a is from you. Then, then next pixel up, pixel b, will be at least as far as pixel a. Thus, you do not need to fire a ray starting from your eye to pixel b, you need only to start as far as pixel a was. In fact, if you are not banked, you need only to move your Pz coordinate up a little and keep going.

Therefore, if pixel a is determined to be a background pixel, so will all the pixels above it.

You might want to render all of you polygons and other entities using a z-buffer algorithm. As an example, say you have 3 polygons, A, B and C, and a sphere S. You could tell your renderer to render poly A, which it would do right away, filling the z-buffer at the same time. That is, for each screen pixel that the buffer A covers, it would calculate the z-value of the polygon in that pixel, then check with the z-buffer if it stands in front of whatever is already drawn. If so, it would draw the pixel and store the polygon's z-value for that pixel in the z-buffer. Then render sphere S, which it would do right away again, calculating the z-value for each pixel and displying only the visible pixels using the z-buffer, and updating the z-buffer accordingly. Lastly polygons B and C would be displayed.

What you gain by using this algorithm instead of a scan line algorithm is obvious. Interpenetrating polygons behave correctly, mixing rendering techniques is straightforward. There is but one annoying thing: calculating the z value for a polygon in any point on the screen involves a division, hence a division per pixel if using z-buffer. But we can do better.

We are not really interested in the z value of a polygon in a determined pixel, what we want really is to know what is visible. To that purpose, we can obviously calculate z and show the object with the smallest z value. Or we could evaluate z^2 and show the one with the smallest z^2 value. If we want whatever has the smallest z value, this is the same as saying that we want whatever has the largest 1/z value. Fortunately 1/z is way easier to calculate. Let the plane equation be Ax+By+Cz=D, and the projection equations be u=x/z and v=y/z, or

$Ax+By+Cz=D$
$x=uz$
$y=vz$

or

$A(uz)+B(vz+)Cz=D$

or

$z(Au+Bv+C)=D$

$(Au+Bv+C)/D=1/z$

$1/z= (A/D)u+(B/D)v+(C/D)$

Let $M=A/D$ (a constant), $N=B/D$ (another constant) and $K=C/D$ (a third constant), then

$1/z=Mu+Nv+K$

As we can see plainly, we don't have a divide. This is linear so can be calculated incrementally. E.g. for a given scanline ($v=k$), the equation is

$1/z=Mu+Nk+K$

Let $P=Nk+K$ (a constant)

$1/z=Mu+P$

A linear equation. When going from pixel u to pixel u+1, we just need to add M to the value of 1/z. A single add per pixel.

There's also a double edged optimization that you can make, which follows.

Sort your polygons in generally increasing z order (of course, some polygons will have an overlap in z, but that's everybody's problem with this algorithm - your solution is just as good as anybody else's.) Now, you're scan-converting a polygon. If you can find a span (e.g. a part of the current scanline) for which both ends are behind the same convex object, then that span is wholly hidden. As an example of this, if both enpoints of the current span (on the scanline) are behind the same convex object, then the current span is entirely hidden by that convex object. If all of your objects are convex polygons, then you don't have to check for convexity of the object. Another interesting example is this:

```
If current span's left endpoint is hidden
        let A be the object that hides the left endpoint (of the span)
        if A hides the right endpoint
                stop drawing the scanline
        else
```

```
                start drawing from the right end of the span, leftwards, until object A is in front of the
span
        end if
else
        if current span's right endpoint is hidden
                let B be the object that hides the right endpoint
                scan rightwards until object B is in front of the span
        else
                draw span normally
        end if
end if
```

This can be applied also to polyhedras if they are convex. This last optimization should be relatively benign (e.g. I expect a factor of two or some other constant from it).

---

11 Bitmap scaling and mixing rendering techniques

Another popular technique is called bitmap (or more properly pixmap) scaling. Say you are looking in a parallel direction to the z axis, and you are looking at a picture with constant z. Through the projection, you will realize that all that happens to it is that it is scaled by a constant factor of 1/z. That is, if you look at a photograph lying in the plane z=2, it will look exactly the same (once projected) as the same photograph at z=1, except that it will be twice as small. You could also do that with the picture of a tree. From a great distance, the approximation will be fairly good. When you get closer, you might realize that the tree looks flat as you move. But nonetheless, it can yield impressive results. Bitmap scaling often suffices for ball-like objects. You could use it for an explosion, smoke, spherical projectile, etc... The advantage of bitmap scaling is that it has only one point, probably it's center, to transform (i.e, translate and rotate into place in 3d), and scaling it is very fast.

The advantages of such a technique are obvious. First, it is very easy to scale a pixmap by a given factor. Second, the z value is a constant, thus it can easily be incorporated in the above z-buffer algorithm with depth-field rendering for very nice results. However, it will be obvious to the careful observer that you can not smoothly go around it, as it is always facing you the same way (i.e. you cannot see the back door to a house if it is rendered this way, because, obviously, the bitmap being scaled does not have a back door).

Moreover, the depth-field rendering technique blends quite smoothly with vector graphics. First, render the depth field in a buffer B. Do a scan-line algorithm for the vector graphics, but with an added twist: the «background» bitmap is B, and use the z-buffer algorithm to determine if a pixel in B lies in front of the current polygon if any.

