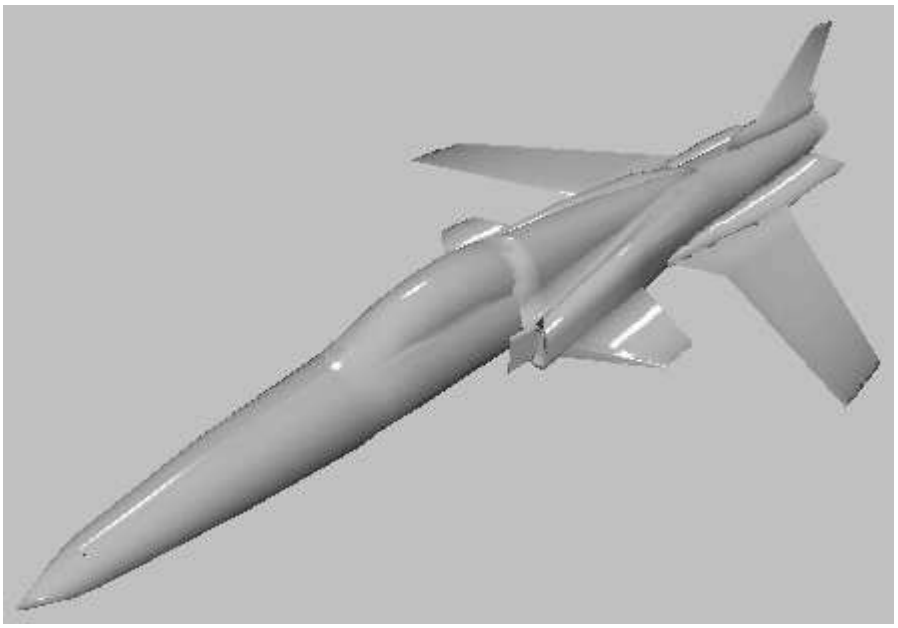
Zed3D
A compact reference for 3d computer graphics programming

by Sébastien Loisel

© 1994, 1995, 1996

Zed3D - A compact reference for 3d computer graphics programming



Copyright 1994, 1995, 1996 by Sébastien Loisel All Rights Reserved, version 0.95β

Zed3D is a document about computer graphics, more particularly real-time 3d graphics. This document should be viewed as a practical reference for a first and perhaps second course in computer graphics.

The original Zed3D document grew out of my work notes. As a matter of fact, the original Zed3D, up to version 0.61beta, was my work notes. As such, it was messy, incomplete and often incorrect. I have attempted to correct this a bit now. I still consider these my work notes, but I have added more formal introductory material which was 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 theoretical 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@cs.mcgill.ca

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

Sébastien Loisel
3436 Aylmer Street, apartment 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 work something 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 theoretical, 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, polygon triangulation and quaternions to represent orientations, which are especially useful in keyframe animations.

There is also a short glossary and even shorter bibliography. [1] is a highly recommended reading to anyone intending to do serious computer graphics. There is a lot of overlap between Zed3D and [1], though [1] doubtlessly contains a great deal more information than this text. However, Zed3D does cover a rare few topics which are more or less well covered in [1] (example: quaternions).

Of course, a lot of topics remain to be covered, such as real-time collision detection, 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 programming 2

Contact information 3
Registration 3
Overview 4

Table of Contents 6

Vector mathematics 10

Introduction 10
On notation 10
Vector operations 11
Exercises 12
Answers 13
Alcoholism and dependance 13
Exercises 14
Answers 15
On a plane (and of motion sickness) 15
Exercises 16
Answers 17
Orthonormalizing a basis 17

Matrix mathematics 19

Introduction 19
Matrix operations 19
Exercise 21
Answer 22
Matrix representation & linear transformations 22

Affine transforms 25

Introduction 25
Affine transformations 25
Exercise 26
Affine transform combination and inversion 27
Exercise 27
Answer 28

Applications of linear transformations 30

Introduction 30
World space, eye space, object space, outer space 30
Transformations in the hierarchy (or the French revolution) 31
Some pathological matrices 31

Perspective 34

Introduction 34
A simple perspectively incorrect projection 35
The perspective transformation 35
Theorems 38
Other applications 39
Constant Z 41
Texture mapping equations revisited 41
Bla bla 43
Reality strikes 44

Interpolations and approximations 46

Introduction 46
Forward differencing 46
Approximation function 48

Polynomial Splines 51

Introduction 51
Basic splines 51
Parametrized splines 54
Uniform splines 54
Examples 57
Frequently used uniform cubic splines 60
Hermite splines 60
Bézier splines 61
Convex hull 62

Bernstein polynomials 63
Uniform nonrational B-spline 64
Catmull-Rom splines: a non-uniform type of spline 65

Rendering 66

Introduction 66
The point 66
Lines 69
Polygon drawing 70

Visible surface determination 73

Introduction 73
Back-face culling 74
Sorting 75
Painter's algorithm and depth sorting 77
Z-Buffering 79
Binary Space Partitioning 80

Lighting models 84

Introduction 84
Lighting models 84
Smooth shading 86
Texture mapping & variants on the same theme 89

Computer graphics related problems 91

Introduction 91
Generating edge normals 91
Triangulating a polygon 92
Computing a plane normal from vertices 93
Generating correctly oriented normals for polyhedra 94
Polygon clipping against a line or plane 95

Quaternions 97

Introduction 97
Preliminaries 98
Conversion between quaternions and matrices 99
Orientation interpolation 99

Antialiasing.....101
Introduction..... 101
Filtering..... 101
Pixel accuracy..... 102
Sub-pixel accuracy..... 104
Time antialiasing, a.k.a. motion blur..... 105
Mipmapping..... 106
Uniform Mipmapping..... 107
Nonuniform Mipmapping..... 108
Summed area tables..... 109
Bi-linear interpolation..... 110
Tri-linear interpolation..... 111

Glossary.....112

Bibliography.....115

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 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 its 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 for any 3d graphics engine to be useful.

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

On notation

I will frequently use the sigma symbol for sums, for example, something like this:

$\sum_{0 \le i \le n} a_i$

which stands for

$a_0+a_1+a_2+a_3+...+a_n$.

