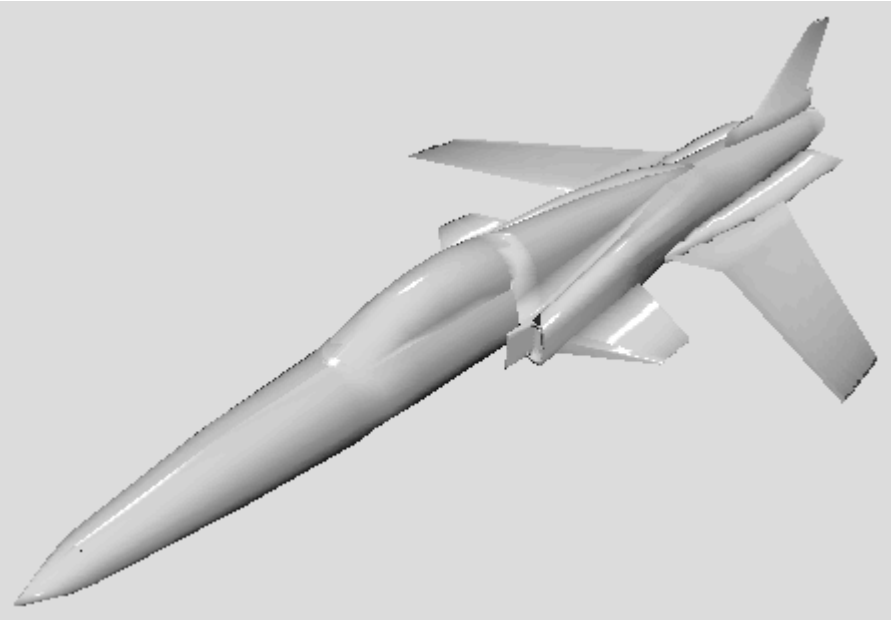
Zed3D
A compact reference for 3d computer graphics

by Sébastien Loisel

© 1994, 1995

Zed3D - A compact reference for 3d computer graphics



Copyright 1994, 1995 by Sébastien Loisel All Rights Reserved, version 0.80β

Welcome to Zed3D the entirely reformatted, entirely re-written, and hopefully, more useful version of Zed3D. Zed3D is a document about computer graphics, more particulary real-time 3d graphics. This document is not meant as a complete reference to 3d graphics, but rather perhaps as a handy place where you can find interesting stuff, and also sometimes stuff to supplement your paperback reference(s). It can also serve as introductory material for a mathematically able person without experience in computer graphics. In this case, this document would give the reader a very solid understanding of the fundamentals of computer graphics, and some more.

The original Zed3D document grew out of my worknotes. As a matter of fact, the original Zed3D, up to 0.61beta, was my worknotes. As such, it was messy, incomplete and often incorrect. I have attempted to correct this a bit now. I still consider these my worknotes, but I have added more formal introductory materials which were not in the original document.

In this document, I will attempt to describe various aspects of computer graphics in a clear, useful and complete fashion. You will find quite a bit of the theoritical aspects, copies of the calculations and proofs I made and so forth.

However, there is the painful fact that I am merely a student, trying to mark my territory in the university work, and since this work does not serve that purpose very well, Zed3D will oftentimes be lacking in areas that I wish I had more time to document. Also, I will attempt to distribute another nice portable graphics engine in the future, but that's only if I can find the time to make it.

Also, please note that this document and any accompanying documentation/software for which I am the author should not be considered public domain. You can redistribute this whole thing, unmodified, if no fee is charged for it, otherwise you need the author's written permission. Also I am not responsible for anything that might happen anywhere even if it's related directly or indirectly to this package. If you wish to encourage my effort, feel free to send me a 10$ check, which will be considered to be your official registration. If you're really on a budget, I would appreciate at least a postcard. At any rate, please read the registration paragraph below. There have been rumours about a 0.70 version of Zed3D about. This would be a fake, versions between 0.63 and 0.79 do not exist, and have never existed.