If you do decide to use depth-field rendering along with vector graphics, and you decide not to bank your eye, then you can eliminate one full rotation from all calculations. E.g. if your eye is not banked, you do not need to rotate around the z axis.

Scaling a bitmap can be done fairly easily this way. The transformation is like this: we want to take every (u,v) point and send it into (u/s,v/s), where s is the scaling factor (s=2 means that we are halving the size of the bitmap). The process is linear.

(i,j)=(u/s,v/s)
(is,js)=(u,v)

Thus, if we are at pixel (i,j) on screen, it corresponds to pixel (is,js) in the bitmap. This can also be done incrementally. If un=A, then un+1=A+s. No multiplications are involved. Thus, if we are viewing a bitmap at z=2 according to the above said 3d projections, then we can render it using a scan-line algorithm with these equations and s=z=2.

If a bitmap has holes in it, we can precalculate continuous spans in it; e.g., in an O, drawing a scanline through the center shows you that you are drawing two series of pixels. If you want the center to be transparent (allow a background to show through), then you can divide each scanline of the O in two spans, one left and one right, and render them as two separate bitmaps; e.g., break the O in (and) parts, both of which have no holes. Note however that we have a multiplication at each beggining of a span, so this might not be efficient. Or you could select a color number as being the «transparent» color, and do a compare per pixel and not blit if it is the transparent color. Or you can allocate an extra bitplane (a field of bit, one bit per pixel) where each bit is either 1 for opaque (draw this pixel) or 0 for transparent (do not draw this pixel).

Partially translucid primitives can be rendered pretty easily. This is an old trick, you see. You can even scavenge the lookup table you used for anti-aliasing. If you want to have a red glass on a part of the screen, the lookup table will tell you how to redden a color. That is, the item #c in the lookup table tells you what color is color c with a little red added. The lookup table need not be calculated on the fly, of course. It can be calculated as part of your initialization or whatever.

---

12 About automatically generating correctly oriented normals to planes.

This bit is not meant to be a blindingly fast algorithm for region detection. It is assumed that these calculations are made before we start generating pictures. Our objective is to generate a normal for each plane in a polyhedra oriented so that it points outwards respectively to the surface. We assume all surfaces are defined using polygons. But before determining normals for planes, we will need to examine polygons in 3d.

Each polygon being constituted of a set of edges, we will later require a vector for each edge that points in the polygon for any edge. See below.

µ §
Figure 8

In figure 8, we can see in red the said vectors for all edges. As we can see, the vectors do not necessarely point towards the general center of mass. We could use the algorithm we used for rendering polygons in two dimensions to determine the direction of the arrows, but that is a bit complicated and involves trying to find a direction that is not parallel to any of the above lines. Thus, we will try another approach.

First, pick any point in the polygon. Then, from that point, take the farthest point from it. See below:

µ §
Figure 9

(Please, do not measure it with a ruler :-))
Now say we started with the point marked by the big red square. Now, both point a and b are the farthest points from the red square. You can pick any of the two. The objective here is to find three consecutive points around the polygon for which the inside angle is less than 180 degrees. If all three are on the circle (worst case), their angle is strictly smaller than 180 degrees (unless they are all the same point, silly).

You could also pick the topmost point. If several are at the same height, use the leftmost (or rightmost, whatever) one in the topmost points.

Now we are certain that the small angle is the inside angle. Let us draw the general situation:

µ §
Figure 10

Now, in a we see the above mentionned farthest point and the two adjacent edges. If we take these edges as vectors as in b and then sum them as in c, we find a point (pointed by the red vector in c) which is necessarely on the in side for both edges. Now it is easy to find a perpendicular vector for both edges and make sure that they are in the good general direction. A way would be to take the perpendicular vector and do a scalar multiplication with the above red vector, the result of which must be strictly positive. If it is not, the you must take the opposite vector for perpendicular vector.

Now we know the situation for at least one edge. We need some way to deduce the other edges' status from that edge's.

If we take Figure 10.c as our general case, the point at the end of the red vector is not necessarely on the in side for both edges. However, if it is on the in side for any of the two edges, it is on the in side for both, and if it is on the out side for any edge, it is on the out side for both.

Normally, we will never find two colinear edges in a polygon, but if we do, it is fairly easy to solve it anyway (left as an exercise).

Now, briefly, here is how to determine if a normal to a plane points in the right direction. Pick the point with the greatest z value (or x, or y). If many points have the same z value, break ties with greatest x value, then greatest y value. Then, take all connected edges. Make them into vectors, pointing from the above point to the other endpoints (that is, the z component is less than or equal to zero). Make them unit vectors, that is, length of one. Find the vector with the most positive (or less negative) z value. Break ties with the greates x value, and then greatest y value. Take its associated edge. Now that edge is said to be outermost.

Now, from that outermost edge, take the two associated polygons. The edge has an associated vector for each connected polygon to indicate the "in" direction. Starting from any of the two points of the edge, add both these vectors. The resulting point is inside. Adjust the normal so that it is on the right side of the plane for both polygons.

Now we have the normal for at least one plane straight. We will deduce correct side for the normal to the other polygons in a similar manner than we did for the edges of a polygon, above.