More generally, the notation

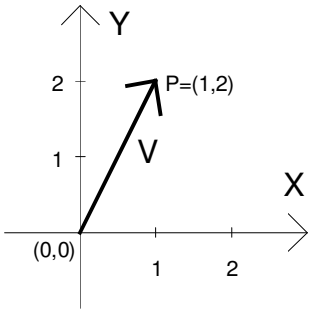
$\sum_{i \in I} a_i$

stands for "sum of all a_i for all i in I". This notation will not be used frequently in this work. Notation from more advanced math might be used especially in proofs.

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.

A vector should be thought of as an oriented line segment from the origin (0) to a point P in space. Let's take a 2d example (this is also valid for 3d or higher dimension). The vector V=(1,2) can be represented as an oriented segment from (0,0) to P=(1,2), as can be seen below. A vector should always be pictured as an arrow from 0 to the point P. Using this model, we can think of a vector from P1=(a,b) to P2=(c,d) as the vector from (P1-P1) to (P2-P1), or from (0,0) to (c-a,d-b). This illustrates a very important point. The vector from P1 to P2 is the exact same vector as the vector from 0 to P2-P1. Two vectors that differ only by a translation are considered equivalent.



Vector addition is defined as follows. 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 defined 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 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. Note that the 1d case corresponds to normal multiplication of real numbers in a certain way.

Vectors have a **length**, defined as follow. The length (or **module**, or **norm**) of vector U is written **|U|** and has the value of $\mathbf{(U \bullet U)^{1/2}}$. If the length of a vector is one, the vector is said to be of **unit length**, or a **unit or normal vector**. Multiplying a vector V by the scalar 1/|V| is called **normalizing** a vector, because it has the effect of making V a unit vector. In the 1d case, length simplifies to **absolute value**, thus the notation |U|.

Dot product is also used to define 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 **(u2v3-u3v2, u3v1-u1v3, u1v2-u2v1)**.

It is demonstrable 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 which 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) (-3,6,-3)

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). All the vectors of the said plane can be expressed as p(1,-1,0)+r(0,-1,1) [for example] for some real numbers p and r. This last notation is also known as a local coordinates system for the u1+u2+u3=0 plane.

Alcoholism and dependance

Given a set of vector U0, U1, U2, ... , Un, these vectors are said to be **linearly independent** 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 at least one solution for which (a0, a1, a2, ...,an) is **not** zero, then there exists an infinity of them, and the vector are said to be **linearly dependant**.

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

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

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

The term **orthogonal** is very frequently used to describe **perpendicular** vectors. If a basis is made of orthogonal unit vectors (unit vectors are vectors of norm 1), the base is said to be **orthonormal**. Orthonormal basis are the most useful kind in typical 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 simplest. 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 traditionally 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.

Note: normally, to specify which space a vector is expressed in, we should write the space in subscript. Example: as in the preceding paragraph, (a,b,c) is written either $\mathbf{(a,b,c)_{pqr}}$ or $\mathbf{(a+c,a+b,b+c)_{ijk}}$ depending on whether we want it in pqr or ijk space. This notation will help avoid many mistakes.

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 $\mathbf{(a,b,c)_{pqr}}$ expressed in pqr space (a×p+b×q+c×r) can be written as $\mathbf{(a,b,c)_{pqr}\bullet(p,q,r)_{ijk}}$.

Exercises

Q1 - Are the vectors (1,2,0), (4,2,4) and (-7,-4,-8) linearly independent?

Q2 - Say vector $U=(1,3,2)_{pqr}$ is expressed in pqr space, where pqr expressed in ijk space is $(1,2,0)_{ijk}$, $(2,0,2)_{ijk}$, $(0,-1,-1)_{ijk}$. 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 - $(7,0,4)_{ijk}$

A3 - (1/3, 1/3, -1/3) - 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 independent vectors to form a basis. Call them U and V. Then, if you take $a\times U+b\times V$ 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 $\mathbf{a\times U+b\times V+W}$ 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 $\mathbf{(a,b)\bullet(U,V)+W}$. 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 $A\bullet X=0$, 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 literature 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 $A\bullet 0=0$ 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 follows. (This uses an intuitive form of affine transformations, described 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 signed distance equation simplifies to A•P-D.

It is easy 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.

Answers

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.

Orthonormalizing a basis

Sometimes we might have a basis B which is meant to be orthonormal, but due to accumulation in roundoff error in the computer, the vectors are slightly off the unit length and not quite perpendicular. Then it is useful to have a way of finding an orthonormal basis O from our basis B while making sure that O and B are "very similar" in a certain sense.

The meaning of "very similar" can be made explicit easily. Let B be the basis (b_1, b_2, ..., b_n) for a n-dimensional space (b_i's are vectors). Let O be the basis (o_1, o_2, ..., o_n). Then, we measure the "similitude" of O and B by taking $\text{Max}(|o_i-b_i|)$, that is, the greatest difference between the same vector in O and B. The closer this number is to 0, the more similar O and B are. The method given below will generate O from B such that the similitude is small enough (note that it will not necessarily be the smallest possible, it will simply be small enough).

The process in n-dimensional space is as follows. Let

$v_1=b_1$

$v_n=b_n-\sum_{1\leq i<n}(b_n \bullet o_i)o_i$

$o_n=v_n/|v_n|$

Then, the basis O is orthonormal and has good similitude with the basis B. (Proof is left as an exercise. Hint: find an upper bound on the similitude as a function of the maximum of the dot product between two vectors of the basis B and as a function of the length of the vectors in the basis B. Proof of orthogonality comes from examining v_n closely. Unit norm of the vectors of the basis is obvious.)

Explicitly for the 3d case, this simplifies to:

$o_1=b_1/|b_1|$

$v_2=b_2-(b_2 \bullet o_1)o_1$

$o_2=v_2/|v_2|$

$v_3=b_3-(b_3 \bullet o_1)o_1-(b_3 \bullet o_2)o_2$

$o_3=v_3/|v_3|$

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:

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

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 $\mathbf{M=(m_{ij})}$ with i and j varying from 1 to 3. The **first** index is the row number, the **second** index is the column number, as in the example above.