Contact information

If you wish to contact me for any reason, you should be using the following snail-mail address or my e-mail address. Given that snail-mail addresses tend to be more stable over time, you might try it if I don't answer to your electronic messages.

```
E-Mail Address: zed@binkley.cc.mcgill.ca

Snail Mail Address:

For the 1995-1996 school year, I will reside at:

Sébastien Loisel
3436 Aylmer Street, appartment 2
Montréal, Québec, Canada
Postal Code: H2X 2B6

Otherwise, it is possible to reach me at:

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

Registration

If you want to register your copy of Zed3D for life, and be able to use the source in any way you want, even commercial (though commercial utilization of the documentation [this file] still requires the written permission of the author), you can send me a cheque of US$10.00. For more information, please consult the file register.frm, which should have come with this document. If you are experiencing difficulties with registration or if the file register.frm is missing, please mail me and we will see if something can be worked out.

Overview

I am trying to put a bit more structure into this document. As such, this is how it is meant to be structured at this moment.

The first section is heavily mathematical. It deals with transformations by and at large. First are discussed linear and affine transformations, which are used to spin and move stuff in space in a useful fashion, then is discussed and justified the perspective transforms. These two sections are very theoritical, but are required for proper understanding of the later sections.

Then there will follow a section dealing specifically with applications of the preceding theory. Namely, rotation matrices and their inverse and object hierarchy.

The third "section" concerns itself mainly with rendering techniques. These are becoming less and less important for several reasons. The complexity of the problem is of course not in the rendering section of the pipeline. Second, the recent trend has pushed the rendering part of the pipeline into cheap video hardware which can do the job fast and effectively while the CPU is off to some other, more important task. Eventually, we can hope that transforming objects will also be made a part of popular low-cost hardware, but that remains to be seen. As it is now, this is often the job of either the CPU, or sometimes we might wish to give this job to a better co-processor (for example, a programmable DSP).

Fourthly, an attempt will be made to describe a few shading models and visible surface determination techniques. Shading models are but loosely related to the way the polygons are drawn. Visible surface determination depends somewhat more on the way polygons are drawn, and is often implemented in hardware.

The following section deals with a few of the computer graphics related problems that are often encountered, such as error tolerant normal computation, the problem of finding a correctly oriented normal and polygon triangulation.

Of course, a lot of topics remain to be covered, such as real-time collision detection, quaternions and SLERP for rotation interpolation and keyframe animations, octrees and other data structures. However, I unfortunately do not have the time to write all of that down for the general public.

Table of Contents

Zed3D - A compact reference for 3d computer graphics 2

- Contact information 3
- Registration 3
- Overview 4

Table of Contents 5

Vector mathematics 8

- Introduction 8
- Vector operations 8
  - Exercises 9
  - Answers 10
- Alcoholism and dependance 10
  - Exercises 11
  - Answers 12
- On a plane (and of motion sickness) 12
  - Exercises 13
  - Anwers 14

Matrix mathematics 15

- Introduction 15
- Matrix operations 15
  - Exercise 17
  - Answer 17
- Matrix representation & linear transformations 18

Affine transforms 20

- Introduction 20
- Affine transformations 20

Applications of linear transformations 22

Introduction 22
World space, eye space, object space, outer space 22
Transformations in the hierarchy (or the French revolution) 23
Some pathological matrices 23

Perspective 26

Introduction 26
The perspective transformation 27
Theorems 29
Other applications 30
Reality strikes 32

Interpolations and approximations 33

Introduction 33
Forward differencing 33
Approximation function 35

Rendering 36

Introduction 36
The point 36
Lines 39
Polygon drawing 39

Visible surface determination 42

Introduction 42
Back-face culling 43
Sorting 44
Painter's algorithm and depth sorting 46
Z-Buffering 47

Lighting models 49

Introduction 49
Lighting models 49
Smooth shading 50

Computer graphics related problems 54

Introduction 54
Generating edge normals 54

Triangulating a polygon 55
Computing a plane normal from vertices 56
Generating correctly oriented normals for polyhedra 57

Glossary 59

Vector mathematics

Introduction

Linear algebra is a rather broad yet basic field of college level mathematics. It is being taught (or should be at any rate) early on to students in mathematics and engineering. However simple it is, it's a lengthy (and somewhat boring, in my humble opinion) topic to discuss. And since this document is not meant as a mathematics textbook, I will only give here the gist of the thing.

If you need further information on the topic, browse your local library for linear algebra books and somesuch, or go ask a professor. As of now, I'm not making any bibliography for this, but if and when I do, I'll try to give a few decent references.

The purpose of linear algebra in 3d graphics is to implement all the rotation, skewing, translation, changes in coordinates, and otherwise affine transformations to 3d object. The applications range from merely rotating an object about it's own system of axis to object hierarchy, moving cameras and can be extended through quaternions for rotation interpolation and such.

As such, linear algebra is something that is essential to the usefulness of a 3d graphics engine.

Since my prime concern is 3d graphics, I will give only whatever theory is absolutely necessary to 3d mathematics. What's below extends in a vary natural way to n dimensions, n>3, except for cross product, which is a bit awkward.

Vector operations

A vector in 3d is noted (a,b,c) where a, b and c are real numbers. Similarly, a vector in 2d is noted (a,b) for a, b real numbers. The vector for which all components are null deserves a special mention, it is usually noted 0, with the proper number of components implied in the notation but not explicitly given.

Vector addition is defined as follow. Let U=(u1,u2,u3) and V=(v1,v2,v3) then the notation U+V means (u1+v1,u2+v2,u3+v3). Similarly, for 2d vectors, (u1,u2)+(v1,v2) means (u1+v1,u2+v2).

Multiplication of a vector by a scalar is defined as follow. Given vector U and a scalar a (a is a real number), then a*U means (a*u1,a*u2,a*u3).

Multiplication by the scalar -1 has a special notation. -1*U is written simply -U.

Vector difference is a theorem from the above. U-V can be rewritten U+-1*V, which is a simple addition and a multiplication by the scalar -1 as above.

Multiplication of two vectors has no intuitive meaning. However, two types of "multiplications" of vectors are usually defined, which have little or no relation to the usual real number multiplication.

The first is dot product. U dot V (usually noted U•V) yields a real number (not a vector). (u1,u2,u3)•(v1,v2,v3) means u1*v1+u2*v2+u3*v3. Similarly for 2d vectors, (u1,u2)•(v1,v2)≡u1*v1+u2*v2.

Vectors have a length, defined as follow. The lentgh (or module, or norm) of vector U is written |U| and has the value of $(U•U)^{1/2}$. If the length of a vector is one, the vector is said to be of unit length, or a unit or normal vector.

Dot product is also used to <u>define</u> angle. U•V=|U|*|V|*Cosθ, where θ is the angle between U and V. Incidentally, if |U|=1 then this simplifies to |V|*Cosθ, which is the length of the projection of V onto U. It is of note that U•V is 0 if and only if either θ is π/2+2kπ or |U|=0 or |V|=0. Assuming |U| and |V| are not 0, this means that if U•V is 0, then U and V are perpendicular, or orthogonal.

The second product usually defined on vectors is the cross product. U cross V (usually noted U×V) is defined using matrix determinant and somesuch. Basically, (u1,u2,u3)×(v1,v2,v3) is u1*(v2-v3)+u2*(v3-v1)+u3*(v1-v2).

It is demonstratable that the cross product of two vectors is perpendicular to the two vectors and has a length of |U||V|Sinθ. The fact that it is perpendicular has applications that we will see later.

Exercises

Q1 - Do the following vector operations:

a) (1,3,2)+(3,5,6)

b) 1.5*(3,4,2)

c) (-1,3,0)•(2,5,2)

d) |(3,4,20/3)|

e) U•V where |U|=2, |V|=3 and the angle between U and V is 60 degrees

f) (1,2,3)×(4,5,6)

Q2 - Which vectors satisfy the equation U•(1,1,1)=0?

Answers

A1 - a) (4,8,8)

b) (4.5,6,3)

c) 13

d) 25/3

e) 3

f) 0

A2 - All vectors that satisfy u1+u2+u3=0. Since the dot product is 0, this means all vectors that are perpendicular to U. Incidentally, these vectors cover the whole plane and nothing but the plane for which the normal is (1,1,1).

Alcoholism and dependance

Given a set of vector U0, U1, U2, ... , Un, these vectors are said to be linearly independant if and only if the following is true:

a0*U0+a1*U1+...+an*Un=0 implies that (a0,a1,a2, ... , an)=0.

If there exists a single solution for which (a0, a1, a2, ...,an) is not zero, then there exist an infinity of them, and the vector are said to be linearly independant.

The geometric interpretation of that is as follows. In 3d, three vectors are linearly independant if none of them are colinear and the three of them are not coplanar. (Colinear means on the same line, coplanar means on the same plane). Any more than 3 vectors and they are certain to be linearly dependant.

For two vectors, in 2d or 3d, they are said to be linearly independant if they are not colinear. 3 or more vectors in 2d are always linearly dependant.

If a set of vectors are linearly independant, they are said to form a basis. 2 linearly independant vectors form the basis for a plane, and 3 linearly independant vectors form the basis for a 3d space.

The term orthogonal is very frequently used to described parallel vectors. If a basis is said to be made of orthogonal unit vectors, the base is said to be orthonormal. Orthonormal basis are the most useful kind in usual 3d graphics. If a basis is not orthonormal, it "skews" the space, where if the vectors are not unit, it "stretches" and/or "compresses" the space.

Each space has a so-called canonical basis, the basis we intuitively find most simple. For 3d space, that basis is made of the vectors (1,0,0), (0,1,0) and (0,0,1). Similarly, the canonical basis for 2d space is (1,0) and (0,1). Note that since a basis is a set of vectors, it would be more formal to enclose the list of vectors in curly braces, for example, {(2,3) , (-1,0)}.

The vectors of the canonical basis are traditionnally noted i, j and k for 3d space or i and j for 2d space. This leads us to introduce another notation.

If vector (a,b,c) is said to be expressed in basis pqr, then it means that the vector is a*p+b*q+c*r. Note that a, b and c are scalars and p, q and r are vectors, thus this combination (formally referred to as linear combination) is defined as discussed earlier. If pqr are ijk, this translates to a*i+b*j+c*k or (a,0,0)+(0,b,0)+(0,0,c) or (a,b,c).

However, if pqr is not ijk, the matter is different. For example (assuming pqr is expressed in ijk space), if p=(1,1,0), q=(0,1,1) and r=(1,0,1), then the vector (a,b,c) in pqr space means (a,a,0)+(0,b,b)+(c,0,c)=(a+c,a+b,b+c) in ijk space. What we just did is called a change of basis. We took a vector that was expressed in pqr space and expressed it in ijk space.

It would be possible for pqr to be expressed in some other base than the canonical base. If that were the case, and if the objective would be to express vector (a,b,c) in ijk space, then it might require several transformations to get there.

For simplicity's sake in the further parts of this document, we will extend our definition of vectors to allow for not only real number components, but also vector components. This means that (a,b,c) expressed in pqr space is a*p+b*q+c*r can be written as (a,b,c)•(p,q,r).

Exercises

Q1 - Are the vectors (1,2,0), (4,2,4) and (-10,-4,-8) linearly independant?

Q2 - Say vector U=(1,3,2) is expressed in pqr space, where pqr expressed in ijk space is (1,2,0), (2,0,2), (0,-1,-1). Express U in ijk space.

Q3 - Using Question 2's values for p, q and r, and the vector V expressed in ijk space as (1,1,1), can you express V in pqr space?

Answers

A1 - No

A2 - (7,0,4)

A3 - (4/3, -1/6, -5/6) - this exercise is in fact called an inverse transform, which will be described later.

On a plane (and of motion sickness)

There are several ways to define a plane in 3d. The first one I will present is useful because it can be used to represent a plane in n dimensional space, even for n>3.

First you need two linearly independant vectors to form a basis. Call them U and V. Then, if you take a*U+b*V for all possible values of a and b (them being real numbers of course), you generate a whole plane that goes through the origin of space. If you want to displace that plane in space by vector W (e.g. you want the point pointed to by W to be part of the plane), then a*U+b*V+W will generate the desired plane. (Proof, for a=b=0, it simplifies to W, thus the point at W is part of the plane).

Note that the above equation can be written (a,b)•(U,V)+W. As such it can be viewed as a change of basis, from the canonical basis of 2d space to whatever space U and V's basis is.

Another way for defining a plane that only works in 3d is as follows. (Note, in n dimensional space, this will define a n-1 dimensional object). As was seen earlier, if A•X=0, then A and X are perpendicular (A and X are vectors). Furthermore, for a given A, if you take all X's that satisfy the equation, you get all points in a certain plane. A is generally called normal to the plane, although the litterature frequently assumes that the normal is also of unit length, which is not necessary (though A must not be the null vector). The values of X that satisfy the plane equation given include X=0, since A•0=0 for any value of A. Thus, that plane passes through the origin.

If one wants a plane that does not pass through the origin, one should proceed as follow. (This uses an intuitive form of affine transformations, descibed in depth later). First, find out the displacement vector K that describes the position of the plane in relation to the origin. Thus, if you subtract K from all the points in the plane, the plane ends up at the origin, and we can use the definition above. Thus, the new definition of the plane is A•(X-K)=0.

To make this a bit more explicit, let A=(A,B,C) and X=(x,y,z) and K=(k1,k2,k3). Then the plane equation can be rewritten as: A*(x-k1)+B*(y-k2)+C*(z-k3)=0. A little algebra allows us to rewrite it as A*x+B*y+C*z=-A*k1-B*k2-C*k3. By setting D=-A*k1-B*k2-C*k3, we can make one more rewrite, which is the final form: A*x+B*y+C*z=D.

It is important to remember that multiplying both side of the equation by a constant does not change the plane. Thus, plane x+y+z=1 is the same as plane 2x+2y+2z=2.

Note that in this last representation, (A,B,C) is the normal vector to the plane. The last equation can also be re-written A•X=D. It would also be easy to demonstrate the following, but I will not do it. For any point P, (A•P-D)/|A| is the signed distance to the plane A•X=D. The sign can help you determine on what side of the plane that the point P lies on. If it is 0, P is on the plane. If it is positive, P is in the direction that the normal points to. If it is negative, P is on the side opposite of the normal. This has application in visible surface determination (namely, back face culling).

Also note that if |A|=1, then the distance equation simplifies to A•P-D.

It is trivial to demonstrate that the equation for a line in n-space, for any integer value of n>0, is t*U+W, where U is a vector parallel to the line and W is a point on the line. As t takes all real values, we generate a line.

Exercises

Q1 - Given the basis U=(1,3,2) and V=(2,2,2), and the position vector W=(1,1,0), find the position in 3d space of the point (3,2) in UV space.

Q2 - Express the plane described in Q1 in the form Ax+By+Cz=D

Q3 - Find the signed distance of point (4,2,4) to the plane using the answer for question 2.

Q4 - Given two basis vectors for a plane, P and Q, in 3d space, and a position vector for the plane, R, plus the direction vector of a line, M, that passes through origin, find the pq space point of intersection between the line and the plane.

Anwers

A1 - (8,14,10)

A2 - x+y-2z=2 (hint : remember that the cross product of U and V is perpendicular to both U and V).

A3 - $-4/(6^{1/2}) \cong -1.633$ - this means that the point (4,2,4) is in the direction opposite of (1,1,-2) from the plane x+2-2z=2.

A4 - See the perspective chapter on texture mapping.

Matrix mathematics

Introduction