Take two adjacent polygons, one for which you do know the normal, the other for which you want to calculate the correct normal. Now take a shared edge (any edge will do). That edge has two "in" vectors for the two polygons. Starting from any of the two points of the edge, add the two «in» vectors. If the resulting point is on the «in» side for a plane, it is also on the «in» side of the other plane. If it is on the «out» side of a plane, it is on the «out» side for both planes.

To get a general picture of what I'm trying to say, here's a drawing. Basically, where two lines intersect, you get four distinct regions. One region lies entirely on the «in» side of both lines and another region lies entirely on the «out» side of both lines (if the two lines define a polygon). The two other regions are mixed region (they lie on the «in» side of a line and on the «out» side of the other line). The «mixed» regions may or may not be part of the actual polygon, depending on whether the angle is lesser or greater than 180 degrees.

µ §
Figure 11

So we are trying to find a point that is in the «in» or «out» region, but not in the «mixed» region. If, as above, we take the edges as vectors and add the together, we end up in the either the «in» region or the «out» region, but not in the «mixed» region. That can be proved very easily, geometrically (yet strictly), but I don't feel like drawing it. :) Anyway, it should be simple enough.

---

13 Reducing a polygon to a mesh of triangles

Now that's fairly simple. The simplest is when you have a convex polygon. Pick any vertex. Now, take its two adjacent vertexes. That's a triangle. In fact, that's your first triangle in your mesh of triangles. Remove it from your polygon. Now you have a smaller polygon. Repeat the same operation again until you are left with nothing. Example:

µ §

Figure 12

For the above polygon, we pick the green edges in 1. The triangle is shown in 2. When we remove it from the polygon, we're left with what we see in 3.

This will take a n sided polygon and transform into n-2 triangles.

If the polygon is concave, we can subdivide it until it is convex and here is how we do that.

Take any vertex where the polygon is concave (i.e. the angle at that vertex is more than 180 degrees) and call that vertex A. From that vertex, it can be proved geometrically that there exists another vertex that is not connected by an edge to which you can extend a line segment without intersecting any of the polygon's edges. In short, from vertex A, find another vertex, not immediately connected by an edge (there are two vertices connected to A by an edge, don't pick any one of them). Call that new vertex B. Make certain that AB does not intersect any of the other edges. If it does, pick another B. Once you're settled on your choice of B, split your polygon in two smaller polygons by inserting edge AB. Repeat the operation with the two smaller polygons until you either end up with triangles or end up with convex polygons which you can split as above. It can again be proved (though a little more difficultly) that you will end up with n-2 triangles if you had a n sided polygon.

Here's a tip on how to find a locally concave vertex.

µ §
Figure 13

[The arrows point towards the «in» side]

The two possible cases are shown in figure 13, either it is or it is not convex. In 1, it is convex, as in 2 it is concave. To determine if it is or not convex, take the line that passes through a and u, above. Now, if v stands on the «in» side of the au line, it is convex. Otherwise, it is concave.

---

14 Gouraud shading

First, let us discuss a simple way for shading polygons. Imagine we have a light source that's infinitely far. Imagine it's at z=positive infinity, x=0, y=0. If it's not, you can rotate everything so that it is (in practice, you'll only need to rotate normals). Now, the z component of the polygons normals give us a

clue as to the angle between the polygon and the light rays. The z component goes from -1, facing away from the light source (should be darkest) to 0, facing perpendicular to the light source (should be in half light/penumbra), to +1, facing the light source (should be the brightest). From that, you can shade it linearly or any way you want.

With this model, a polygon is uniformly shaded. Approximating round objects through polygons will always look rough.

Gouraud shading is what we call an interpolated shading. It is not physically correct, but it looks good. What it does is this: it makes the shade of a polygon change smoothly to match the shade of the adjacent polygon. This makes the polyhedra look smooth and curved. However, it does not change the actual edges to curves, thus the silhouette will remain unsmoothed. You can help that by allowing curved edges if you feel like it.

First, we will interpolate a vertex normal for all vertexes. That is, we will give each vertex a «normal» (note that a point does not have a normal, so you could call it pseudo-normal). To interpolate that vertex «normal», averaging all connected polygons normals would be a way. Now, find the «shade» for each vertex.

µ §
Figure 14

Now, say point a is at (ax,ay) and point b is at (bx,by). We're scan-converting the polygon, and the gray-and-red line represents the current sanline, y0. Point P is the pixel we are currently drawing, located at (x0,y0). Points m and n define the span we're currently drawing. Now, we will interpolate the color for point m using a and b, and the color for n using a and c, then interpolate the color for P using the color for m and n. Our interpolation will be linear. Say color for point a is Ca, color for point b is Cb, for point c it is Cc, Cm for m, Cn for n and CP for P.

We will say that color at point m, Cm, is as follows:

Cm=(y0-ay)/(by-ay) x (Cb-Ca)+Ca

That is, is point m is halfway between a and b, we'll use half ca and half cb. If it's two-third the way towards b, we'll use 1/3Ca and 2/3Cb. You get the picture. Same for n:

Cn=(y0-ay)/(cy-ay) x (Cc-Ca)+Ca.

Then we do the same for CP.

CP=(x0-mx)/(nx-mx) x (Cn-Cm)+Cn.