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

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

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:

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

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

$$c_{11}=a_{11}\times b_{11}+a_{12}\times b_{21}$$

$$c_{12}=a_{11}\times b_{12}+a_{12}\times b_{22}$$

$$c_{21}=a_{21}\times b_{11}+a_{22}\times b_{21}$$

$$c_{22}=a_{21}\times b_{12}+a_{22}\times b_{22}$$

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

It is important to notice that matrix multiplication is **not commutative** in the general case. For example, it is **not true** that $A\times B=B\times A$ with A and B matrices in the general case, even if A and B are square matrices. Matrix multiplication is, however, **associative** (ie, $A\times(B\times C)=(A\times B)\times C$) and **distributive** (ie, $A(B+C)=AB+AC$).

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

Matrix **transposition** of matrix A, noted $\mathbf{A^T}$, 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 usually 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** (sometimes written **detM**) 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.

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

The C_{ij} are the cofactors of M - they 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:

$$C_{ij}=(-1)^{i+j} \times \det M_{ij}$$

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

$$\begin{aligned} D_{3x3}= & \ m_{11} \times (m_{22} \times m_{33}-m_{23} \times m_{32}) \\ & - m_{12} \times (m_{21} \times m_{33}-m_{23} \times m_{31}) \\ & + m_{13} \times (m_{21} \times m_{32}-m_{22} \times m_{31}) \end{aligned}$$

Given a matrix A, the **inverse** of the matrix, noted $\mathbf{A^{-1}}$ (if it exists), is such that $A \times A^{-1}=A^{-1} \times A=I$. 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:

$$A^{-1}=(1/\det A) \times B^{T}$$

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) literally 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:

$M \times X = D$

This is also called the **linear transformation** of X by M. In this case, if the matrix M is invertible, then we can premultiply both sides of the equality by M^{-1}, as follows:

$M^{-1} \times M \times X = M^{-1} \times D$

And, knowing that $M^{-1} \times M = I$ (and that matrix multiplication is associative as we saw before), we substitute into the above:

$I \times X = M^{-1} \times D$

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

$X = M^{-1} \times D$

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:

$P' = M \times P$

$P'' = N \times P'$

By combining these two equations, we get

$P'' = N \times (M \times P)$

However, by associativity of matrix multiplication, we have:

$P'' = (N \times M) \times P$

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.

A very special note. Sometimes it becomes useful to distinguish vectors from points in space. A vector is **not** affected by a translation, while a point is. To illustrate our example, think of a plane and a plane's normal. Let's say we take three points in the plane, rotate them and translate them, we get a new plane. These points are affected by the translation and the rotation. However, the plane normal is only affected by the rotation.

When using affine transforms with the 4x4 matrix above, a vector (x,y,z) is represented by (x,y,z,0) and a point is represented by (x,y,z,1). This way, when you multiply a vector by a 4x4 matrix, the translation does not affect it (try it and you will see), while a point is affected by it.

This very important aspect gives meaning to the various operations on points and vectors. Sums and differences of vectors are still vectors. (E.g. (a,b,c,0)+(d,e,f,0)=(a+d,b+e,c+f,0), which is still a vector). Difference of two points is a **vector**. This is very important:

(a,b,c,1)-(d,e,f,1)=(a-d,b-e,c-f,0) (a vector since the last component is 0)

Sum of two points has **no meaning**. (It can be given one, but for us it has no meaning). This is illustrated this way: (a,b,c,1)+(d,e,f,1)=(a+d,b+e,c+f,2). The last component is no 0, so it's not a vector, and it's not 1 so it's not a point. (We could use homogeneous coordinates and give it a meaning, but this is totally unimportant.)

Sum of a vector and a point is a **point**. Subtracting a vector from a point yields a point, also.

Exercise

Prove that the sum of a vector and a point is a point and not a vector or undefined, and prove that the difference of a point and a vector is a point as opposed to a vector or undefined. What is the meaning if any of multiplying a point by a scalar? a vector by a scalar?

Affine transform combination and inversion

The most straightforward way to compute the affine combination or inversion is to write down the 4x4 matrices and perform the matrix operations on them. This will yield the correct results. It is also possible to proceed in a different way, as presented here.

Let an affine transform be represented by the (M,T) couple, where M is the 3x3 linear transform matrix and T is the 3d vector which is added after the M matrix is applied. Then, the affine transform U of vector V can be written as:

$MV+T=U$

If we want to find the inverse transform, we want V as a function of U. Simple matrix arithmetics tells us the following:

$MV=U-T$

$V=M^{-1}(U-T)$

$V=M^{-1}U-M^{-1}T$ (distributivity of matrix multiplication over matrix addition)

Hence, the inverse of affine transform (M,T) is $(M^{-1},-M^{-1}T)$.

Affine transform combinations can be computed in a similar way. Let's assume we want to find the affine transform U of V by (M,T), then the affine transform W of U by (N,S). This means:

$U=MV+T$ $W=NU+S$

$W=N(MV+T)+S$

$W=NMV+NT+S$

$W=(NM)V+(NT+S)$

Thus, the combination of (M,T) followed by (N,S) is simply (NM, NT+S).

Exercise

Q1- Assume we have three points P={P1, P2, P3} in 3d and the three points Q={Q1, Q2, Q3} also in 3d. These points are read from a special device and their real location in 3d space is known with very good precision (note: this means there is no perspective distortion in our data). We know that the points P and Q are the very same points, except that they're viewed from a different location. This means that the points in Q are the points in P transformed by some affine transform A. However, we do not know which points in Q correspond to which points in P. (ie, Q1 is **not** necessarily the affine transform of P1, it might be the affine transform of P2 or P3). You can assume that the points P1, P2 and P3 form a nondegenerate triangle whose sides all measure a different length.

Q2- We have roughly the same problem as in Q1, except now we have n>3 points P=(P1, P2, P3, ..., Pn) and the corresponding Q=(Q1, Q2, Q3, ..., Qn). Can you find a way to compute the affine transform while minimizing error? (Warning - this is difficult.)

Answer

A- First step is to determine which point in Q correspond to which point in P. Since they're the same points viewed from different angles, we can assume the linear transform part of the affine transform is orthogonal, therefore it preserves lengths and angles. We can use that to find which points should be associated. To this purpose, let u=P2-P1, v=P3-P1. Then, find the i,j, k such that $|u|=|Qj-Qi|$, $|v|=|Qk-Qi|$. Since we assumed the sides of the triangle have all different lengths, there is only one i,j,k which will work. We can simply try all 6 combinations until one works. Then, we know that P1 corresponds to Qi, P2 corresponds to Qj, P3 corresponds to Qj.