Matrices are a tool used to handle a great deal of linear combinations in a homogeneous way. The operations on matrices are so defined as to ease whatever task you want to do with them. Be it expressing a system of equations, or making a change of basis, to some peculiar uses in calculus.

Normally, matrices are noted using large parenthesis and the numbers written down in a grid-like disposition as follows. This is a generic 3x3 matrix:

$$\begin{pmatrix} m11 & m12 & m13 \\ m21 & m22 & m23 \\ m31 & m32 & m33 \end{pmatrix}$$

In general, a pxq matrix is noted as above with the exception that it has p rows and q columns. The above matrix can also be written $M=(m_{ij})$ with i and j varying from 1 to 3.

A matrix for which p=q, such as the M matrix above, is said to be a square matrix. There exist a particular type of square matrix called an identity matrix. There is one such matrix for each type of square matrix (e.g. one for 1x1 matrices, one for 2x2 matrices, one for 3x3 matrices, etc...) As an example, the 3x3 matrix is given here:

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

Strictly speaking, the identity matrix $I=(m_{ij})$ is defined such as:

$m_{ij}=0$ if $i \neq j$ and $m_{ij}=1$ if $i=j$

Matrix operations

Matrix addition is defined as follow. Given 2 matrices $A=(a_{ij})$ and $B=(b_{ij})$ of same dimension pxq, then $U=(u_{ij})=A+B$ is defined as being $(u_{ij})=(a_{ij}+b_{ij})$.

Matrix multiplication by a scalar is defined also as follows. Given the matrix M and a scalar k, then the operation $U=(u_{ij})=k*M$ is defined as $(u_{ij})=(k*m_{ij})$.

Matrix multiplication is a bit more involved. It is defined using sums, as follows. Given matrix A of dimension pxq, and matrix B of dimension qxr, the product C=AxB is given by:

$$(c_{ij})=\sum_{1\leq k\leq q}(a_{ik}*b_{kj})$$

More explicitly, for example, we have, for A and B 2x2 matrices:

$$c_{11}=a_{11}*b_{11}+a_{12}*b_{21}$$

$$c_{12}=a_{11}*b_{12}+a_{12}*b_{22}$$

$$c_{21}=a_{21}*b_{11}+a_{22}*b_{21}$$

$$c_{22}=a_{21}*b_{12}+a_{22}*b_{22}$$

(Note: $\sum_{1\leq k\leq q}(a_{ik}*b_{kj})$ means "sum of $(a_{ik}*b_{kj})$ for k varying from 1 to q.")

It is important to notice that matrix multiplication is <u>not</u> commutative in the general case. For example, it is not true that A*B=B*A with A and B matrices in the general case, even if A and B are square matrices. Matrix multiplication is, however, associative.

The identity matrix has the property that, for any matrix A, A*I=I*A=A (I is the neutral element of matrix multiplication).

Matrix transposition of matrix A, noted A^T, reflects the A matrix along the great diagonal. That is, say $A=(a_{ij})$ and $A^T=(b_{ij})$, then we have $b_{ij}=a_{ji}$.

There are also other interesting operations you can do on a matrix, however they are much, much more involved. As of now, I am not willing to get too deeply into this. The topics of interest are matrix determinant (which has a recursive definition) and matrix inversion. I will content myself by giving one definition of matrix determinant and one way of finding matrix inverse. Note that there are at least a couple of different definitions for determinant, though they boil down to the same thing. Also, there are many ways of finding the inverse of a matrix, I will contend myself with presenting only one method. Strict definitions will be given, for more extensive coverage, consult a linear algebra book.

Given a matrix $M=(m_{ij})$, of size 1x1, the determinant is defined as $D=m_{11}$. For matrices of size nxn with n>1, the definition is recursive. First, pick an integer j such that $1\leq j\leq n$. For example, you could pick j=1.

$D=m_{j1}*C_{j1}+m_{j2}*C_{j2}+...+m_{jn}*C_{jn}$

The C_{ij} terms require a bit more explaining, which follows.

First let us define the minor matrix M_{ij} of matrix M. If M is a nxn matrix, then the M_{ij} matrix is a (n-1)x(n-1) matrix. To generate the M_{ij} matrix, remove the i^{th} line and j^{th} column from the M matrix.

Second what interests us is the cofactor C_{ij}, which is defined to be:

$C_{ij}=(-1)^{i+j}*|M_{ij}|$, where $|M_{ij}|$ denotes the determinant of M_{ij}.

As an example, the determinant of the 2x2 matrix M is m11*m22-m12*m21, and the determinant of a 3x3 matrix M is

$$D_{3x3}= m_{11}*(m_{22}*m_{33}-m_{23}*m_{32}) - m_{12}*(m_{21}*m_{33}-m_{23}*m_{31}) + m_{13}*(m_{21}*m_{32}-m_{22}*m_{31})$$

Given a matrix A, the inverse of the matrix, noted A^{-1} (if it exists), is such that $A*A^{-1}=A^{-1}*A=I$. It is possible that a matrix has no inverse.

To inverse the matrix, we will first define the adjacent matrix of A, which we will call $B=(b_{ij})$. Let C_{ij} denote the i,j cofactor of A. Then, we have:

$b_{ij}=C_{ij}$

Which completely defines the cofactor matrix B. The inverse of A is then:

$A^{-1}=1/|A|*B^T$

Another method of inverting matrices, which might be preferable for numerical stability reasons but will not be discussed here, is the Gauss-Jordan method.

Exercise

Q1 - Compute the product of these two matrices:

$$\begin{pmatrix} m11 & m12 & m13 \\ m21 & m22 & m23 \\ m31 & m32 & m33 \end{pmatrix} \cdot \begin{pmatrix} n11 & n12 & n13 \\ n21 & n22 & n23 \\ n31 & n32 & n33 \end{pmatrix}$$

Answer

A1

$$\begin{pmatrix} m11 \cdot n11 + m12 \cdot n21 + m13 \cdot n31 & m11 \cdot n12 + m12 \cdot n22 + m13 \cdot n32 & m11 \cdot n13 + m12 \cdot n23 + m13 \cdot n33 \\ m21 \cdot n11 + m22 \cdot n21 + m23 \cdot n31 & m21 \cdot n12 + m22 \cdot n22 + m23 \cdot n32 & m21 \cdot n13 + m22 \cdot n23 + m23 \cdot n33 \\ m31 \cdot n11 + m32 \cdot n21 + m33 \cdot n31 & m31 \cdot n12 + m32 \cdot n22 + m33 \cdot n32 & m31 \cdot n13 + m32 \cdot n23 + m33 \cdot n33 \end{pmatrix}$$

Matrix representation & linear transformations

The following set of equations:

m11*x+m12*y+m13*z=A

m21*x+m22*y+m23*z=B

m31*x+m32*y+m33*z=C

is equivalent to the matrix equation that follows:

$$\begin{pmatrix} m11 & m12 & m13 \\ m21 & m22 & m23 \\ m31 & m32 & m33 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} A \\ B \\ C \end{pmatrix}$$

It is also equivalent to the following vector equations

P=(m11,m21,m31), Q=(m12,m22,m32), R=(m13,m23,m33)

X=(x,y,z)

D=(A,B,C)

D=X•(P,Q,R)

This means that matrix can be used, amongst other things, to represent systems of equations, but also a change of basis. Look back on the vector mathematics chapter and you will see that D=X•(P,Q,R) litterally means "transform X, which is expressed in PQR space, in whatever space PQR is expressed in (could be ijk space for example), the answer is labeled D."

The matrix form can also be written as follows:

M*X=D

In this case, if the matrix M is inversible, then we can premultiply both sides of the equality by M^{-1}, as follows:

M^{-1}*M*X=M^{-1}*D

And, knowing that $M^{-1}*M=I$, we substitute into the above:

$I*X=M^{-1}*D$

And knowing that $I*X=X$, we finally get:

$X=M^{-1}*D$

That is a very elegant, efficient and powerful way of solving systems of equations. The difficulty is of course finding M^{-1}. For example, if we know M, D but not X, we can use the above to find X. This is what should be used to solve question 3 in chapter "Alcoholism and dependance". For 3d graphics people, this is the single most useful application of matrix inversion: sometimes you have a point in ijk space, and you want to express them in pqr space. However, you don't originally have ijk expressed in pqr space, but you have pqr expressed in ijk space. You will then write the transformation of a point from pqr space to ijk space, then find the inverse transformation as just described and then inverse transform the point to find it's position in pqr space.

Another very interesting aspect is as follows. If we have a point P to be transformed by matrix M, and then by matrix N. What we have is:

$P'=M*P$

$P''=N*P'$

By combining these two equations, we get

$P''=N*(M*P)$

However, by associativity of matrix multiplication, we have:

$P''=(N*M)*P$

If for instance, we plan to process a great many points through these two transformations in that particular order, it is a great time saver to be able to first calculate $A=N*M$, and then simply evaluate $P''=A*P$ for all P's, instead of first calculating P' then P''. In linear transformations terminology, A is said to be the linear combination of M and N.

Affine transforms

Introduction

As of now, we have seen linear transformations. Linear transformations can be used to represent changes of basis. However, they fail to take into account possible translation, which is of top priority to 3d graphics. An affine transform is, roughly, a linear transform followed by a translation (or preceded, though it is more useful for 3d graphics to picture them as being followed by the translation instead).

Affine transformations

A simple proof can be used to demonstrate that a 3x3 matrix cannot be used to translate a 3d point. Given any 3x3 matrix A and the point P=(0,0,0), then A*P=(0,0,0), thus the point is untranslated. It is merely rotated/skewed/stretched about the origin.

However, there is a neat trick. A linear transform in 4d space projected in a particular fashion in 3d space is an affine transformation. Without going into the details, a 4x4 matrix can be used to model an affine transform in 3d. The matrix has the following form:

$$\begin{bmatrix} m11 & m12 & m13 & Tx \\ m21 & m22 & m23 & Ty \\ m31 & m32 & m33 & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The (m_{ij}) 3x3 submatrix is the normal rotation/skew/stretch (the linear transform we studied previously). The (Tx,Ty,Tz) vector is added to the point after transform. A point (x,y,z) to be transformed into (p,q,r) is noted:

$$\begin{bmatrix} m11 & m12 & m13 & Tx \\ m21 & m22 & m23 & Ty \\ m31 & m32 & m33 & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} p \\ q \\ r \\ 1 \end{bmatrix}$$

Another way of modelling affine transform is to use the conventional 3x3 matrix we were using previously, and to add a translation vector after each linear transform. The advantage of this is that we do not do unnecessary multiplications for translation and also the bottom row of the 4x4 matrix which is (0,0,0,1) that can be optimized out. However, the advantage of using the 4x4 matrix on the conceptual level (not on the implementation level) is that you can then compute affine transformation combinations and inversions, the exact same way that we were doing in the previous section.

Applications of linear transformations

Introduction

In this section we will discuss the applications of the linear transformation theory we saw in the previous sections. When doing 3d graphics, the usual situation occurs. We have a description of one or more objects. We have their locations and orientations in space, relative to some point of reference. We move them around, rotate them, usually about their own coordinate system. The camera might also be moving, rotating and such. In that case, it is likely that we have an orientation and position for the camera object itself. We would also like that the eye points in the direction of (0,0,1) in camera space, and that up be (0,1,0) in camera space.

Orientation and position will be given by an affine transform matrix. The (m_{ij}) submatrix gives orientation and the 4th column has the translation vector.

World space, eye space, object space, outer space

First off we are going to require a global system of reference for all the objects. This is usually called "World space". An affine transform that describes an object's position and orientation usually does so in relation to world space (this is generally not true for hierarchical structures, as we will see later). This introduces a new concept; a matrix A, representing an affine transform that takes an object from space M to space N (in our example, M is object space and N is world space) is usually noted $A_{N\leftarrow M}$. This has the natural tendency to make us combine the affine transform from right to left instead of left to right, which is correct (see a preceding section).

The most typical example is as follows. We have an object and its affine transform $A_{World\leftarrow Object}$. We also have a camera position and orientation given by $C_{World\leftarrow Camera}$. In that case, the first thing we want to do is invert the transform $C_{World\leftarrow Camera}$ to find the $C_{Camera\leftarrow World}$ transform. Then you will be transforming the points P_i in the object with $C_{Camera\leftarrow World}*A_{World\leftarrow Object}*P_i$.

As a helper, notice that the little arrows make a lot of sense, as shown below:

Camera←World, World←Object, which concatenates intuitively to Camera←World←Object or simply Camera←Object. Thus, the above transformation transforms from object space to camera space directly. One merely calculates $M_{Camera \leftarrow Object} = C_{Camera \leftarrow World} * A_{World \leftarrow Object}$ and multiplies all P_i's with is.

Transformations in the hierarchy (or the French revolution)

It may be useful to express an object A's position and orientation relative not to the world, but to some other object B. This way, if B moves, A moves along with it. In plain words, if we say "The television is resting 2 centimeters above the desk on its four legs", then moving the desk does not require us to change our "2 centimeters above the desk" position - it is still 2 centimeters above the desk as it is moving along with the desk (careful not to drop it). On the other hand, if we had said "The television is 1 meter above the floor" and "The desk is 95 centimeters above the floor", and then proceed to move the desk up 1 meter, then the position of the desk is "1m95 above the floor". Additionnally, we have to edit the position of the television and change it to "The telebision is 2 meters above the floor". Notice the difference between these two examples.

This can be implemented very easily the following way. Make an affine transform that describes orientation and position of television in relation to the desk. This is called $A_{Desk \leftarrow Television}$. Then we have an orientation and position for the desk, given by $B_{World \leftarrow Desk}$. Notice that this last affine transform is relative to world space. We then of course have the mandatory $C_{World \leftarrow Camera}$ which we invert to find the $C_{Camera \leftarrow World}$ transform. We then proceed to transform all points in the television to camera space, and also all points from the desk to camera space. The former is done as follows:

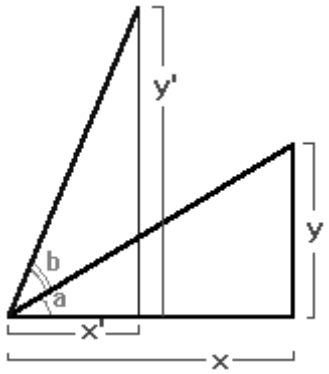
$C_{Camera \leftarrow World} * B_{World \leftarrow Desk} . * A_{Desk \leftarrow Television} * P_i .$

Notice again how the arrows concatenate nicely. The points on the desk are transformed with this:

$C_{Camera \leftarrow World} * B_{World \leftarrow Desk} . * Q_i .$

Again, the arrows make all the sense in the world.

Some pathological matrices



Rotating a point in 2d is fundamental. In the example above, we wish to rotate (x,y) to (x',y') by an angle of 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)

Rotation in 3d are done about one of the axis. The exact rotation used above would rotate about the z axis. In matrix representation, we write the x, y and z axis rotations as follows:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \qquad \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix} \qquad \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

(x axis) (y axis) (z axis)

These matrices can be extended to 4x4 matrices simply by adding a rightmost column vector of (0,0,0,1) and a bottom row vector of (0,0,0,1) (e.g. the 1 in the bottom right slot is shared by the column and the row vector).

If you want, you can always specify the orientation of an object using three angles. These are formally referred to the euler angles. Unfortunately, these angles are not too useful for many reasons. If two angles change with linear speed, the object will definetly not rotate in a linear fashion. Also, sometimes, a problem known as gimbal lock occurs, where you suddenly lose one degree of freedom (this looks like the object's rotation in a direction stops, to start again in another strange direction). Furthermore, the angles are not relative to object coordinate system nor world coordinate system.

Thus it is preferable to specify object orientation with an orientation matrix. When rotation about a world axis is desired, the orientation matrix is premultiplied by one of the above rotation matrices, and when a rotation about an object axis is desired, the orientation matrix is postmultiplied by one of the above rotation matrices. Note that it is possible to rotate about an arbitrary vector when using quaternions, but this will not be covered here.

Perspective

Introduction

Perspective was a novelty of the Renaissance. It existed a long time before but had been forgotten by the western civilizations until that later time. As can be seen from paintings before Renaissance, artists had a very poor grasp of how things should appear on a painting. The edges from tables and desks were not drawn converging to an "escape point", but rather all parallel. This gave these paintings the peculiar feeling they have when compared to more modern, more perspective-correct paintings.