If you think a little, these calculations are exactly the same as Bresenham's line drawing algorithm, seen previously in chapter 2.1. It is thus possible to do them incrementally. Example:

Say we start with the topmost scanline. Color for point m is at first Cb. Then, it will change linearly. When point m reaches point a, it will be color Ca. Now, say dy=by-ay, dC=Cb-Ca, and u=y0-ay.

Cm=dC/dy x u+Ca.

Then, when y0 increases by 1, u increases by one and we get

Cm'=dC/dy x (u+1)+Ca=dC/dy x u+Ca + dC/dy
Cm'=Cm+dC/dy

Same as Bresenham. Thus, when going from one scanline to the next, we simply need to add dC/dy to Cm, no multiplications or divisions are involved. The same goes for Cn. CP is done incrementally from pixel to pixel.

---

15 Clipping polygons to planes

Eventually, you might need to clip your polygon, minimally to the z>0 volume. Several approaches can be used. We will discuss here the Sutherlan-Hodgman algorithm. But first, let us speak of trivial rejection/acceptation.

If a polygon does not intersect the clipping plane, it can be either trivially accepted or rejected. For example, if every single point in a polygon are in z>0 and we are clipping with z>0, then the whole polygon can be accepted. If every single point is in z<=0, the whole polygon can be trivially rejected. The real problem comes with cases where some points are in z>0 and some are in z<=0. Another nifty thing you can do is pre-calculate the bounding sphere of a poly with its center on a given point in the poly. Let O be the sphere's center and R it's radius. If Oz-R>0, then the poly can be trivially accepted even faster (no need to check every single point). If Oz+R<=0, the poly can be trivially refused. You can extend this to polyhedras. You could also check wether the intersection of the sphere and the polygon plane is in z>0 (or z<=0), which might be slightly better than checking for the whole sphere.

Here comes the Sutherland-Hodgman algorithm. Start with a point that you know is going to be accepted. Now, move clockwise (or counter-clockwise) around the poly, accepting all edges that should be trivially accepted. Now, when an edge crosses from acceptance to rejection (AtoR), find the intersection with the clipping plane and replace the offending edge by a shortened one that can be trivially accepted. Then, keep moving until you find an edge that crosses from rejection to acceptance (RtoA), clip the edge and keep the acceptable half. Add an edge between the two edges AtoR and

RtoA. Keep going until you are done.

µ §
Figure 15

Figure 15 illustrates the process of clipping the abcde polygon to the clipping plane. Of note is this: when performing this kind of clipping on convex polygons, the results are clear. Furthermore, a convex polygon always produce a convex polygon when clipped. However, concave polygons introduce the issue of degenerate edges and whether they should be there in a correctly clipped polygon. Figure 16 shows such a case of degenerate edge generation.

µ §
Figure 16

In figure 16, the polygon to the left is clipped to the plane shown, the result is on the right. The bold edge is double. That is, two edges pass there. This is what we call a degenerate edge. Degenerate edges don't matter if what interests us is the area of the polygon, or if they happen to fall outside of the view volume. What interests us is indeed the area of the polygon, but due to roundoff error, we could end up with the faint trace of an edge on screen. On could eliminate these degenerate edges pretty easily.

To do so, do not add an edge between the points where you clip at first (edge xy in figure 15). Instead, once you know all clipping points (x and y in the above example), sort them in increasing or decreasing order according to one of the coordinate (e.g. you could sort the clipping point in increasing x). Then, between first and second point, add an edge, between third and fourth, add an edge, between fifth and sixth, add an edge and so on. This is based on the same idea that we used for our polygon drawing algorithm in an earlier chapter. That is, when you intersect an edge, you either come out of the polygon or go in the polygon. Then, between the first and second clipped point belongs an edge, etc...

However, since we are clipping to z=0, the degenerate edge has to be outside the view volume. (Division by a very small z, when making projections, ensures that the projected edge is far from the view window). Since we are clipping to z>0, let us pull out our limit mathematics.

Let us first examine the case where the edge lies entirely in the z=0 plane.

Let us assume we are displaying our polygon using a horizontal scan-line algorithm. Now, if the both of the offending edge's endpoints are at y>0, then the projected edge will end up far up of the view window. Same goes if both endpoints are at y>0. If they are on either side of y=0, then all projected points will end up above or below the view window, except the exact point on the edge where y=0. If that point's x<0, then x/z will end up far to the left, and if x>0, the point will end up far to the right. If x=0, then we have the case x=y=0 and z tends towards positive zero (don't flame me for that choice of words). In the case x=0, x/z will be zero (because 0/a for any nonzero a is 0) and so for y. Thus, the projected edge should pass through the center of the view window, and go to infinity in both directions. It's slope should be the same as the one it has in the z=0 plane. Just to make it simple, if we have the plane equation (as we should) in the form Ax+By+Cz+D=0, we can find the projected slope by fixing z=1: Ax+By+C+D=0, or y=-A/Bx-(C+D). Thus the slope of the projected line is -A/B. If B is zero, we have a vertical line.

What we conclude is that if the edge's z coordinate is 0 and both of its endpoints are on the same side

of plane y=0, then the edge will not intercept any horizontal scanline and can thus be fully ignored. If the endpoints are on either side of y=0, then it will have an intersection with all horizontal scanlines. The projected edge will end up left if, when evaluated at y=0 and z=0 its x coordinate is less than 0 (from the plane equation, x=-D/A). If the x coordinate is positive, the edge ends up far to the right. If x=y=0 when z=0, the edge will be projected to a line passing through the center of the screen with a slope of -A/B.