Now, let R1, R2, R3 be Qi, Qj, Qk respectively (this is to simplify notation a bit). We need a third vector, which we generate as follows. Let $w=u\times v$. Note that, as seen in the last section, u, v and w are vectors and therefore are **not** affected by translations. Let the affine transform A be represented by (M,T) a 3x3 matrix and a 3d vector. Let p=R2-R1, q-R3-R1 and $r=p\times q$.

Then, we have that p=Mu, q=Mv, r=Mw (prove it, especially the last one).

This can be re-written as

$M(u|v|w)=(p|q|r)$

where (u|v|w) denotes the 3x3 matrix formed by taking the vectors u, v and w and putting them in as column vectors. Then, we can compute W by calculating

$M=(p|q|r)(u|v|w)^{-1}$ (*)

Now we have computed the M matrix. We need to compute the T vector. We know that R1=MP1+T, hence T=R1-MP1 and we are done.

A2- The general outline is similar to A1, except that at step (*), instead of using the conventional matrix inversion, we need a so-called pseudoinverse matrix, denoted M^+, which is

$$M^+=(M^TM)^{-1}M^T$$

This matrix is a generalization of the conventional matrix inverse. It minimizes mean square error in overconstrained sets of equations like we have here. See [2] for more information on this topic. Note that finding which Qi correspond to which Pj is slightly more difficult, but a similar method can be used. Also note that the T vector should be computed for all points and then averaged to minimize error. Additionally, we were generating a w vector which was the cross product of u and v. Now we **might** require something analogous to generate a linearly independent component else the matrix will be degenerate and inversion will be highly error prone. This especially if the points are suspected to be coplanar.

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 $\mathbf{A_{N\leftarrow M}}$. This has the natural tendency to make us combine the affine transform from right to left instead of left to right, which is correct.

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 $C_{Camera\leftarrow World}\times A_{World\leftarrow Object}\times P_i = M_{Camera\leftarrow Object}\times P_i$.

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 $M_{Camera\leftarrow Object}=C_{Camera\leftarrow World}\times A_{World\leftarrow Object}$ and multiplies all Pi'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". Additionally, we have to edit the position of the television and change it to "The television 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:

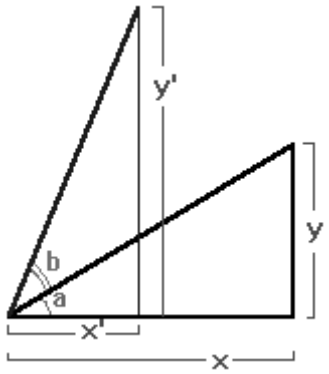
$$C_{Camera\leftarrow World}\times B_{World\leftarrow Desk}.\times A_{Desk\leftarrow Television}\times P_i.$$

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

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

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)

Rotations 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:

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

(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 constant speed, the object will definitely not rotate with constant speed. 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 and/or interpolate between any two given orientations when using **quaternions**, which is covered in a later chapter..

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 flowers 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.

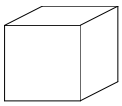
In normal, day-to-day life, when you're significantly larger than an atom but significantly smaller than a planet, light is usually pretty linear. It travels in straight rays, only bending at discrete points that are more or less easy to calculate, definitely more than the fuzzy way light bends in a 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 starships are pretty dull in appearance, with perhaps a diffuse highlight from where the light is coming from.

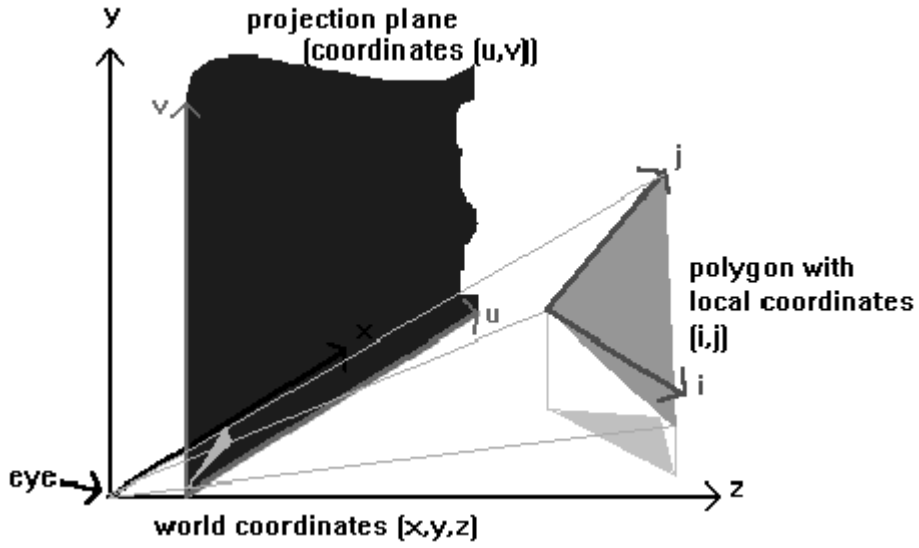
Another simplification we usually make comes from the fact that light bounces off everything and eventually starts coming from about all direction with a low intensity. This is often called the ambient light. Some further optimizations, more hacks than actual physical observations, will make you go faster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

A simple perspectively incorrect projection

The most simple **projection** is an affine transform from 3d to 2d, sometimes referred to as **parallel projection**. As an example, the transform $(x,y,z)\rightarrow(x,y)$ transforms the point (x,y,z) in 3d to the point (x,y) in 2d, is such a transform. Another simple example is the $(x,y,z)\rightarrow(x+z,y+z)$ transform. The problem with this is, no matter how far or close in z the object is, it always appears the same size on the screen. This, or a variant of this, is true for all of the parallel projections. These projections are called parallel because parallel lines in 3d remain parallel once projected in 2d. The image below is a parallel projected cube:



The perspective transformation



The **perspective transformation** (or **perspective projection**) 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 high-school.

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 referred 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 referred 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 mentioned 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).

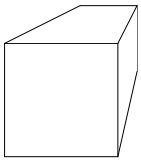
Trivially from that, we find that the point (a,b) on screen are (k×x/z,k×y/z). Thus,

(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≤0, do not project. The image below is a perspective projected cube. Compare with the parallel projected cube of the preceding section.



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 necessarily 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.