Perspective is the name we give to that strange distortion that happens when you take a real-life 3d scene (your garden) and take a picture of it. The flowers in the foreground appear larger than the barn in the background. This particular effect is sometimes referred to as *foreshortening*. Other effects come into play, such as focus blur (very likely, you were either focussed on the flower or the barn; one looks clear, the other is very fuzzy), light attenuation, atmospheric attenuation, etc...

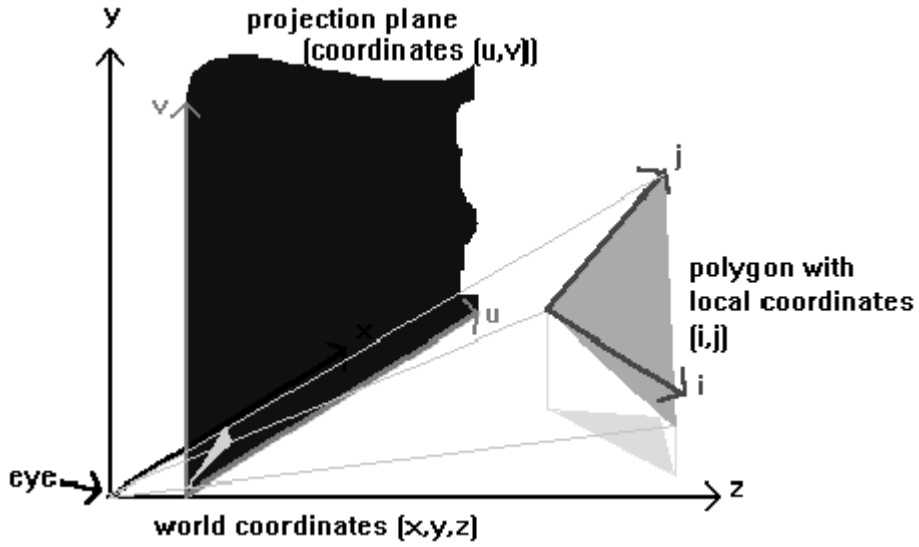
We know today that light rays probably aren't moving in a straight line at all. Even in the vacuum, they oscillate a bit. When travelling through matter, it is deviated all the time, split, reflected and all sorts of other nonsense. Sometimes it can be useful to model all these nice effects, however, they are not always necessary or desirable. One thing is for sure, a perfect or near-perfect simulation of all that we know about light today would be tremendously CPU-intensive, and would require an incredible amount of work on the software end of the project.

In normal, day-to-day life, when you're significantly larger than an atom but significantly smaller than a planet, light is usually pretty linear. It travels in straight rays, only bending at discrete points that are more or less easy to calculate, definetly more than the fuzzy way light bends in a prism.

A further simplification that we can make is that light only reflects diffusely on the objects around you. This is usually the case, unless you come up to a highly polished or metallic surface where you can see your reflection. But the usual desk, bed, snake and computers are pretty dull in appearance, with usually a soft diffuse highlight from where the light is coming from.

Another simplification we usually make comes from the fact that light bounces off everything and eventually starts coming from about all direction with a low intensity. This is often called the ambiant light. Some further optimizations, more hacks than actual physical observations, will make you go faster and still look good.

The perspective transformation



The perspective transformation is incredibly simple once you know it, but it is often good to know where it comes from. We will put to use some of the assumptions we previously stated.

The first assumption we made is that light goes in a straight line. This is great because it will allow us to make maximum use of all the linear math we have learnt since highschool.

What we have to realize is, for the eye to see an object, light has to travel from the object to the eye. Since light travels in a straight line, it has to either go straight to the eye or bounce off a few reflective surfaces before getting there. However, since we are assuming there are no such reflective surfaces in the environment, the only possibility left is that the light comes straight from the object to the eye. This line is formally refered to as a projector.

Another way doing it is the exact inverse. Starting from the eye, shoot a ray in a direction until it hits something. That is what you are seeing in that direction.

Obviously, we are not going to shoot an infinite number of rays in all direction, we would never even start generating an image if we did that. The usual approximation is to shoot a finite amount of rays spread over an area in an arbitrary manner.

There is another matter that needs to be taken care of. In reality, the image will be sent to screen, paper or some other media. This means that, in our model, the light does not reach the eye, it stops at the screen or paper, and that is what we display, so that reality takes over for the rest of the way and carries real light rays from the screen to the real eyes. This poses a problem of finding where the light rays intersect the screen or paper.

Using the material in the previous section, we are able to transform all objects to camera space, where forward is (0,0,1) and up is (0,1,0) and the eye is at (0,0,0). We still do not know where in space the screen lies. We will have to make a few more assumptions, that it is in front of the eye, perpendicular to the eye direction which is (0,0,1), and flat. The distance at which it lies is still undecided. We will just work with the constant k for the distance, then see what value of k interests us most. The eye is formally refered to as center of projection, and the plane the surface of projection.

Since it is flat, it lies on a plane. The plane equation in question is Ax+By+Cz=D as seen before, where (A,B,C)=(0,0,1) is the plane normal. Thus the plane equation is z=D. The distance from the eye is thus D, and we want it to be k, so we set D=k. The plane equation is therefore z=k. We set a local basis for that plane with vectors i=(1,0,0) and j=(0,1,0) and position W=(0,0,k). The plane equation is thus (a,b)•(i,j)+W. (a,b) are the local coordinates on the plane. They happen to correspond to the (x,y) position on the plane in 3d space because (i,j) for the plane is the same as (i,j) for the world.

The question we now ask ourselves is this: given a point that is reflecting light, say point (x,y,z), what point on screen should be lit that crosses the light ray from (x,y,z) to the eye, which is at (0,0,0).

Here we will use the definition of the line in n space we mentionned before (namely, tV+W). Since the light ray goes from (x,y,z) to (0,0,0), it is parallel to the vector (x,y,z)-(0,0,0)=(x,y,z). Thus, we can set V=(x,y,z). (0,0,0) is a point on the line, so we can set W=(0,0,0). The line equation is thus t(x,y,z).

We now want the intersection of the line t(x,y,z) with the plane z=k. Setting t=k/z (assuming z is nonzero), we find the following: k/z(x,y,z)=(k*x/z,k*y/z,k). This point has z=k thus it is in the plane z=k, thus it is the intersection of the plane z=k and the line t(x,y,z).

Trivially from that, we find that the point (a,b) on screen are (k*x/z,k*y/z). Thus,

(x,y,z) perspective projects to (k*x/z, k*y/z).

A small note on aspect ratios. Sometimes, a screen's coordinate system is "squished" on one axis. In this case, it would be wise to "expand" one of the coordinates to make it larger to compensate for the screen being squished. For example, if the screen pixels are 3/4 as wide as they are high, it would be wise to multiply the b component of screen position by 3/4, or the a component by 4/3. This can be computed using 2 different values of k instead of the same. For example, use k1=k and k2=ratio*k. Then, the perspective projection equation is:

(x,y,z) perspective projects to (k1*x/z, k2*y/z).

Referring again to physics, only one point gets to be projected to a particular point on screen. That is, closer objects obscure objects farther away. It will thus be useful to do some form of visible surface determination eventually. Another special case is that anything behind the eye does not get projected at all. Thus, if before the projection, $z \leq 0$, do not project.

Theorems

The following theorems are not always entirely obvious, but they are of great help when doing 3d graphics. I will attempt to give the reader rough proofs and justifications when possible, usually they will be geometrical proofs for they are much more natural in this case. These proofs are not very formal, but formal proofs are not hard to find, just much less natural.

A line in 3d perspective projects to a line in 2d. However, line segments sometimes have erratic behavior. The proof is as follows. If the object to project is a line, then the set of all projectors pass through the center of projection, which is a point, and the line. Since projector are linear, they all belong to the plane P defined by the line and the point. Thus, the projection will lie somewhere in the intersection of the plane P and the projection plane. However, the intersection of two planes is generally a line. Here follows the exception.

If the planes are parallel, since the projection plane does not pass on the eye, they are necessaraly disjoint. The projection in this case is nothing.

A line segment generally projects to a line segment. First, the only portion of the line segment that needs to be projected is the portion for which $z>0$, as seen previously. If the segment crosses $z=0$, it should be cut at $z=0$, and only the $z>0$ section kept. Second, the projectors for a line segment all lie in a scaled up triangle which intersects the projection plane in a particular way, and the intersection of a triangle and a plane is always a line segment.

Next proof is the proof that a n-gon (a n-sided polygon, example, triangles are 3-gons, squares are 4-gons, etc...) projects to a n-gon. It can be demonstrated that any polygon can be triangulated in a finite set of triangles, so the proof is kept to triangles only. Also, if the n-gon crosses z=0, it should be cut at z=0, and only the z>0 section kept.

A triangle projects to a triangle. The projectors of a triangle all lie in an infinitely high tetrahedron, and the intersection of an infinitely high tetrahedron and the projection plane, in the non-infinite direction is always a triangle.

In a similar line of thought, the set of all projectors of a sphere form a cone. The intersection of the cone with the projection plane can form any conic. Namely, a hyperbola, an ellipse or a circle. If the sphere contains the origin, the projection fills the whole projection plane.

Other applications

By not losing sight of the idea behind the projection, one can accomplish much more than what has been just described. One example is texture mapping. Often, a polygon will be drawn on screen, but some properties of the polygon (say color for example) changes across the polygon in 3d space. When this happens, we want to know what point from the polygon we are currently drawing. An application of this is texture mapping.

Texture mapping involves taking the point on screen, finding the projector that goes through it and finding the intersection of that projector with the polygon. We then have a point in 3d space. However, it is usually much more useful to make a local 2d coordinate system for the plane containing the polygon and make the property a function of the location in that 2d coordinate system. This is what I did below in the snapshot of the screen from my math software.

Let (u,v) be the coordinates on the projection plane, (p,q) the coordinates on the projected plane, (Xp,Yp,Zp) and (Xq,Yq,Zq) the two vectors defining the plane, (A,B,C) the origin of that plane and the projection x=k1*z*u, y=k2*z*v. Then, the intersection of the projection ray and the projected plane in the projected plane's coordinate system (p,q) in function of the projection plane coordinate system (u,v) is:

$$z := p \cdot Zp + q \cdot Zq + C$$

$$k1 \cdot z \cdot u = p \cdot Xp + q \cdot Xq + A$$

$$k1 \cdot (p \cdot Zp + q \cdot Zq + C) \cdot u = p \cdot Xp + q \cdot Xq + A \qquad \text{Equation A}$$

$$k2 \cdot z \cdot v = p \cdot Yp + q \cdot Yq + B$$

$$k2 \cdot (p \cdot Zp + q \cdot Zq + C) \cdot v = p \cdot Yp + q \cdot Yq + B \qquad \text{Equation B}$$

Now, let's solve for p then for q.

$$k1 \cdot (p \cdot Zp + q \cdot Zq + C) \cdot u = p \cdot Xp + q \cdot Xq + A$$

$$p = \frac{-(k1 \cdot u \cdot q \cdot Zq + k1 \cdot u \cdot C - q \cdot Xq - A)}{(k1 \cdot Zp \cdot u - Xp)}$$

$$k2 \cdot (p \cdot Zp + q \cdot Zq + C) \cdot v = p \cdot Yp + q \cdot Yq + B$$

$$q = \frac{-(-k2 \cdot v \cdot Zp \cdot A + k2 \cdot v \cdot C \cdot Xp - Yp \cdot k1 \cdot u \cdot C + Yp \cdot A + B \cdot k1 \cdot Zp \cdot u - B \cdot Xp)}{(-k2 \cdot v \cdot Zp \cdot Xq + k2 \cdot v \cdot Zq \cdot Xp - Yp \cdot k1 \cdot u \cdot Zq + Yp \cdot Xq + Yq \cdot k1 \cdot Zp \cdot u - Yq \cdot Xp)}$$

$$\frac{((Zp \cdot A - C \cdot Xp) \cdot v \cdot k2 + (Yp \cdot C - B \cdot Zp) \cdot u \cdot k1 + B \cdot Xp - Yp \cdot A)}{((-Yp \cdot Zq + Yq \cdot Zp) \cdot u \cdot k1 + (-Zp \cdot Xq + Zq \cdot Xp) \cdot v \cdot k2 + Yp \cdot Xq - Yq \cdot Xp)}$$

Similarly,

$$p = \frac{(k1 \cdot u \cdot Zq \cdot B - k1 \cdot u \cdot C \cdot Yq + Xq \cdot k2 \cdot v \cdot C - Xq \cdot B - A \cdot k2 \cdot v \cdot Zq + A \cdot Yq)}{(-k2 \cdot v \cdot Zp \cdot Xq + k2 \cdot v \cdot Zq \cdot Xp - Yp \cdot k1 \cdot u \cdot Zq + Yp \cdot Xq + Yq \cdot k1 \cdot Zp \cdot u - Yq \cdot Xp)}$$

$$\frac{((Zq \cdot B - C \cdot Yq) \cdot u \cdot k1 + (Xq \cdot C - A \cdot Zq) \cdot v \cdot k2 + A \cdot Yq - Xq \cdot B)}{((-Yp \cdot Zq + Yq \cdot Zp) \cdot u \cdot k1 + (-Zp \cdot Xq + Zq \cdot Xp) \cdot v \cdot k2 + Yp \cdot Xq - Yq \cdot Xp)}$$

As can be quite plainly seen above, the equations for p and q above are of the form:

$p=(Du+Ev+F)/(Au+Bv+C)$

$q=(Gu+Hv+I)/(Au+Bv+C)$

Notice that the denominator is the same for both p and q. The values for the coefficients A through I can be found by examining the snapshot of the math software screen above.

Other applications can also be found to the theory. A popular application is for the rendering of certain types of space partitions, popularly referred to as voxel spaces. Start with a short vector in the direction you want to shoot the light ray, and start at the eye. Move in short steps in the direction of the light ray until you hit an obstacle, and when you do, color the screen point with the color of the obstacle you hit.

Basically, everything flows from the idea of this projection.

Reality strikes

In reality it is impossible to shoot enough projectors through points to cover any area of the projection plane, no matter how small. The compromise is to accept an error of about one pixel, and shoot projectors only through pixels. This means you might entirely miss things that project to something smaller than a pixel, or incorrectly attrubute them a whole pixel. These details become important in quality rendering. In that case, steps have to be taken to ensure that sub-pixel details have some form of impact on the global outlook of the image. Different techniques can be used which will not be described here.

Another thing we're going to do is only project the vertices of lines and polygons and use the theorems we found earlier to figure out the aspect of the projected object. For example, when projecting a triangle, the projection is the triangle that passes through the projection of the vertices of the unprojected triangle. However, these projected vertices will very likely not fall on integer pixel values. In this case, you have the choice of either rounding or truncating to the nearest pixel, or taking into account sub-pixel accuracy for vertices in your drawing routine. The former can be easily done, the latter is a much more involved topic which will not be discussed.

The state of things as they are at the moment of this writing makes the texture mapping equations a bit too expensive at 2 divisions per pixel. On most processor today, division is usually significantly slower than multiplication, and multiplication itself is significantly slower than addition and subtraction. This is expected to change in the near future however. In the mean time, one can use interpolations instead of exact calculations. These are discussed in the next section.

Interpolations and approximations

Introduction

Frequently in computer graphics, calculations require too much processing power. When this problem arises, many solutions are at hand. The most straightforward solution is to completely forget about whatever causes the lengthy calculations. However, that might not be satisfying. The second most straightforward solution, in a certain sense, is to get faster hardware and contend with slower image generation. That still might not be satisfying. If this is the case, the only solution left to us is to approximate.

Generally speaking, given a relatively smooth function of x over a finite range, it is usually possible to approximate it with another, easier to compute function over the same range. Of course, this will generate some form of error. Ideally, we should pick the approximating function as to minimize this error while conforming to whatever constraints we may impose. However, minimizing error is not always straightforward, and it is also usually preferable to find a good approximating function quick than the best approximating function after complicated computations. (In the latter case, we might as well not approximate.) Error computation is a rather complicated topic, and I do not wish to get involved with it in here. For the more formally oriented reader, one popular definition of error between f(x) and g(x) is $\int(f(x)-g(x))^2dx$, the integral is to be taken over the interval over which g(x) is to approximate f(x).

One of the more popular type of approximating functions are polynomials, mainly because they can usually be computed incrementally in a very cheap and exact manner. Fourier series are generally not useful because trigonometric functions cannot be computed very quickly. A very nice way of generating an approximating polynomial is to use the Taylor series of the function we want to approximate, assuming we have an analytical form of the said function.

Polynomials will be used to approximate everything from square roots to texture mapping to curves.

Forward differencing