In the case where one of the edge's endpoints is in the z=0 plane, that endpoint will be projected to infinity and the other will not, we will end up with a half line in the general case, sometimes a line segment.

Let us name the endpoint whose z coordinate is 0 P0. If P0's x<0, then it will be projected to the left of the window. If P0's x>0, it will be projected to the right. Likewise, if P0's y<0, it will be projected upwards, if y>0 it will be projected downwards.

The slope will be (v0-v1)/(u0-u1) where (u0,v0) is projected P0 and (u1,v1) is projected P1. This can be written as
u0=P0x/p0z u1=P1x/P1z v0=P0y/P0z v1=P1y/P1z

It is of note that u1 and v1 are both relatively small numbers compared to u0 and v0. The slope is therefore:

m=(P0y/P0z-P1y/P1z)/(P0x/P0z-P1x/P1z)
m=(P0y/P0z)/(P0x/P0z-P1x/P1z)-(P1y/P1z)/(P0x/P0z-P1x/P1z)
m=(P0y/P0z)/([P0xP1z-P1xP0z]/P0zP1z)-0
m=(P0y/P0z)(P0zP1z/[P0xP1z-0])
m=P0yP1z/(P0xP1z)
m=P0y/P0x.

By simmetry we could have found the inverse slope p=P0x/P0y. Thus, in the case where P0x=0, we are facing a vertical projection. Else, the projection's slope is P0y/P0x. Anyway, the line passes through (u1,v1).

In the case where P0=0, (u0,v0)=(0,0). Then the projected edge remains a line segment.

Oftentimes, us graphics programmer want to interpolate stuff, especially when the calculations are very complex. For example, the texture mapping equations involve at least two divides per pixel, which can be just a bit too much for small machines. Most of the time, we have a scanline v=k on screen that we are displaying, and we want to display a span of pixels, e.g. a scanline in a triangle or something. The calculations for these pixels can sometimes be complicated. If, for example, we have to draw pixels in the span from u=2 to u=16 (a 14 pixel wide span) for a given scanline, we would appreciate reducing the per-pixel calculations to the minimum.

Let's take for our example, the z value of a plane in a given pixel. Let's say the plane equation is x+2y+z=1. We have already seen that z can be calculated with z=D/(Au+Bv+C), or z=1/(u+2v+1). Let us imagine that we are rendering a polygon, and that the current scanline is v=1. Then, z=1/(u+3) for the current scanline. If that a span goes from u=0 to u=20 (that is, on line v=.5, the polygon covers pixels u=0 through u=20). We can see form the equation for z that when u=0, z=1/3, and when u=20, z=1/23.

Calculating the other values is a bit complex because of the division. (Divisions on computers are typically slower than other operations.) As of such, we could perhaps use an approximation of z instead. The most naive one would be the following: just assume z is really of the form z=mu+b and passes through the points u=0, z=1/3 and u=20, z=1/23. Thus:

z=mu+b verifies
1/3=m(0)+b and
1/23=m(20)+b

thus

1/3=b
and
1/23=20m+b
or
1/23=20m+1/3
1/23-1/3=20m
m=1/460-1/60 or approximately -0.01449275362319.

Thus, instead of using z=1/(u+3), we could use z=-0.01449275362319u+1/3. In this particular case, this would be somewhat accurate because m is small.

Example, we know from the real equation that z=1/(u+3), thus when u=10, z=1/13, approximately 0.07692307692308. From the interpolated equation, when u=10, we get z=-0.01449275362319*10+1/3, or approximately 0.1884057971014. The absolute error is approximately 0.11, which may or may not be large according to the units used for z. However, the relative error is around 30%, which is quite high, and it is possible to create cases where the absolute error is quite drastic.

To reduce the error, we could use a higher degree polynomial. E.g. instead of using z=mx+b, we could

for instance use z=Ax^2+Bx+C, a second degree polynomial. We would expect this polynomial to reduce the error. The only problem with this and higher degree polynomials is to generate the coefficients (A, B and C in this case) which will minimize the error. Even defining what "minimizing the error" is can be troublesome, and depending on the exact definition, we will find different coefficients.

A nice way of "minimizing the error" is using a Taylor series. Since this is not a calculus document, I will not teach it in detail, but merely remind you of the sum. It can be demonstrated that all continuous functions can be expressed as a sum of polynomials for a given range. We can translate that function so as to center that range about any real, and such other things. In brief, the Taylor series of f(x) around a is:

T=f(a)+f'(a)(x-a)/1!+f''(a)(x-a)^2/2!+f'''(a)(x-a)^3/3!+f''''(a)(x-a)^4/4!+...

This may look a bit scary, so here is the form of the general term:

Ti=fi(a)(x-a)^i/i!

and

T=T0+T1+T2+T3+...+Ti+...

Where fi(a) is the ith derivative of f evaluated at point a, and i! is the factorial of i, or 1x2x3x4x....xi. As an example, the factorial of 4 is 1x2x3x4=24. It is possible to study this serie and determine for which values of x it converges and such things, and it is very interesting to note that for values of x for which it does not converge, it diverges badly usually.