Constant Z

There is one specific case that might be especially interesting, given slow division but fast addition. The plane equation for a polygon is Ax+By+Cz=D. The projection is u=k1x/z, v=k2y/z. Then, we get x=uz/k1, y=vz/k2. By substituting this into the plane equation of the polygon, we find A(uz/k1)+B(vz/k2)+Cz=D. Then, we transform as follows:

z(A'u+B'v+C)=D (A'=A/k1, B'=B/k2)

Let us examine what happens when we look at a slice of constant z in the polygon's plane.

k(A'u+B'v+C)=D

Mu+Nv+K=0 (M=kA', N=kB', K=C-D)

This is a line equation in (u,v) space. This means that, assuming non degenerate cases, a constant z slice of the polygon's plane projects to a line in the projection plane. Furthermore, and interestingly enough, the slope of the line is *independent of z*. Therefore, for a given polygon plane, all the constant-z lines of that plane project to parallel lines on screen. However, looking back at the Ax+By+Cz=0, taking a constant z, we get a line equation of x and y, therefore, the intersection of a constant z plane with the polygon plane is also a line.

Now let's examine the projection equation. Let us assume that we wish to project everything that's on a specific constant-z line of the polygon. Then, the projection equation is simply u=Px, v=Qy, where P=k1/z, Q=k2/z, constants.

This is what it all boils down to. In any polygon, there are lines of constant z. If we want to texture map the polygon, we only need to find these lines and draw them on screen, merely scaling the texture for such lines by a constant. Since all these lines are parallel on screen, it is possible to find the slope the line on screen that will yield a constant z on the polygon's plane, and then draw to the screen using these as scanlines. One has to be careful to cover each pixel, but that is not too difficult.

As an example, a wall's constant-z lines are vertical once projected (assuming we're looking at it upright). A floor or ceiling's constant-z lines are horizontal once projected. This can be exploited to texture map floors, ceilings and wells very quickly.

Texture mapping equations revisited

We derived the texture mapping equations using the intuitive math above, and got nasty looking rational expressions with even nastier coefficients (the constants A, B, C, D, E, F, G, H and I). In practice it might be useful to try to find an efficient way of computing these constants.

There is a clever way to calculate these constants, but first we have to write down a few properties. First let us observe that our texture map is an affine mapping from our (x,y,z) 3d space to the (p,q) 2d texture map, which means that:

1- $p=P1\times x+P2\times y+P3\times z+P4$

$q=Q1\times x+Q2\times y+Q3\times z+Q4$

(for some P1, P2, P3, P4, Q1, Q2, Q3, Q4).

Second, assume that the plane equation of the polygon to be texture mapped is given by

2- $Ax+By+Cz=D$ (where (A,B,C) is the plane's normal, of course)

Third, write down the perspective projection:

3- $(u,v)=(k1x/z,k2x/z)$

From 3, get (x,y) as a function of u, v and z:

3a- $(x,y)=(uz/k1, vz/k2)$

Substitute x and y into the equation we had in 1 and 2 to get:

4- $p=P1uz/k1+P2vz/k2+P3z+P4$

$q=Q1uz/k1+P2vz/k2+P3z+P4$

5- $Auz/k1+Bvz/k2+Cz=D$

Now divide 4 across by z, get:

6- $p/z=P1/k1\times u+P2/k2\times v+P3+P4/z=R1\times u+R2\times v+P3+P4/z$

$q/z=Q1/k1\times u+Q2/k2\times v+Q3+Q4/z=S1\times u+S2\times v+Q3+Q4/z$

From 5, find 1/z, get:

7- $1/z=(A/(D\times k1))\times u+(B/(D\times k2))\times v+(C/D)=Mu+Nv+O$ (*)

Look at 7 and compute P4/z and Q4/z by multiplying across by P4 and Q4 respectively:

8- $P4/z = P4 \times Mu + P4 \times Nv + P4 \times O$

$Q4/z = Q4 \times Mu + Q4 \times Nv + Q4 \times O$

Substitute these two equations into 6 and get:

9- $p/z = R1 \times u + R2 \times v + P3 + P4 \times Mu + P4 \times Nv + P4 \times O$

$= (R1 + P4 \times M) \times u + (R2 + P4 \times N) \times V + (P3 + P4 \times O)$

$= J1 \times u + J2 \times v + J3$ (**)

(similarly)

$q/z = K1 \times u + K2 \times v + K3$ (***)

Now examine (*), (**) and (***). These are all linear expressions in (u,v). This means that:

- 1/z is linear in screen space (u,v) after the perspective transform
- p/z is also linear in screen space after perspective transform
- q/z is also linear after perspective transform

Which leads us to the following conclusions: we can interpolate linearly 1/z across the screen for a polygon, and that will be perspective correct. We can linearly interpolate p/z across the screen for a polygon, and that will also be perspective correct. We can interpolate linearly q/z across the screen for a polygon and that will also be correct. Then, we can find the (p,q) texture coordinate of any texel as follows:

$p = (p/z) / (1/z)$

$q = (q/z) / (1/z)$

A simple quotient of our linearly interpolated values.

This simply allows us to use maybe already existing linear interpolation routines to figure out the perspective correct texture mapping, with only a simple tweak added.

Bla bla

Other applications can also be found to the theory of the perspective projection. 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 attribute 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.

Note that the operations X=A/C and Y=B/C can be replaced by the operations T=1/C, X=T×A, Y=T×B. This essentially replaces two division by one division and two multiplications, which can sometimes be actually faster. This exploits the fact that the denominators are the same, just as in texture mapping. Additionally, the T=1/C computation can be implemented using a lookup table. Or logarithm tables can be used, by noting that a×b=exp(log(a)+log(b)) and a/b=exp(log(a)-log(b)), replacing a multiplication or division by three lookups and an add or subtract. All these tricks have been used at some time or other. They all have the disadvantage of being less precise and taking up memory. Moreover, as CPUs become faster at math, these method are actually slower than a normal division operation (example, PowerPC). As such, these methods are quickly becoming obsolete, except on legacy hardware such as all PC's which use Intel CPUs.

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 tremendously 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:

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

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

$=2kAn+Ak^2+kB$

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+k]-dy[n], which is simply $2k^2A$.

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=32. 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), 32 to dy(x) and 4 to x)

7 66 80

(once more, add dy(x) to y(x) and 32 to dy(x) and 4 to x)

11 146 112

(etc...)