Forward differencing is used to evaluate a polynomial at regular intervals. For example, given the polynomial y=a*x+b, which is a line, one might want to evaluate it at every integer value of x to draw a line on screen.

We must of course initially compute y(0)=a*0+b, or y(0)=b. But then, we can exploit *coherence*. Coherence is something that occurs just about everywhere in computer graphics, and exploiting it can tremendeously cut down on the computations.

The next value we are interested in is y(1). But, y(1)=y(0)+y(1)-y(0). (Notice that the y(0)'s cancel out). However, y(1)-y(0)=a. Thus, y(1)=y(0)+a. Furthermore, y(2)-y(1)=a, thus we can add a to y(1) to find y(2) and so on. Generally speaking, y(n+1)-y(n)=[a*(n+1)+b]-[a*n+b]=a.

This extends to higher order polynomials. As an example, let's do it on a second degree polynomial, and in a more formal manner. We will suppose a step size of k instead of 1 for more generality, and the following generic polynomial:

$y=Ax^2+Bx+C$

First, let's find y(n+k)-y(n) as we did before:

$y(n+1)-y(n)=[A(n+k)^2+B(n+k)+C]-[An^2+Bn+C]$

$=[An^2+2kAn+Ak^2+Bn+kB+C]-[An^2+Bn+C]$

$=2kAn+Ak^2+kB$

Let's call that last result dy. Thus, y(n+1)=y(n)+dy(n). Now the problem is evaluating dy(n). However, dy(x) is itself a polynomial (first order; a line), so forward differencing can also be applied to it. We thus need dy[n+1]-dy[n], which is simply 2kA.

Concretely, this is what happens. Let's say we have the polynomial x^2+2x+3 with a step size of 4 over the range 3-19, inclusively. We thus have A=1, B=2, C=3. We calculated dy(x) to be $2kAx+Ak^2+kB=2*4*1*x+1*4^2+4*2=8x+24$.

First of all we calculate the initial values for y and dy, which are y(3)=18 and dy(3)=48. The incremental value for dy is 2kA=8. Then, we proceed as follows:

Value of...

x	y(x)	dy(x)
3	18	48

(as initially calculated - now add dy(x) to y(x), 8 to dy(x) and 4 to x)

7	66	56

(once more, add dy(x) to y(x) and 8 to dy(x) and 4 to x)

11 122 64

(etc...)

15 188 72

19 260 80

These can be trivially extended to multi-variable polynomials.

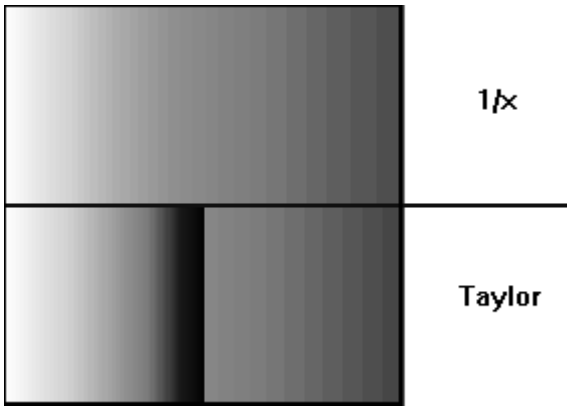
Approximation function

Finding the approximation function is the real problem. When trying to approximate a function, we usually want to minimize error measure in some specific way. However, sometimes additionnal contraints have to be taken into account. For example, when interpolating values across polygons, care should be taken that they are interpolated in a way that does not cause too much contrast and/or mach banding along the edges shared by more than one polygon.

One of the more obvious ways of generating an approximation polynomial is to make a Taylor series expansion of whatever function you want to approximate. This, however, does nothing about the edge constraint we just mentionned. However, Taylor series do just fine, for example, for normalizing vectors that are supposed to be normal but due to error buildup aren't. The problem with normalizing a vector is that vector (a,b,c) has a norm of $N=(a^2+b^2+c^2)^{1/2}$, and that the normalization of (a,b,c) is 1/N*(a,b,c). Usually, extracting a square root is very long, and a division is also longer than a multiplication. It would be nice if we could find an approximation of 1/sqrt(x) and multiply by that instead. Actually, the two first terms of the Taylor series expansion of 1/sqrt(x) about 1 are:

$1/\sqrt{x} \cong (3-x)/2$ (around $x \cong 1$)

The division by two can be accomplished with a bit shift, and subtraction is usually fairly fast on any CPU. Using $x=a^2+b^2+c^2$, we can find 1/sqrt(x) much faster than otherwise.



The picture above demonstrates what happens when one approximates a value that varies smoothly across faces with a Taylor series. The upper half of the picture shows a square for which the intensity of a pixel (x,y) is 1/x. The leftmost pixels have x=1 (intensity 1), and the rightmost pixels have x=3 (intensity 1/3). The lower half shows two Taylor approximations. The first Taylor series expansion was done around x=1, the Taylor polynomial is thus $4-6x+4x^2-x^3$. This corresponds to the lower left square. As can be seen, near the left edge, the Taylor series is nearly perfect. Near x=2, though, it goes haywire. The bottom right square is a Taylor series expansion about x=2 (the polynomial is $2-3/2x+1/2x^2-1/16x^3$). As can be seen, it is much closer to the real thing, but that's only because 1/x becomes more and more linear after that point. But things that are linear after the perspective transform are the exception rather than the rule.

The moral of this story is that if two faces are next to each other, and that the shading (or any other property) is really a continuous function, but we approximate it using Taylor series about arbitrary points, it is very easy to get something that does not look continuous at all.

Thus, it would be unwise to do a Taylor series expansion of texture mapping equations, or Phong shading and the like. Note that a property that varies with 1/x is not a rare thing in computer graphics because of the perspective transform, thus the example is very relevant.

Rendering

Introduction

Rendering is the phase where we do the actual drawing. There is a general tendency to download this particular task to a slave graphics processor and leave the CPU to do better things. However, it will always be useful for everyone to have a general understanding of how things work. And also likely is the fact that we're going to need software renderers for a while more. And one last fact is that people have to write the software for the slave processor.

We will first study the drawing of a point, which will be used to draw other primitives. Then we will study lines and polygons. Curved surfaces can also be supported but will not be discussed. The curved primitive that tend to be faster in drawing are ellipses and polynomials. However, some other forms of curved primitives definitions are often preferred, mainly splines.

The point

A geometric point is a 0 dimensional object. It could also be defined very strictly with neighborhoods and somesuch. However, this is not particularly useful to the computer graphics specialist. One thing that we must remember, though, is that it is impossible to display a point on any medium. Quite simply, a point has a size of zero, no matter if the definition of size is length, area or volume. It cannot be seen under any magnification.

What the computer graphics expert usually refers to the point is the smallest entity that can be displayed on the display device. These are not necessarely circular or rectangular things - and more often than not, they are slightly blurred.

We will refer to this point as a *pixel*. We will also need to make a few basic assumptions. Generally speaking, pixels are of an arbitrary shape (often rectangular-like), and are aligned in a very structured way on the display device. Note that some devices do not use this method of displaying things, these are commonly referred to as vector devices. There was an old Star Wars (trademark of LucasArts I beleive) game made with one of these.

We will also very much like pixel to be aligned in a cartesian plane like manner. We generally assign pixels integer position, and what's not exactly on a pixel has a noninteger position. But what is the position of the pixel? That's another entirely arbitrary matter. Generally speaking, we might want to simplify the pixel's location to its centroid. Also, there is the problem of axis orientation. For a combination of arbitrary and historical reasons, the screen coordinates origin is very usually centered on the upper left corner and goes positively down and right in hardware. Operating sometimes hide that from the user application and use a coordinate system centered on the lower left corner, and go positively up and right, just like the usual cartesian plane.

The way that pixels are stored internally is also of importance. Generally speaking, each pixel is assigned a color, and the number of colors available per pixel is defined by the number of bits allocated to each pixel. For instance, if each pixel has 6 bits of color data to it, then each pixel can be one of 64 colors. When an odd number of bits per pixel is used, it often happens that the bits are spread in a less intuitive way. For example, in the 6 bit case, instead of using one byte per pixel and wasting 2 bits per pixel, the system will likely store bit 0 of all pixels, then bit 1 of all pixels, then bit 2 of all pixels, and so on. This is called a bit-plane arrangement.

If the number of bits per pixel is closer to 8, 16, 24 or 32, then some systems will allow what is generally referred to as a linear arrangement of pixels. For example, if 8 bits are allocated per pixel, then one byte corresponds to one pixel.

It is generally accepted that with 24 bits per pixel, the human eye cannot see the difference between two very similar shades of the same color. However, 24 bits per pixel is roughly 16 millions of colors. Thus, when using a mere 6 or 8 bits per pixel, sacrifices must be made.

One way that the industry has found is to make a look-up table. As an example, each pixel is assigned a value from 0 to 255 (for 8 bits), and the hardware is given a lookup table of 256 color entries. Each color entry can contain a 16 or 24 bit color for example. Then, the hardware automatically substitute the proper entry in the table for each pixel when it has to display them. The lookup table is often referred to as the *palette* or *color map*, the last term being slightly more appropriate in my opinion.

At any rate, eventually one needs a way of identifying colors, either to select the color map colors, or in the case of a 24 or 32 bits system, select the pixel color. There are several ways of doing this. Two of the most popular means are RGB colors. RGB is an acronym for red, green and blue. Colors are specified by their red, green and blue contents. It is interesting to note that displays usually do not use red, green and blue, but colors that are close to these. The colors actually used by the displays were selected to allow for a broader range of colors.

Another popular means of selecting a color is with HSV values. This is another acronym that stands for Hue, Saturation and Brightness (isn't that last one obvious?) This model is more intuitive than the RGB model for most people. Other models include the CMY (Cyan, Mangenta and Yellow) and YIQ (Luminance and Chromaticity) models.

Writing to a specific pixels usually involves finding an address and then putting a value in it. Since memory is usually mapped in a one dimensional fashion, device pixels are mapped in an arbitrary way to memory. Usually, finding a memory location for a pixel involves a multiplication. However, multiplications are typically expensive, thus we might want to look into that a bit further. Let us assume a 1024x768 display, with 8 bits per pixel and linear mapping.

A base address A has to be given for the top-leftmost pixel (assuming origin is at top-left). Then, the first row of 768 pixels would be the next 768 bytes. Then next row of 768 pixels would follow and so on. Generally speaking, pixel (x,y) for an integer x and y can be found at memory location A+x+y*768. Note that some systems will want to pad each row with a few bytes sometimes.

Multiplying once per pixel write is a bit expensive. There are several workarounds. The first one is straightforward but hardware dependant. The second one assumes that we access pixels in a coherent way (not totally random).

If the width of the display device in pixels is known in advance, the multiplication can be removed the following way. Say the pixel to be addressed is at memory location A+x+y*768. 768 is 512+256, thus we have A+x+y(512+256) = A+x+512y+256y. However, $512=2^9$ and $256=2^8$. Thus, the pixel memory location is $A+x+2^9y+2^8y$. Note that a multiplication by a power of two can be optimized out with left shifts, which are typically much faster than multiplications. Let a<<b denote a shifted left by b bits, then the memory location of the pixel can be expressed as:

A+x+y<<9+y<<8.

Similar decompositions in powers of two of various scalars can be found. The problem with this is that it requires the scalar (display device width in pixels) to be hard-coded in the program.

Another way of accessing pixels is by exploiting coherence. If we plane to access all the pixels on a given scanline, starting from left to right, then the following is true.

Pixel at memory location A+y*width is the leftmost pixel on the scanline. The second leftmost value is the above value plus one. The third one can be found by adding one again, and so forth. Thus, accessing pixels that are adjacent on a scanline can be done in one add per pixel only.

Accessing pixels that are on the same column is similar, except that <width> is added each time to the memory location.

Lines

There are a number of ways to draw lines. Generally, they all boil down to the same basic ideas, and have roughly comparable speeds. The algorithm presented here is the one I felt had the best mixture of simplicity, efficiency and small size. It has the disadvantage of being less exact than some other algorithms for lines of rational slopes. We will first start with special cases, then move to more general cases.

The simplest lines to draw are the horizontal and vertical ones. As can be imagined easily by the reader from the last section, we start with the topmost or leftmost pixel, draw it, then either add 1 or <width> to memory location and draw the next pixel. And so on, for the length of the line.

The next step up is an angled line. If the line is not vertical nor horizontal, then it can be expressed as y=mx+b or x=ny+c, with n=1/m and c=-b/m, whichever is preferred. The representation one has to use is whichever of the two has the smallest m or n. This is to ensure that there are no large gaps between the pixels of a line. Afterwards, we initialize (x0,y0) to be one endpoint of the line. If we chose y=mx+b, we should be using the leftmost endpoint for (x0,y0). We draw (x0,y0), then increment x, and add m to y. Then we draw the new point. The extension to the x=ny+c form is left to the reader.

Notice that the previous paragraph is simply an application of forward differencing studied previously. The witty reader will easily imagine the extension of this to higher degree polynomials. It is also possible to extend incremental algorithms to circles and ellipses, amongst others, but we will not go into this.

In some applications, such as polygon drawing, one of either y=mx+b or x=ny+c will be prefered even if the slope is greater than 1.

Note that the topic of line drawing can be extended much more than this. Anti-aliased lines, line width and patterns, endpoint shape are all topics that we will not cover.

Polygon drawing

Let us first define a few terms, in an intuitive and geometric fashion. A polygon is, as can be seen above, a 2d object with area, delimited by edges. The edges are line segments, and there is a finite number of edges.

Polygons that do not self-intersect can be said to be either convex or concave. The polygon above is self-intersecting. A convex polygon is one for which the inside angle at any vertex is less than or equal to 180 degrees. All other polygons are said to be concave. Sometimes, it is said that a particular vertex is concave, which is not entirely correct, but rather means that the inside angle at that vertex is more than 180 degrees.

What interests us most is filled primitive. It is relatively easy to draw a wire frame polygon using only line drawing routines described previously (hidden line removal then becomes a problem).

The star-shaped polygon shown above is very interesting to us because it exhibits the more interesting properties we want our polygons to have. The grayed areas are considered to be inside the polygon, where the white areas are outside the polygon. This means that the inner pentagon is considered to be outside. The rule for determining whether a point lies inside or outside the polygon is as follows.

To determine if a point lies in or out of a polygon, draw a line from that point to infinity (any direction, far far away). Now find the number of times that line intersects the polygon edges. If it is odd, the point is in, if it is even, the point is out. It is recommended that you try this with the star above and that it works no matter what point you pick and no matter what direction you draw the line in.

The basic idea of the line polygon drawing algorithm is as follows. For each scanline (horizontal line on the screen), find the points of intersection with the edges of the polygon, labeling them 1 through n for n intersections (it is of note that n will always be even except in degenerate cases). Then, draw a horizontal line between intersections 1 and 2, 3 and 4, 5 and 6, ..., n-1 and n. Do this for all scanlines and you are done.

Probably, you might want to restrict yourself to scanlines that are actually spanned by the polygon. Also, there are a few things to note.

If the polygon is convex, there will always be only one span per scanline. That is generally not true for concave polygons (though it can accidentally happen).

Here is pseudocode for a polygon filling algorithm.