Taylor series usually do a good job of converging relatively quickly in general. For particular cases, one can usually find better solutions than a Taylor series, but in the general case, they are quite handy. Saying that they converge "relatively quickly" means that you could evaluate, say, the first 10 terms and be quite confident that the error is small. There are exceptions and special cases, of course, but this is generally true.

What interests us in Taylor series is that they give us a very convenient way of generating a polynomial of any degree to approximate any given function for a certain range. In particular, the z=1/(mu+b) equation could be approximated with a Taylor series, and it's radius of convergence determined, etc... (Please look carefully at the taylor series and you will see that it is indeed a polynomial).

Here are a few well known and very used Taylor series expansions:

exp(x)=1+x+x^2/2+x^3/3!+x^4/4!+x^5/5!+x^6/6!+....
sin(x)=x-x^3/3!+x^5/5!-x^7/7!+x^9/9!-...
cos(x)=1-x^2/2!+x^4/4!-x^6/6!+x^8/8!-...

Taylor series for multivariable functions also exist, but they will not be discussed here. For a good description of these, you can get a good advanced calculus book. My personnal reference is Advanced engineering mathematics, seventh edition by Erwin Kreysig, ISBN 0-471-55380-8. As for derivatives,

limits et al, grab an introductory level calculus book.

Now the only problem is evalutating a polynomial at any point. As you know, evaluating Ax^2+Bx+C at point x0 requires that we square x0, multiply it by A, add B times x0 and then finally add C. If we're doing that once per pixel, we are losing a lot of time. However, the same idea as in incremental calculations can be recuperated and used for any degree of polynomial. Generally speaking, we're going to try to find what happens to f(x) as we increase x by one. Or, we're trying to find f(x+1)-f(x) to see the difference between one pixel and the next. Then, all we need to do is add that value as we move from one pixel to the next. Let's do an example with f(x)=mx+b.

f(x+1)-f(x)=[m(x+1)+b]-[mx+b]
f(x+1)-f(x)=m

So, as we move from a pixel to the next, we just need to add m to the previous value of f(x).

If we have a second order polynomial, the calculations are similar:

f(x)=Ax^2+Bx+C

f(x+1)-f(x)=[A(x+1)^2+B(x+1)+C]-[Ax^2+Bx+C]
=[A(x^2+2x+1)+Bx+B+C]-[Ax^2+Bx+C]
=[Ax^2+(2A+B)x+A+B+C]-[Ax^2+Bx+C]
=2Ax+A+B

Let's name that last equation g(x). So, as we move from x to x+1, we just need to add g(x) to the value of f(x). However, calculating g(x) involves two multiplications (one of which which can be optimized out by using bit shifts) and at least an add. But g(x) is a linear equation, so we can apply forward differencing to it again and calculate:

g(x+1)-g(x)=[2A(x+1)+A+B]-[2AX+A+B]
=2A

So what we can do as we move from x to x+1 is first add g(x) to f(x) and then add 2A to g(x), only two adds per pixel.

This can be extended to any degree of polynomial. In particular, NURBS (not described in this document) can be optimized this way. (I do not intend to discuss NURBS, but this optimization is considered less efficient that subdivision, but is worth mentionning.)

So what is the use for all this? Well, you could use a Taylor series to do texture mapping instead of two divisions per pixel. This is a generalization of linear approximation for texture mapping. You could also use it for evaluating the z value of a polygon in any pixel, as we were doing in the introduction to this chapter. This however might not be very useful since you might not need the actual z value, and the 1/z value might suffice which, as we have already seen, can be calculated incrementally without approximation (e.g. no error except roundoff). This would be useful in visible surface determination.

17 Specular highlights and the Phong illumination model

Up to now, we have been using a very straight illumination model. We have assumed that the "quantity of light at a given point" can be calculated with some form of dot product of the light vector with the normal vector. (Let me remind you that (a,b,c) dot (d,e,f) is ad+be+cf.) You should normalize the light vector for this, and the plane normal should be of unit length too. Let us assume the plane normal is (A,B,C) and that it is already normalized. Let us further assume that the light vector (vector from light source to point that is lighted in 3d) is (P,Q,R). Then, the normalized light vector is (P,Q,R) times 1/sqrt(P*P+Q*Q+R*R). As you can see, normalizing the light vector is very annoying. (See chapter 2 for a more lengthy discussion of related topics).

Now, we have been imagining that no matter what angle you look at the object from, the intensity of the light is the same. In reality, that may not be the case. If the object is somewhat shiny and a bright light shines on it and you are looking at the object from an appropriate angle, there should be a very intense spot. If you move your eye, the spot moves, and if you move your eye enough, the spot disappears entirely. (Try it with an apple in a room with a single, intense light source, no mirrors et al). This means that the light is reflected more in one general direction than in the others.

Let us look at a little drawing. Let N be the surface normal, L the light vector, R the reflection vector and V the eye-to-object vector.