15 258 144

19 402 176

These can be extended to certain multi-variable polynomials also. Typically, however, the simple fact that the polynomial can be incrementally evaluated across a scanline is sufficient. Bilinear interpolations (r=Ap+Bq+C) are a special case of multi-variable polynomials which can be evaluated especially well in an incremental fashion. These occur naturally when interpolating texture coordinates linearly or **Gouraud shading** across a triangle.

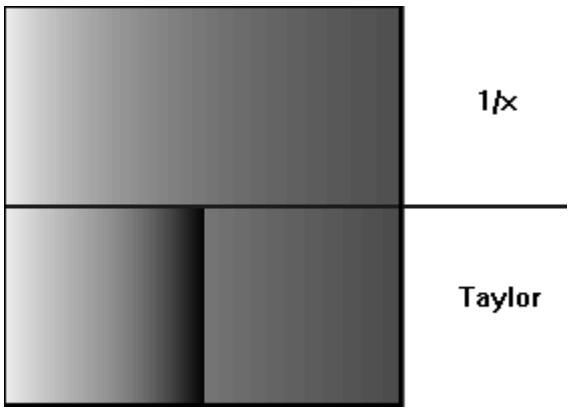
Approximation function

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 additional constraints 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 and only use the first few terms (Taylor series is taught in Calculus and is beyond the scope of this text). This, however, does nothing about the edge constraint we just mentioned. 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 $1/N \times (a,b,c)$. 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:

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

A theorem of analysis that interests us is as follows. Given n points in the plane (assuming none have the same x coordinate), there is a unique polynomial that passes through all these points. This polynomial can be found using the linear mathematics we were using previously. Here follows an example with a second degree polynomial.

Let's say we want to find the quadratic polynomial that goes through the points (x0,y0), (x1,y1) and (x2,y2). We know the polynomial is of the form $Ax^2+Bx+C=y$. We rewrite all these constraints as the following equations:

$Ax0^2+Bx0+C=y0$

$Ax1^2+Bx1+C=y1$

$Ax2^2+Bx2+C=y2$

This can be re-written as the following matrix equation:

$$\begin{bmatrix} x0^2 & x0 & 1 \\ x1^2 & x1 & 1 \\ x2^2 & x2 & 1 \end{bmatrix} \cdot \begin{pmatrix} A \\ B \\ C \end{pmatrix} = \begin{pmatrix} y0 \\ y1 \\ y2 \end{pmatrix}$$

Which can in turn be re-written as $X \times A=Y$. Notice that X and Y are constant. Then we solve for A, writing

$A=X^{-1} \times Y$

Different types of constraints can be put on the polynomials or its derivative(s), yielding different types of polynomials. The subject of interpolation is quite extensive and will not be fully discussed here.

Polynomial Splines

Introduction

In this section, I will have to assume a basic knowledge of calculus. Note that the topic of spline is rather broad, hence only the basics will be covered here. For a more detailed discussion, one can see [5].

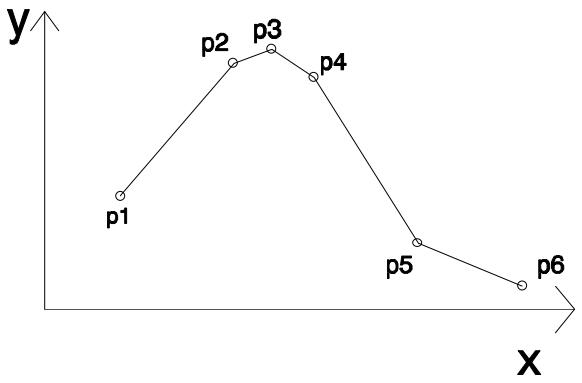
Sometimes we have many control points (10, for example) that we want to use to generate an interpolating polynomial. However, we might not want to use a 10th degree polynomial for several reasons. They're hard to evaluate. They're numerically unstable. They tend to oscillate wildly between control points.

To resolve this, we make lower degree interpolating polynomials for each section of the curve. Typically, polynomials of degree 1 (lines), 2 (quadratics) or 3 (cubics) are used. Polynomials of degree higher than 5 are unwieldy and also sometimes exhibit undesirable behavior.

Basic splines

A spline will be defined by its type and a list of control points of the form {p1, p2, p3, ..., pn} where pi=(xi,yi) some point in 2d space. The type can be a simple line segment joining each control points, or something more complex like a Catmull-Rom cubic spline. Note that a spline does **not** necessarily pass through all interpolation points. It is even possible that a spline does not pass through **any** of the control points. We will examine such cases later.

We will start by an example spline of degree 1 that interpolates through all control points. An example picture is shown below:



In the diagram, the points pi are ordered from left to right, and this is what will happen most of the time, though it is not necessary for now.

The spline is made up of 5 spline segments, which are line segments in this particular case. Let's look at the first segment that goes from p1 to p2. We can easily find the equation of that line using basic algebra. Remember that p1 is the point (x1,y1) and p2 is the point (x2,y2). The spline segment, since it is a line, is of the form y=mx+b. This line segment goes through p1 and p2, hence these two equations have to verify:

$y1=mx1+b$

$y2=mx2+b$

We have two equations and two unknowns (m and b), so we can solve for m and b. Note that this equation can be represented in matrix form:

$$\begin{pmatrix} y1 \\ y2 \end{pmatrix} = \begin{pmatrix} x1 & 1 \\ x2 & 1 \end{pmatrix} \cdot \begin{pmatrix} m \\ b \end{pmatrix}$$

This form will be more interesting for higher degree splines, so we will use this notation from now on. Using linear algebra, we can solve to the (m,b) column vector above and we then know the spline segment from p1 to p2. One of the many problems with this spline is that it's not very smooth. How do we express smoothness? We use the principle of continuity.

A curve is said to be in C1 continuous if its first derivative is continuous. A curve is in C2 if its second derivative is continuous. Generally speaking, a curve in Cj's jth derivative is continuous. For completeness, we let C0 be the family of curves which are continuous.

Smoothness can be expressed as continuity. The spline we made above is in C0, but not in C1. (Indeed, at control points, the slope changes abruptly, hence the first derivative is not continuous).