```
Let an edge be denoted (x0,y0)-(x1,y1), where y0≤y1. Edges also
have a "current x" value, denoted cx. Initialize cx to x0. One
should also compute the slope of all edges, denoted s, which is
(x1-x0)/(y1-y0) (we are using the x=ny+c representation).
Let IET be the inactive edge table, initially containing all edges
Let AET be the active edge table, initially empty

Sort the IET's edges by increasing values of y0
Let the initial scanline number be the y0 of the first edge in the
IET

Repeat
      While scanline≥y0 of the topmost edge in the IET
            Move topmost edge from IET to AET
      End while

(*)   Sort AET in increasing values of cx

      For every edge in the AET
            If edge's y1≥scanline, then remove edge from AET
            Else add the slope "s" to "cx".
      End for

      For each pair of edge in the AET
            Draw a horizontal segment on current scanline between
            column "cx0" and "cx1", where "cx0" is the "cx" value
            for the first edge in the pair and "cx1" is the "cx"
            value for the second edge in the pair
      End for
Until the AET is emtpy
```

It is of note that the line marked by (*) can be optimized out. If the polygon is not self-intersecting, we just need to make sure the AET is properly sorted when we insert a new edge into it.

It should be noted that edges that are parallel to the scanline should not be put in the IET. You might also need to clip the polygon to the viewport, which can be added to the polygon blitting code.

Visible surface determination

Introduction

One of the problems we have yet to address, when several objects project to the same area on screen, how do you decide which gets displayed. Intuitively, the closest object should be the one to be displayed. Unfortunately, this definition is very hard to handle. We will usually say that the object to be displayed will be the one with the smallest z value in eye space, which is a bit easier to work with. A corollary of this is that objects with the largest 1/z value get displayed, this latter observation has applications which will be explained later.

Visible surface determination can be done in a number of ways, each has its advantages, disadvantages and applications. Hidden line removal is used when wire frames are generated. This might be useful for a vector display, but will not be covered in here. When dealing with filled primitives, there are several classes of visible surface determination. There is also the question of object precision, device precision, and more, these topics will not be discussed here.

Perhaps the most intuitive visible surface determination algorithm is the so-called "painter's algorithm", which works the same way a painter does. Namely, it draws objects that are further away first, then draws objects that are closer. The advantage of this is it's simple. The disadvantages are that it writes several times to some areas of the display device, and also that some objects cannot be ordered correctly.

The painter's algorithm can be generalized into the depth-sorting algorithm, which sort the primitives from back to front and then draw. The depth sorting algorithm also resolves cases that painter's algorithm does not.

Another algorithm is space partitioning trees such as BSP (binary space partitions) trees. The advantage of this algorithm is to generate a correct ordering of the primitives quickly and correctly no matter where the viewer is. The disadvantage is that the object has to be static (e.g. no stretching), but still can move and rotate in space. It is also quite hard to merge two BSP trees together. Approximations can be made.

Yet another way of doing visible surface determination is the class of algorithms generally referred to as "scan-line algorithms". These algorithms, though somewhat slower than depth sorting, have the advantage of drawing to each and every pixel of the display device once and only once. Thus there is no need to clear the display in the first place, and pixels are not written to needlessly. Incidentally, this algorithm is very useful for display devices where it is impossible or difficult to erase or rewrite to pixels, such as printers. The disadvantages are that it's slightly slower, and usually quite more messy to code than a depth sorting algorithm. Also, visible surface determination becomes an integral part of the polygon drawing routine in most cases, making it hard to download the polygon drawing code to some hardware, or to make several versions of polygon drawing code for different drawing modes.

A very popular way of doing visible surface determination is called z-buffering. This works by storing the z value whatever occupies a pixel for each pixel. This way, one can add new primitives to a scene, visible surface determination is just a compare away. Incidentally, it is usually much more efficient to use 1/z values than it is to use z values, since 1/z varies linearly but z does not.

Another algorithm worth mentioning is the Weiler-Atherton algorithm, which clips primitives so that they do not intersect before drawing, and Warnock's algorithm, which recursively subdivides the display area until it can trivially determine which primitive to draw. These algorithm are fairly slow.

An optimization that can be made to any visible surface determination algorithm is back-face removal or back-face culling. This is based on the observation that faces that are facing *away* from the observer

As of now, the only algorithms discussed will be the depth sort and painter's algorithm, along with z buffering and back-face culling.

Back-face culling

Back-face culling exploits the observation that a face in a closed polyhedron always has two sides. One faces inside, and can never be seen by an observer outside the polyhedron (rather obviously since the polyhedron is closed), the other faces outside and can be seen. However, if it is determined that the side facing the eye is the inside of the face, that face will assuredly not be seen, because it is impossible to see a face from the inside.

The side that faces the eye can be determined easily with dot product. Take a vector V from the eye to any point withing the polygon (for example, from the eye to a vertex). Let A be the normal of the polygon, assuming that A points outwards of the polyhedron. Then, compute V•A. If it is positive, the inside of the face is towards the camera, do not display or transform the face. If it is negative, the face is facing the camera and might be seen (though this is not guaranteed).

Back-face culling is generally not sufficient for visible surface determination. It is merely used to remove faces that assuredly cannot be seen. However, it will do nothing for faces that are only obscured by faces that are closer. Also, back-face removal assumes that the objects are closed polyhedra, and that faces are opaque. If this is not the case, back-face culling can not be applied.

Note that if the only thing displayed is a convex object, back-face culling is sufficient for visible surface determination (it will only leave the faces that are actually visible).

Also note that back-face removal should be done in *object* space, not in world or eye space. That's because, in order to do it in world space, one has to transform all plane normals before doing the dot product, which is rather expensive. However, if performing the culling in object space, one only needs the location of the eye in object space, and normals need not be transformed.

It can be shown that back-face culling is expected to cull roughly half of the number of vertices, faces and edges in a scene, except for special scenes that are made to be viewed from a particular angle or somesuch.

Sorting

With the painter's algorithm, one has to assign a z-value to all primitives. Then, the primitives are sorted according these values of z, and the resulting image is drawn back-to-front. Several sorting algorithms can be used for this purpose, and even though basic algorithmics is not the purpose of this document, we will discuss two simple sorting schemes now.

The simplest sorting algorithm, and a frightfully slow algorithm in most cases, is the bubble sort. Here follows pseudocode for the bubble sort.

```
Let z[1..n] be the array of n values to sort
Let f be a flag

Repeat
      Clear f
      For i varying from 1 to n-1
            If z[i]>z[i+1] then
                  Set f
                  Exchange z[i] and z[i+1]
```

```
            End if
      End for
Until f is clear
```

As can be seen, the algorithm is exceedingly simple. For small values of n (say, n<10), this algorithm can be used and will be close to optimal. However, if the list is very badly ordered initially, the sort could take up to n^2 iterations before finishing.

Small improvements can be made to the algorithm. For one thing, instead of always scanning in the same direction (from the first element to the last), one alternates directions, sorting an already close to sorted list is very efficient. The loop will execute roughly n times (actually, it would execute k times n, where k is some small constant). In the worse case though, it still executes in n^2 iterations.

A second, more clever algorithm that works well on numbers, is the bucket sort, or radix sort. This sort can be done in any base (useful bases for a computer would be 2, 16 or 256, because they're powers of two). However, for the sake of simplicity in this example, we will use base 10.

Using base n, n buckets are created (in our example, 10 buckets), labeled 0 through n-1 (0-9 in our example). Then, the numbers to be sorted are put in the bucket that corresponds to their lower digit. The buckets are concatenated, and the step is repeated for the next lower digit. An so on, until we get to the highest digit, at which point we stop. The result is a sorted list. Pseudocode is given below for base n. Note that the i^{th} digit of base n number z is $(z \text{ div } n^i)\%n$ where div stands for integer division, truncating off any fractions, and % is the modulo operator, or remainder after division by n (a value from 0 to n-1 inclusive).

```
Let b[0..n-1] be n buckets, labeled 0 through n-1
Let z[1..m] be the m numbers to sort
Let D be the largest number of digits used

For foo varying from 0 to D-1 inclusive, do
      For i varying from 1 to m inclusive, do
            Put z[i] into its bucket, namely b[(z[i]/n^foo)%n]
      End for
      Concatenate all buckets, in order from 0 to n-1, back into z
End for
```

Note that division and modulo operations, when done with base two divisors, can be implemented strictly with bit shifts.

This algorithm can be implemented with lists or arrays. Lists ensure that no unnecessary copying is done, and allow buckets to grow dynamically. This is not so easily accomplished with arrays, but the pseudocode below essentially does that. It only needs to be repeated for every byte in the numbers to be sorted.

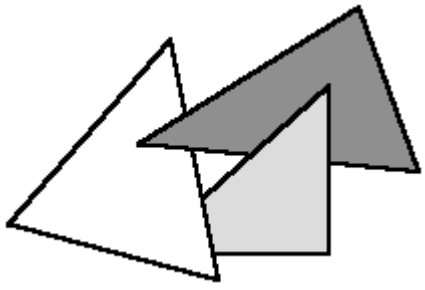
```
Let i[0..256] be 257 indices, initialized to 0
Let z[1..m] be the m numbers to sort
Then o[1..m] will be the m numbers once concatenated
```

```
Comment: The first step we take is to count the elements that will
go into each bucket

For j varying from 1 to m inclusive, do
      Let foo be the bucket to which z[j] belongs
      Increment i[foo]
End for