µ §
Figure 17

In the above picture, we see clearly that R is L mirrored by N. Please not that the angle between L and R is <u>not</u> 90 degrees as it may appear, but rather twice the angle between L and N. It can be demonstrated that:

R=2N(N dot L)-L

Or, in more detail, if R=(A,B,C), N=(D,E,F) and L=(G,H,I), then N dot L is DG+EH+FI, and thus we have

A=2D(DG+EH+FI)-G
B=2E(DG+EH+FI)-H

C=2F(DG+EH+FI)-I

It is of note that, if the light is at infinity, for a flat polygon, N dot L is constant. If the light is not at infinity or if the surface is curved, then N dot L varies.

Now, if we do not take into account the angle b in figure 17, the amount of light perceived should be something like
I=Kcos(a)

where K is some constant that depends on the material and a is the angle shown in figure 17. If to that we want to add the specular highlight, we should add a term that depend on angle b of figure 17. I becomes

I=Kcos(a)+Ccos^n(b)

Where C is a constant that depends on the material. The constant n is a value that will specify how sharp is the reflection. The higher the value for n, the more localized the reflection will be. Lower values of n can be used for matte surfaces, and higher values can be used for more reflective surfaces. Most light sources don't shine as bright from a distance (but some do shine as bright at any small distance, such as the sun, but that light can be considered to lay "infinitely" far). In that case, we could want to divide the above intensity by some function of the distance. We know from physics that the quantity of energy at a distance r of a light source is inversely proportional to the square of the radius. However, this may not look good in computer graphics. In short, you might want to use the square of the length of the light vector, or the length of the light vector, or some other formula. Let us assume we are using the length of the light vector. Then, the intensity becomes:

I=1/|L| * [Kcos(a)+Ccos^n(b)]

Furthermore, it is known that if t is the angle between the vectors A and B, then A dot B is |A||B|cos(t). Then,

cos(t)=(A dot B)/(|A||B|)

If |A| and |B| are both one, we can ignore them. So, let L' be normalized L, and R' be normalized R and V' be normalized V, and of course N is of unit length, then the equation is:

I=D * (K[N dot L']+C[R' dot V'])

Where D=1/|L|. We could have used some other function of |V|.

To that equation above, you may want to add some ambiant light, e.g. light that shines from everywhere at the same time, such as when you are standing outside (you wouldn't see a very dark corner in daylight, normally, so the light that is still present in a "dark corner" is the ambiant light). If you have multiple light sources, just calculate the intensity from each light source and add them together.

This of course assumes that a is comprised between 0 and 90 degrees, otherwise the light is on the wrong side of the surface and doesn't affect it at all. (In which case the only light present would be the

light of intensity A).

---

18 Phong shading

Polygons are flat, and as such have a constant normal across their surface. When we used pseudo-normals for gouraud shading. This was because we were attempting to model a curved surface with flat polygons. We can use the Phong illumination model with flat polygons if we want. That is, forget about the pseudo-normals, use the plane normal everywhere in the Phong illumination equation. This will yield very nice looking flat surfaces, very realistic. However, most of the time, we want to model curved surfaces and the polygons are merely an approximation to that curved surface. In a curved surface, the normal would not be constant for a large area, it would change smoothly to reflect the curvature. In our polygons, of course, the normal does not change. However, if we really want to model curved surfaces with polygons, we find that the sharp contrasts between facets is visually disturbing and does not represent a curved surface well enough. To this end, we made the intensity change gradually with Gouraud shading. There is however a problem with Gouraud shading.

It is possible to demonstrate that no light inside a Gouraud shaded polygon can be of higher intensity than any of the vertices. As of such, if a highlight were to fall inside the polygon (as is most probably the case) and not on a vertex, we would miss it perhaps entierly. To this end, we might want to interpolate the polygon normal instead of intensity from the vertices pseudo-normals. The idea is the same as with Gouraud shading, except that instead of calculating a pseudo-normal at the vertices, calculating the intensities at the vertices and then interpolating the intensity linearly, we will calculate pseudo-normals at the vertices, interpolate linearly the pseudo-normals and then calculate the intensity. The intensity can be calculated using any illumination model, and in particular the Phong illumination model can be visually pleasent, though some surfaces don't have any specular highlights. (In which case you can remove the specular factor from the equation entirely).

The calculations for Phong shading are very expensive per pixel, in fact too expensive even for dedicated hardware oftentimes. It would be absurd to think that one could do real-time true Phong shading on today's platforms. But in the end, what is true Phong shading? Nothing but an approximation to what would happen if the polygon mesh really were meant to represent a curved surface. I.e. Phong shading is an approximation to an approximation. The principal objective of Phong shading is to yield nonlinear falloff and not miss any specular highlights. Who is there to say that no

other function will achieve that?

One of the solutions to this has been proposed in SIGGRAPH '86, included with this document. What they do is pretty straightforward (but probably required a lot of hard work to achieve). They make a Taylor series out of Phong shading and use only the first two terms of the series for the approximation. Thus, once the initialization phase is completed, Phong shading requires but two adds per pixel, a little more with specular highlights. The main problem, though, is the initialization phase, which includes many multiplications, divisions, and even a square root. It will doubtlessly be faster than exact Phong shading, but I am not so confident that it will be sufficiently fast to become very popular. Using Gouraud shading and a very large number of polygons can yield comparable results at a very acceptable speed.

In future versions of this document, I hope to discuss other ways of not missing the specular highlight. But since this is exam week, I'll keep it to this for the moment being.