A very important note: all polynomials are in C∞, that is, they can be differentiated infinitely many times. Therefore, if our spline is made of one polynomial, it is inherently very smooth. The problem is, splines aren't *exactly* polynomials, they're polynomial segments glued together. However, if you look only at one segment of the curve, excluding its endpoints, then it's in C∞. Therefore, the only thing that might make it less than C∞ is what happens at the endpoints of curve segments.

To illustrate this, we will now create a quadratic spline which is in C1. Since we will be using this spline repeatedly in our examples, we will name it the Zed splineThis is how we define the curve segment from p[i] to p[i+1]:

1- the quadratic goes through p[i]

2- the quadratic goes through p[i+1]

3- at p[i], the slope is the same as whatever the previous spline segment has at that point.

Assume the previous curve segment was $y=A[i]x^2+B[i]x+C[i]$. Then, the derivative of that curve segment is:

$y'=2A[i]x+B[i]$

And at p[i], the derivative is:

$y'[i]=2A[i]x[i]+B[i]=K$ (*)

The new curve segment is $y=A[i+1]x^2+B[i+1]x+C[i]$. It goes through p[i] and p[i+1] hence:

$y[i]=A[i+1]x[i]^2+B[i+1]x[i]+C[i+1]$

$y[i+1]=A[i+1]x[i+1]^2+B[i+1]x[i+1]+C[i+1]$

Its derivative at y[i] is

$y'[i]=2A[i+1]x[i]+B[i+1]=K$ (K comes from (*))

We can re-write these three equations as:

$$\begin{bmatrix} y_i \\ y_{i+1} \\ K \end{bmatrix} = \begin{bmatrix} (x_i)^2 & x_i & 1 \\ (x_{i+1})^2 & x_{i+1} & 1 \\ 2 \cdot x_i & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} A_{i+1} \\ B_{i+1} \\ C_{i+1} \end{bmatrix}$$ ♦

Then we can solve for the (A,B,C) vector.

There is still the question of generating the very first spline segment, since there is no such thing as "previous spline segment's slope at p[1]" (we'll have a hard time computing K). One solution is to let the first spline segment be a quadratic that interpolates through p[1], p[2] and p[3] exactly. Then the second curve segment will maintain the same slope as the first curve segment at p[2], and interpolate through p[2] and p[3].

Parametrized splines

As of now, our splines are functions, that is, they cannot curl backwards very easily. Infinite slopes are impossible. That and other things lead us to parametric splines.

Right now, we have y as a function of x. We will now replace this with y a function of t, and x a function of t. Then we plot all points (x,y) for some values of t and we get the desired spline. We use t as a variable name because it can sometimes be useful to think of the spline as an (x,y) point moving in time. For example, a spaceship's movement could possibly be described as (x,y,z)=(f(t),g(t),h(t)), a function of time for each coordinate. The key point here is that this allows us to extend splines to any number of dimensions elegantly.

The control points are now of the form (x,y,t) or (x,y,z,t), depending on the number of dimensions we want. A parametric quadratic spline segment in 2d, for instance, would be something of the form:

$x=At^2+Bt+C$

$y=At^2+Bt+C$

We just take each component individually and make it a spline as a function of time. For example, if we have the control points (x_0,y_0,z_0,t_0) (x_1,y_1,z_1,t_1), ..., (x_n,y_n,z_n,t_n). Then, we look at x as a function of t, and make it a spline with the control points (x_0,t_0), (x_1,t_1), (x_2,t_2), ..., (x_n,t_n). We do the same for y as a function of t, with the control points (y_0,t_0), (y_1,t_1), ..., (y_n,t_n), and similarly for z as a function of t.

Uniform splines

Uniform splines are a special breed of splines which the control points are regularly spaced in a special way. That is, a spline of the form (x,f(x)) where the control points are (0,y0), (1/(n-1),y1), (2/(n-1),y2), ..., (1,yn) are called uniform. Notice that the control points x components are uniformly distributed between 0 and 1.

Uniform splines have special uses. When we want to specify an object's position at an instant with a parametric spline, we want to be able to specify the t component exactly. However, when we're more interested in the shape of the spline, the t component matters less and we use uniform splines.

Now look back at the equation marked ♦ for the Zed spline a few pages back. In the case of a uniform Zed spline, we can substitute the values x0=0 and x1=1, since there are but two control points. Then we get:

$$\begin{pmatrix} A \\ B \\ C \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix}^{-1} \cdot \begin{bmatrix} y_i \\ y_{i+1} \\ K \end{bmatrix}$$

$$\begin{pmatrix} 0 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix}^{-1} = \begin{pmatrix} -1 & 1 & -1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} = M \qquad G = \begin{bmatrix} y_i \\ y_{i+1} \\ K \end{bmatrix}$$

$$\begin{pmatrix} A \\ B \\ C \end{pmatrix} = M \cdot G$$

The column vector G is called the *geometry vector*. The product of M and G can be viewed as a linear transformation of the vector G, thus the matrix M is called the *basis matrix* for the Zed spline. A basis matrix completely defines a uniform spline type, and along with a geometry vector, it defines completely a specific spline.

To illustrate a few additionnal properties, we need a second type of quadratic spline. We will call it the Baker spline, and it is defined by two control points p0 and p1, and a constant J as follows:

1- The spline interpolates through p0

2- The spline has a slope of J at p0

3- The spline has, at x1, a slope of whatever slope the vector p1-p0 has

Now, let us see what these three constraint imply. First, let us notice that since it is a quadratic spline, it is of the form $y=Ax^2+Bx+C$. Hence, $y'=2Ax+B$. Then:

1- means that $y0=Ax0^2+Bx0+C$. Since the spline is uniform, x0=0 and y0=C.