Comment: now compute the index at which buckets start

For j varying from 1 to 255 inclusive, do
      Add i[j-1] to i[j]
End for

Comment: lastly, put the numbers into the bucket and concatenate

For j varying from 1 to m inclusive, do
      Let foo be the bucket to which z[j] belongs
      Put z[j] into o[i[foo]]
      Increment i[foo]
End for
```

Other sorting algorithms that might be of interest are the quick sort, heap sort and insertion sort. These will not be discussed here, they each have their advantages and drawbacks.

Painter's algorithm and depth sorting

As was previously mentionned, painter's algorithm assigns a z value to each primitive, then sorts them, then draws them from back to front. Objects that lie behind are then written over by objects that lie in front of them. Note that, no matter the scheme used to select the z value for an object, primitives that have overlap in z may be incorrectly ordered. But there is worse. Note the pathological case below, where it is impossible to generate a proper ordering for the three triangles:



In this case, it is necessary to cut one triangle into two parts and sort the parts individually.

A way of handling all cases is as follows. Assign a z value to all polygons equal to the vertex belonging to the polygon that has the largest z coordinate value in eye space. Then sort as per painter's algorithm. Before actually drawing, we need to do a postsort stage to make sure the ordering is correct for polygons that have z overlap.

Assuming we sorted in increasing values of z, it means that we need only to compare the last polygon with the consecutive previous polygons for which the furthest point is in the last polygon's z span. Once the last polygon is processed, we will not touch it anymore (unless the last polygon is moved to some other position in the list). Thus, we just consider the list to be one element shorter and recurse the algorithm.

The steps that should be taken are as follow (P and Q are the polygons we are comparing).

1- Check wether the polygons x and y extent overlap on screen. If they do not, there is no need to compare the polygons. Otherwise, we are undecided (go to 2)

2- Check on what side of P's plane the eye lies. If Q lies entirely on that side of P's plane, Q is considered to be in front of P. If Q lies entirely on the opposite side of the eye in relation to P's plane, then P is in front of Q. If Q crosses P's plane, we are still undecided.

3- Repeat 2 above, but with Q and P inverted.

4- Check if the polygons overlap on screen (find wether the edges of the polygons intersect)

Once a polygon has been moved in the list, mark it so that it is not moved again. If one of the above steps would say that a polygon that has already been moved in the list should be moved again, then you will have to use the last resort, clipping.

Of course, one needs not to perform all these tests if they are deemed to be more expensive than clipping. For instance, the only tests one could do is test for overlap in z, then x and y on screen, then check for step 2 and if it is still unresolved, simply clip the polygons and put the pieces where they belong.

When polygon ordering can not be resolved, pick one of the two polygons for clipping plane and clip the other polygon with it. Then, insert the two pieces at the appropriate positions in the list.

A very nice way of doing all these tests is as follows. Calculate bounding boxes for z value in 3d, and u,v in 2d (screen space, after perspective transform) of the polygon. Then, sort the bounding boxes in x, u and v. This can be done in linear time using the radix sort algorithm (or by exploiting coherence in a comparison sort algorithm). Then, only the polygons for which the bounding boxes overlap in all three axis need to be checked further.

Z-Buffering

This algorithm tends to be slightly slower than painter's algorithm for low number of polygons (less than 5000). It would appear that it would gain as the numer of polygons increases though.

The idea is to make an array of z or 1/z values, one for each pixel. As you draw a polygon, compute the z or 1/z value at a pixel, compare it with the current value for the pixel, and if closer, draw, otherwise, do not draw.

This algorithm has the advantage that it requires no sorting whatsoever. However, the algorithm performs one comparison per pixel, which tends to be a bit expensive. Also, memory requirements tend to be bigger than other algorithms. Nevertheless, the simplicity of the implementation makes it very attractive.

As a side note on evaluating z or 1/z, the latter can be shown to be linear while the former is not. Thus, it will likely be preferable to store 1/z values instead of z, because they can typically be computed much more quickly. The mathematics of that are shown below.

Let Ax+By+Cz=D be the plane equation for the polygon in eye space. Let (u,v) denote the pixel on screen, and let the perspective projection equation be u=px/z and v=qy/z for some constants p and q. This can be rewritten as:

x=uz/p y=vz/q

Auz/p+Bvz/q+Cz=D

z(Au/p+Bv/q+C)=D

And then, depending wether we are interested in z or 1/z, we get:

z=D/(Au/p+Bv/q+C)

or

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

The latter can be rewritten as

1/z=Mu+Nv+K

M=A/(Dp), N=B/(Dq), K=C/D

Thus, 1/z varies linearly across the (u,v) plane of the display device. When forward differencing is applied, calculations for values of 1/z are reduced to one add per pixel, with a small setup cost.

Lighting models

Introduction

After all we have covered, we still have to decide how much light gets reflected off things and such, and how it gets reflected. For example, some objects that are facing towards a light source will appear much more bright, perhaps with a very brilliant spot somewhere, than objects facing away from it. Objects also cast shadows, which are much harder to compute. We might also want to somehow take into account that a certain quantity of light bounces off everything and lights up things equally from all directions.

Furthermore, we might want to vary the intensity of light across a given polygon, especially if these polygons are big. If not, one gets a somewhat ugly effect called mach banding where the contrast between faces gets amplified by our brains. This will raise the question of how the light should vary across a polygon, and why.

The first part discusses actual lighting models, and the second discusses shading algorithms for rasterization of nonuniformly shaded polygons.

Lighting models

The most basic idea we can have is to make light intensity a function of the angle between the direction of the light rays from the light source to a point, and the normal to the point. This is called diffuse lighting. This means that light reflects off the face equally in all directions, so the direction in which the eye is is not relevant.

Looking back on the vector algebra chapter, we had a definition of angle with the dot product. This is written as:

$\text{Cos}\theta = A \bullet B / (|A| \times |B|)$

If A is the plane normal (of unit length), then $|A|$ is 1 and can be removed from the equation. B would be the vector from light source to point to be lit. Then, we make light intensity a function of $\text{Cos}\theta$, which is calculated to be $A \bullet B/|B|$, which is fairly easy to calculate. Note that if θ is less than $\pi/2$, it means that the face is actually facing away from the light source, in which case it should not receive any light from that light source. This can be recognized when $A \bullet B/|B| > 0$.

Usually, one makes the intensity of the light received from a light source $A \bullet B/|B|$ times some negative constant (since positive values of $A \bullet B/|B|$ mean that the face is facing away and that intensity is then 0).

One might want to take into account the fact that light usually diminishes the further away you are from a light source. Physics say that light intensity is inversely proportional to the square of the distance to the light source. This can be written as $k/|B|^2$, and multiplying that by the value previously given:

$$I = k \times A \bullet B/|B|^3$$

However, as experimentation shows, it is sometimes useful to use some other falloff than square of distance. Generally, people have been using a second degree polynomial of $|B|$. However, if we try the specific case of linear falloff, we get this very interesting simplification:

$$I = k \times A \bullet B/|B|^2$$

If $B=(a,b,c)$, then $|B| = (a^2+b^2+c^2)^{1/2}$. Thus, $|B|^2 = a^2+b^2+c^2$, which eliminates the square root, which is usually the most expensive calculation we have.

The question of what point on the polygon should be used for calculating the vector B from light source to the point on the polygon is answered as follows. Theoritically, B should be recalculated for each point on the polygon. This might turn out to be expensive, and a constant B is then used across the polygon. In this case, however, the $B/|B|^2$ factor should be calculated only once for the whole polygon.

The Phong illumination model also includes a specular component. In that case, a function of the angle between the ideal reflection vector and the eye-to-point vector is added.

These calculations are done on a per light source basis, and should be summed. Ambient light can also be added.

Shadow casting involves more complex calculations which will not be discussed here.

Smooth shading

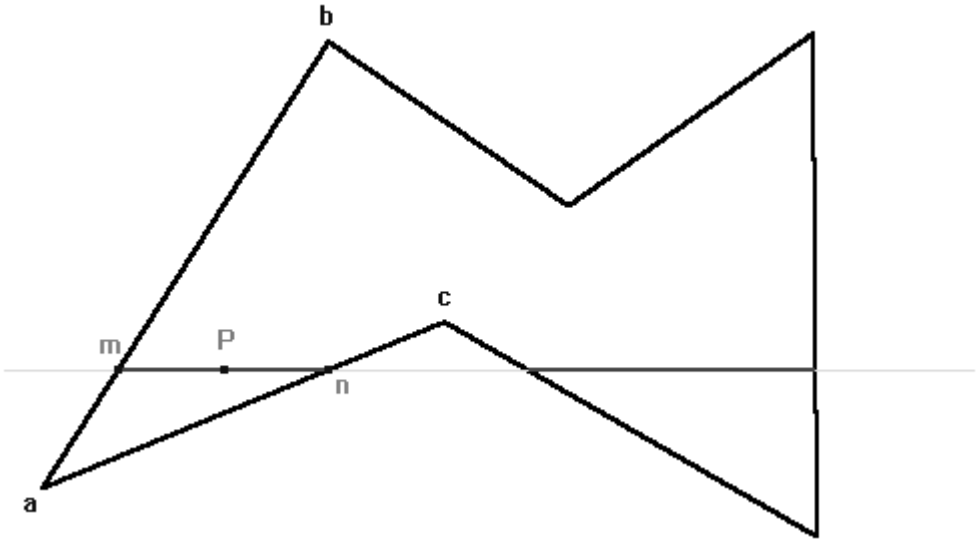
The simplest form of polygon shading calculates one value of intensity and uses that value across the whole polygon.

The other forms of shading require that we first examine our polyhedral model of objects. The assumption we are making is that the polyhedral model is really an approximation to a curved object. Thus, we would like the normal vectors and the shading intensitites to vary smoothly across the surface of the objects, just as it does on a curved surface.

The usual way of accomplishing this is by computing a pseudo normal vector at each vertex. (Keep in mind that a point in 3d has no normal vector, ergo we call it pseudo-normal.) That pseudo-normal per vertex is not the normal of the vertex, but rather the normal we think represents best the curved surface at that point. If we have actual information about the curved surface, we should use that information if we can to compute the pseudo-normal. Otherwise, one good way of doing this is by computing the weighted sum of the faces that touch the vertex. For example, you could sum all normals of the faces that touch the vertex and then normalize. Or, you could make each face's contribution to the pseudo-normal a function of the face's area or the angle made by that face at the vertex, and so on. For ease in calculations, pseudo-normals should be made unit in length.

Then, one can go either one of two ways. The first one is interpolated shading, or Gouraud shading. The second one is Phong shading, which is a bit more complex.

In Gouraud shading, one calculates the intensity of reflected light on the vertices. Then, we linearly interpolate the intensity of the light across the polygon, as shown below. Note that for a n-gon, with n>3, gouraud shading is ambiguous in the sense that it depends on scanline orientation. However, with n=3, the shading is unambiguous. As a matter of fact, given a triangle (x0,y0), (x1,y1) and (x2,y2) and the three intensities at the points, respectively i0, i1, i2, we can view this as three points in 3d (x0,y0,i0), (x1,y1,i1), (x2,y2,i2). Since we are linearly interpolating, and that we have 3 points, then there is only one solution, of the form i=Ax+By+C. Using matrix mathematics, one can find the coefficients A, B and C, and then computations are reduced to one add per pixel with very little setup.



As can be seen above, intensities are calculated for all vertices, particularly vertices a, b and c. Then, intensity is linearly inerpolated between a and b (assuming m is 1/5 of the way between a and b, we'll assign m an intensity of 4/5×a+1/5×b). It is also interpolate linearly between a and c. Then, given the light intensities at m and n, the intensity is interpolated linearly between m and n. Assuming P is midway between these two, then its intensity should be (m+n)/2.

It is easy to demonstrate that no point within the polygon will be brighter or darker than the brightest or darkest vertex, respectively. If a specular highlight should fall within a polygon, Gouraud shading will miss it entirely.

Phong shading works around this the following way. Instead of interpolating the intensity linearly, it interpolates the (x,y,z) values of the pseudo-normals linearly, then normalizes, and the does the lighting calculations once per pixel.

As you might imagine, this is extremely expensive. Many approximations, workarounds and somesuch have been devised. Here we will study one such approximation.

We will interpolate the (x,y) value of pseudo-normals linearly, but we will set $z=(1-x^2-y^2)^{1/2}$. Note that we still have a square root. However, since z is a function of x and y only, and that x and y vary between -1 and +1 only, we can make a lookup table for z, which makes it a lot faster. Then we can do the lighting calculations. However, this is still a bit slow. If we know our light vector to be constant across the screen, then we can optimize it further.

Assuming the light vector is (0,0,1), then the lighting calculations for diffuse shading only is (x,y,z)•(0,0,1). This simplifies to z, which is $(1-x^2-y^2)^{1/2}$, which is the value we stored in the lookup table. So, interpolate (x,y) linearly and lookup intensity in the lookup table. As a matter of fact, one can even put some other values than $(1-x^2-y^2)^{1/2}$ in the lookup table. These can be used to achieve specular highlights, multiple light sources, or even a nice reflection effect.

Computer graphics related problems

Introduction

In the process of learning computer graphics, one comes across several of the classical questions in one version or another. These include "how do I compute the plane normal of a triangle" or more generally "how do I compute the plane normal of a polygon, preferably using all vertices to minimize error", "how do I make a normal that points outwards" and such.

These technical questions need to be addressed individually, since they typically have very little in common. First will be covered generating normals that point outwards for polygons. An application of that will be covered, which is triangulation of a concave polygon. Computation of a normal for any polygon is then considered, by using all vertices to compute it. Then will be covered the problem of generating plane normals that point outwards of a polyhedron, which relies on edge normals that point outwards of a polygon.

It is of note that the problem of convex decomposition of a concave polyhedron, which has applications in various fields (3d collision detection and visible surface determination) amongst others, is a very complex problem (it has been demonstrated to be NP-Complete) which will not be described herein.

Generating edge normals

It will prove to be essential for the later problems to have normals for the edges that point outwards from the polygon. We might as well start by saying that for an edge of slope m, the normal would be (-m,1) unitized. The second preliminary is defining modulo space addition and subtraction.

Let a and b be integers of modulo n space. Then, a⊕b is defined to be (a+b)%n, where x%y means "remainder of the division of x by y" (the remainder is always positive, between 0 and y-1). Similarly, aΘb is defined to be (a-b)%n. As an example, let's assume we are working in modulo 8 space. Then,

3⊕2=5%8=5 5⊕6=11%8=3 4Θ3=1%8=1 4Θ7=-3%8=5

The first step is to generate normals for all edges by calculating (-m,1) and unitizing it. These normals will not all be oriented correctly.

Let x_0, x_1, x_2, ..., x_{n-1} be the vertices in a clockwise or counter-clockwise order around a n-sided polygon. Furthermore, let N_i be the normal of the edge between x_i and $x_{i\oplus 1}$.

The second step is finding the topmost vertex. In cases of ambiguity, of all topmost vertices, take the leftmost. This vertex is certain to be convex. Say this is vertex is vertex x_i.

Let U be the vector from x_i to $x_{i\oplus 1}$, and V be the vector from x_i to $x_{i\ominus 1}$. Then calculate the value of $U \bullet N_i$. If it is positive, invert N_i, otherwise do nothing. Similarly, calculate $V \bullet N_{i\ominus 1}$ and if it is positive, invert $N_{i\ominus 1}$, otherwise do nothing. N_i and $N_{i\ominus 1}$ are now correctly oriented.

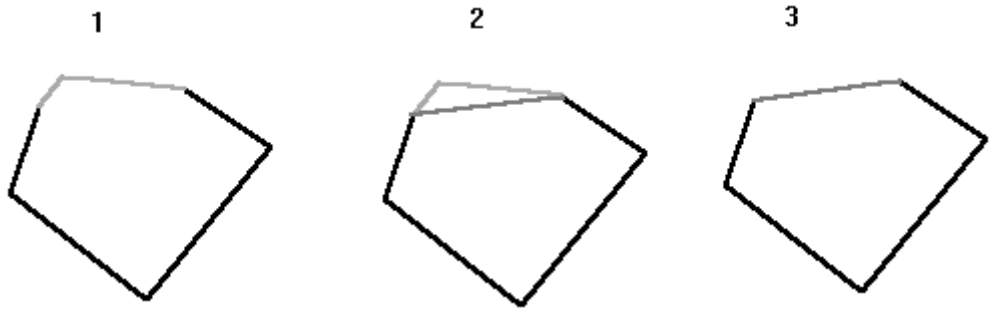
The point of that first step was to make at least one correctly oriented normal. Then, start following the edges and generate correctly oriented normals as follows.

Given a vertex x_i for which $N_{i\ominus 1}$ is known to be correctly oriented, N_i can be computed as follows. Let U be the vector from x_i to $x_{i\oplus 1}$, and V be the vector from x_i to $x_{i\ominus 1}$. Calculate $N_{i\ominus 1} \bullet (U+V)$ and $N_i \bullet (U+V)$. If the results are of the same sign do nothing. If they are of different signs, invert N_i. N_i is now correctly oriented.

Triangulating a polygon

Let us first cover the convex scenario. We will be using the same notation as in the previous section.

Take any triplet of vertices $x_{i\ominus 1}$, x_i, $x_{i\oplus 1}$. These three vertices form the first triangle. Then, remove vertex x_i from the list, and the polygon has now one less vertex. Repeat until the polygon is a triangle, at which point you are finished.



One step of the algorithm is shown above.

The concave scenario is a bit more complicated. What we will do is split the concave polygon into smaller polygons, eventually resulting in either triangles or convex polygons that can be triangulated as above.

Find a vertex that is concave. Let U be the vector from $x_{i\ominus 1}$ to x_i. Then, vertex x_i is concave if and only if $U \bullet N_i$ is more than zero. Loop through the vertices until you find such a vertex. If you do not find one, then the polygon is convex and triangulate it as above.

From that vertex, find a second vertex x_j for which the line segment from x_i to x_j does not intersect any other edge. Then, insert that new edge, making two polygons, one that has the vertices x_i, $x_{i\oplus 1}$, $x_{i\oplus 2}$, ..., x_j, and one that has vertices x_j, $x_{j\oplus 1}$, $x_{j\oplus 2}$, ..., x_i. Re-apply the algorithm on these two smaller polygons.

It can be demonstrated that using the above algorithm on a n sided polygon will generate exactly n-2 triangles.

Computing a plane normal from vertices

It can be shown that the (P,Q,R) components of the normal vectors are proportional to the signed area of the projection of the polygon on the yz, xz and xy plane respectively.

The signed area of a polygon in (u,v) coordinates can be shown to be:

$$A(u,v)=1/2\times\sum_{0\le i<n}(v_i+v_{i\oplus 1})\times(u_{i\oplus 1}-u_i)$$

where (u_i, v_i) are the coordinates of vertex x_i in 2d.

Since we're not really interested in the signed area, but some constant time the signed area, the 1/2 can be safely ignored without loss of precision.

Given a polygon in 3d, one can compute the above with:

P=A(y,z) Q=A(z,x) R=A(x,y)

Or, if you want, P is the area as calculated using only the y and z components of the points in 3d, Q is the area as calculated using the z and x components of points in 3d, and R is the area as calculated using the x and y components of points in 3d.

Once this value of (P,Q,R) is known, the result should be normalized, and then correct orientation should be checked as described hereafter.

It should be noted that the A(u,v) equation simplifies to

$A(u,v)=1/2\times(u0-u1)(v0-v2)-(v0-v1)(u0-u3)$

in the case of a triangle. Again, the 1/2 constant can be ignored.

Generating correctly oriented normals for polyhedra

In some cases, normal orientation is implicit in the object description we have. For instance, some modelers output all vertices in a counterclockwise manner when seen from above. If this is the case, then all that is needed is that the normal be computed in a specific way, without changing the ordering of the vertices. Then the normals will be correctly oriented.

If this is not the case, we need some form of algorithm to ensure proper normal orientation.

For this task, we need to have computed the normals to the edges to for all polygons making up the polyhedron, each in their respective plane of course. The edges normals in the polygons planes can be localized in space for the polyhedron, we are going to use this. Note that each edge is connected to two polygons, thus has two normals, one per polygon.

Find the vertex with the smallest x coordinate. In case of ambiguity, resolve with the smallest y coordinate. In case of ambiguity, resolve with the smallest z value. This vertex is known to be convex. Take all edges connected to that vertex, and find the vector U that is the sum of all edge normals (two per edge). Then, for each face touching the point, calculate $A\bullet U$, where A is the face normal. If the result is negative, invert A, otherwise leave it as it is. All such faces now have correctly oriented normal.

From this point, traverse all faces, propagating correctly oriented normals as follows. Let us assume we have two faces F1 and F2, and that F1's normal is correctly oriented. Let A1 and A2 denote F1 and F2's normals respectively. Pick an edge shared by F1 and F2, and compute U, the sum of the two edge normals. Then evaluate A1•U and A2•U. If they are of different signs, invert A2, otherwise leave it that way. A2 is now correctly oriented.

A special note, if the dot products are very close to zero, the face should be initialized with the same normal, and marked as ambiguous. Later, if you can find another face to help you better determine the orientation of that face, use that normal instead. At any rate, ambiguous faces should be avoided when propagating normal orientation.

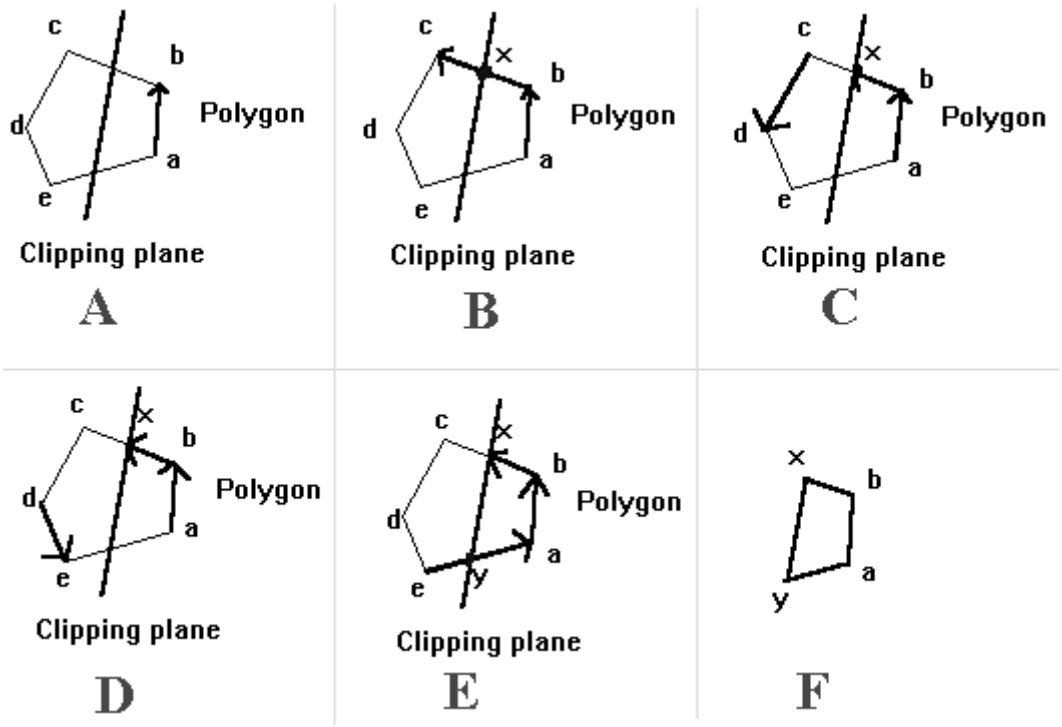
One very good way of propagating the normals is to start with one of the initial faces for which we generated the normal, and then do a depth first search through connected faces. The depth first search is elementary and will not be discussed here because it is not absolutely necessary, though it will tend to minimize time spend computing normals orientation.

Polygon clipping against a line or plane

This problem often occurs in computer graphics, and is often needed real time. Fortunately, convenient solutions exist that work well.

The simplest solution is with convex polygons. In this case, one should note that there are only 2 intersections of the clipping line or plane with the edges of the polygon. When we face a concave case, there is an even number of intersections with the edges, but some ordering should be done for them, or degenerate edges might result.

The method for clipping convex polygons is illustrated below.



Note that if one wants to keep both pieces of the clipped polygon, this algorithm can be trivially extended.

A more formal way of describing this algorithm is as follows.