Version history

Original version (no version number): original draft, chapters 1 through 8.

Version 0.10 beta: added chapters 9 through 11, added version history. Still no proofreading, spellchecking or whatever.

Version 0.11 beta: modified some chapters, noticed than (Q,P,R) is the normal vector (chapter 5). A little proofreading was done.

Version 0.12 beta: I just noticed that this document needs a revision badly. Still, I don't have time. I just added a few things about the scalar multiplication of two vectors being the D coefficient in the plane equation. Added chapter 12. Also corrected a part about using arrays instead of lists. The overhead is just as bad. Better to use lists. Ah, and I have to remember to add a part on calculating the plane equation from more than three points to reduce roundoff error.

Version 0.20 beta: Still need to add the part about calculating A,B and C in plane eq. from many points. However, corrected several of the mistakes I talked about in the preceding paragraph (i.e. revision). Ameliorated the demonstration for 2d rotations. Started work on 3d rotations (this had me thinking...)

Version 0.21 beta: converted to another word processor. Will now save in RTF instead of WRI.

Version 0.22 beta: corrected a few goofs. 3d rotations still on the workbench. Might add a part about subdividing a polygon in triangles (a task which seems to fascinate people though I'm not certain why). Will also add the very simple Gouraud shading algorithm in the next version (and drop a line about bump mapping). This thing needs at least a table of contents, geez... I ran the grammar checker on this. Tell me if it left anything out.

Version 0.23 beta: well, I made the version number at the start of this doc correct! :-) Did chapter 13 (subdividing into triangles) and chapter 14 (Gouraud shading).

Version 0.24 beta: removed all GIFs from file because of recent CompuServe-Unisys thingy. Don't ask me for them, call CompuServe or Unisys [sigh].

Version 0.30 beta: added the very small yet powerful technique for texture mapping that eliminates the division per pixel. It can also help for z-buffering. Corrected the copyright notice.

Version 0.31 beta: I, err, added yet another bit to texture mapping [did this right after posting it to x2ftp.oulu, silly me]. Corrected a typo in chapter 13: a n-sided polygon turns into n-2 triangles, not n triangles. Was bored tonight, added something to the logs chapter (I know, it's not that useful, but hey, why would I remove it?) Added «Canada» to my address below (it seems I almost forgot internet is worldwide tee hee hee). Addendum: I'm not certain that the GIF thing (see the paragraph about v0.24beta above) applies to me, but I'm not taking any chances.

Version 0.32 beta: added bits and pieces here and there. The WWW version supposedly just became available today. Special thanks to Lasse Rasinen for converting it. Special thanks also to Antti Rasinen. I can't seem to be able to connect to the WWW server though, I just hope it really works. Corrected a goof in the free directional texture mapping section.

Version 0.40 beta: Kevin Hunter joined forces with me and made that part in chapter two about changes in coordinates system, finding the square root et al. Modified the thing about triangulating a polygon. Added the Sutherland-Hodgman clipping algorithm and related material (the whole chapter 15). This document deserves a bibliography/suggested readings/whatever, but I don't think I have the time for that. Might add a pointer to the 3d books list if I can find where it is. Anyway, if you're really interested and want to know more, I'm using Computer Graphics, Principles and Practice by Foley, van Dam, Feiner and Hughes, from Addison-Wesley ISBN 0-201-12110-7. There. It's not in the right format, but it's there. MY E-MAIL ADDRESS CHANGED! PLEASE USE THIS ONE FROM NOW ON!

Version 0.41 beta: added to chapter 3. Gave a more detailed discussion of a scan-line algorithm. Put in a line about Paul Nettle's sbuffering in the same chapter. Will now include a "where can I get this doc from" in the readme, and I'll try to put the last few version history entries in my announcements for future versions of the doc.

Version 0.42 beta: removed the buggy PS file. Added an ASCII file containing most of the pics of the document.

Version 0.50 beta: added chapters 16, 17 and 18 about Phong shading and illumination model, interpolations forward differencing and Taylor series. I am also including an extract from SIGGRAPH '86 for fast Phong shading. Source code is coming out good, so it just might be in the next version.

About the author

Sébastien Loisel is a student in university. He is studying computer sciences and mathematics and has been doing all sorts of programs, both down-to-earth and hypothetical, theoritical models. He's been thinking a lot about computer graphics lately and so decided to write this because he wanted to get his ideas straight. After a while, he decided to distribute it on Internet (why the hell am I using third person?)

The author would be happy to hear your comments, suggestions, critics, ideas or receive a postcard, money, new computer and/or hardware or whatever (grin). If you do want to get in touch with me, well here's my snail mail address:

Sébastien Loisel
1 J.K. Laflamme
Lévis, Québec
Canada
G6V 3R1

MY E-MAIL ADDRESS CHANGED! PLEASE USE THIS ONE FROM NOW ON!
loiselse@ift.ulaval.ca

Addendum: see enclosed file, README.3D.

I would like to thank Lasse Rasinen making the WWW version available by converting this doc to the proper format. I would also like to thank Antti Rasinen for the WWW version. Thanks, guys.

I would like to thank Kevin Hunter for writing a big chunk of chapter 2. Thank you Kevin. Kevin Hunter can be reached (at the time of this writing) as HUNTER@symbol.com.