```
Let v1,v2,v3, ..., vn be the list of verticies, listed in a
clockwise or counter-clockwise fashion.

Then P1 will be the first piece of polygon, and P2 will be the
second piece.

For i iterating from 1 to n do
      If the edge from vi to vi⊕1 intersects the clipping line
            break loop
      End if
End for

For j iterating from i to n do
      If the edge from vj to vj⊕1 intersects the clipping line
            break loop
      End if
End for

Let x be the intersection point of edge vi-vi⊕1 with the clipping
line.
Let y be the intersection point of edge vj-vj⊕1 with the clipping
line.

P1 is (in clockwise or counterclockwise)
      v1,v2, ..., vi, x, y, vj+1, vj+1, ..., vn
P2 is x,vi+1,vi+2, ..., vj,y
```

When doing this to a concave polygon, the algorithm is slightly more complex. Find all intersection points of edges with clipping line, and sort them according to some arbitrary axis (try to use one for which the points coordinates vary a bit). Name these sorted points p1, p2, ..., pn. Then, insert the new edges p1-p2, p3-p4, ..., pn-1 - pn. Then separate the two polygons and you are done.

Glossary

Convex: term used to describe polytopes, such as polygons and polyhedra. It means that the inside angle is always less than or equal to 180°. A triangle is always convex. A square or a rectangle is convex, but other quadrilaterals may not be convex. The term convex is sometimes used for a vertex or edge to say that the inside angle at the vertex or edge is less than or equal to 180°. An equivalent definition of convexity is, given a polytope, the intersection of the polytope and any line is always 0 or 2 points except in degenerate cases.

Concave: any polytope that is not convex is concave.

Edge: a line segment between 2 vertices. An edge typically delimitates a polygon. A square has 4 edges, a cube has 12.

Face: a polygon that delimitates a polyhedron. A face is always planar. A cube has 6 faces. Also, since polygons in 3d have two sides, these are sometimes referred to as faces.

Matrix: a 2 dimensional array of real numbers. Can also be thought as an array of vectors, or a vector of vectors.

Polygon: a flat, 2d polytope delimited by straight edges and vertices. Examples include triangles, squares, decagons. We normally prefer all vertices to be distinct, and that edges do not cross. We don't like it either when the polygon is disconnected (e.g. has several, distinct parts that are not connected).

Polyhedron: a 3d polytope delimited by planar faces, linear edges and vertices. Examples include cubes, tetrahedra, icosahedra. As with the polygon, we prefer vertices to be distinct, edges not to cross, faces not to intersect, and the polyhedron to be made of one piece as opposed to several disconnected pieces.

Polynomial: a mathematical entity that can be reduced to the form $a_0+a_1x+a_2x^2+a_3x^3+...+a_nx^n$ for some n, a_i being a real number and x_i a real variable. Example polynomials include: 3, 2+x, x^5+2x+3, x(x+2)(x-3). Examples of things that are not polynomials: x(x+1)/(x+2), $x^2+x+\sin x$.

Polytope: an object in n dimensions defined by linear constraints. Examples in 2d and 3d are polygons and polyhedra, respectively. Polytopes are made of a single piece. That is, from any point in a polytope, there is a path (which might be twisted) that can get you to any other point in the polytope without exiting the polytope.

Taylor series: A power series that approximates a function.

Vector: strictly speaking, a n-uplet, such as (3,2,5,1). It can be viewed as an arrow in any number of dimensions, from one point p1 to a point p2. The vector from (x,y,z) to (a,b,c) is (a-x,b-y,c-z).

Vertex: a point, usually called this way when it is the endpoint of at least one edge. A square has 4 vertices, a cube has 8.