2- means that J=y'(x0) or J=2Ax0+B or J=B

3- The slope of p1-p0 is m=(y1-y0)/(x1-x0)=(y1-y0)/1. We want m=y'(x1)=y'(1)=2A+B. Hence, y1-y0=2A+B

Combining these, we find that

2A+B=y1-y0

2A=y1-y0-J

A=y1/2 - y0/2 - J/2

We can write this in matrix form as:

$$\begin{pmatrix} A \\ B \\ C \end{pmatrix} = \begin{bmatrix} -\frac{1}{2} & \frac{1}{2} & -\frac{1}{2} \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} y_0 \\ y_1 \\ J \end{bmatrix}$$

$$N = \begin{bmatrix} -\frac{1}{2} & \frac{1}{2} & -\frac{1}{2} \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

N is the basis matrix for the uniform Baker spline. Now, given a specific geometry vector for a Baker spline, maybe we want geometry vector for a Zed spline which will generate the exact same spline. This is computed using a change of basis. Let M be the Zed spline basis matrix, N is the Baker spline basis matrix, V is the geometry vector for the Baker spline and G is the geometry vector for the Zed spline. K is the (A,B,C) vector of the coefficients of the quadratics. Then we have these equations:

$K=MG$ or $G=M^{-1}K$

K=NV

Therefore,

$G=M^{-1}(NV)$ or $G=(M^{-1}N)V$

Let $L=M^{-1}N$ then

G=LV

L is called the transition matrix from Baker spline to Zed spline. This is all nothing but linear algebra. L is a transform from one space to another space, there is nothing more to it.

Examples

Here follows an example calculation for one segment of a parametric nonuniform quadratic spline. Note that this spline is not of the Zed type. This is merely a parametric quadratic spline which interpolates through all 3 control points.

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} (t_1)^2 & t_1 & 1 \\ (t_2)^2 & t_2 & 1 \\ (t_3)^2 & t_3 & 1 \end{bmatrix} \cdot \begin{bmatrix} J_1 \\ J_2 \\ J_3 \end{bmatrix} \qquad \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} (t_1)^2 & t_1 & 1 \\ (t_2)^2 & t_2 & 1 \\ (t_3)^2 & t_3 & 1 \end{bmatrix} \cdot \begin{bmatrix} K_1 \\ K_2 \\ K_3 \end{bmatrix}$$

$$x = J_1 \cdot t^2 + J_2 \cdot t + J_3 \qquad y = K_1 \cdot t^2 + K_2 \cdot t + K_3$$

These are all the equations we need. Next, given the control points x1, x2 and x3, y1, y2 and y3 at times t1, t2 and t3, we can solve for J and K.

$$\begin{bmatrix} J_1 \\ J_2 \\ J_3 \end{bmatrix} = \begin{bmatrix} (t_1)^2 & t_1 & 1 \\ (t_2)^2 & t_2 & 1 \\ (t_3)^2 & t_3 & 1 \end{bmatrix}^{-1} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

Similarly for K. Now let us give ourselves sample control points:

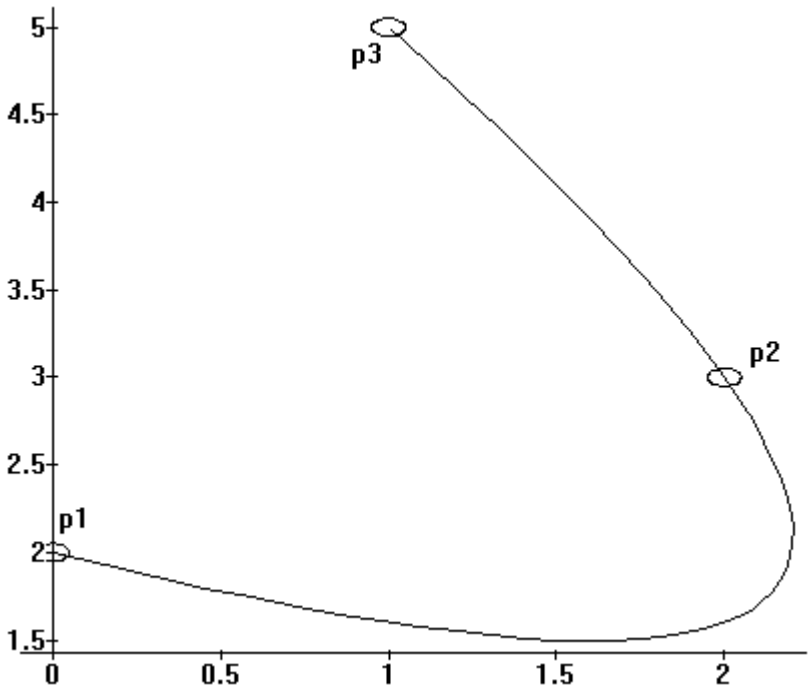
$$x = (0 \quad 2 \quad 1) \qquad y = (2 \quad 3 \quad 5) \qquad t = (1 \quad 4 \quad 5)$$

$$J = \begin{bmatrix} -\frac{5}{12} \\ \frac{11}{4} \\ -\frac{7}{3} \end{bmatrix} \qquad K = \begin{bmatrix} \frac{5}{12} \\ -\frac{7}{4} \\ \frac{10}{3} \end{bmatrix}$$

$$x = -\frac{5}{12} \cdot t^2 + \frac{11}{4} \cdot t - \frac{7}{3}$$

$$y = \frac{5}{12} \cdot t^2 - \frac{7}{4} \cdot t + \frac{10}{3}$$

Now, let us plot the spline:



And now, an example Baker spline.

The uniform Baker spline will be defined by its geometry vector V:

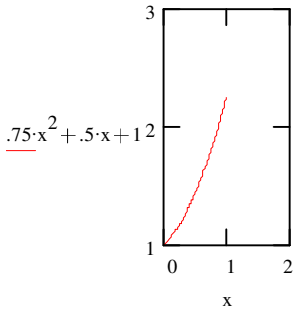
$$V=\begin{pmatrix}1\\3\\.5\end{pmatrix}$$

That is, p1=(0,1), p2=(1,3) and slope at p1 is .5. Slope at x=1 will be the slope of p2-p1, or 2.

$$\begin{pmatrix}A\\B\\C\end{pmatrix}=\begin{bmatrix}-\frac{1}{2} & \frac{1}{2} & -\frac{1}{2}\\ 0 & 0 & 1\\ 1 & 0 & 0\end{bmatrix}\cdot\begin{pmatrix}1\\3\\.5\end{pmatrix}=\begin{pmatrix}0.75\\0.5\\1\end{pmatrix}$$

The spline is defined by the quadratic y=Ax^2+Bx+C:

$$y=.75\cdot x^2+.5\cdot x+1$$



Note that the spline goes trough p0=(0,1) and apparently has a slope of 2 at x=1. This is verified because y'=1.5x+.5, thus y'(1)=2.

Now to get the equivalent geometry vector for a Zed spline. First find the transition matrix L:

$$L:=\begin{pmatrix}-1 & 1 & -1\\ 0 & 0 & 1\\ 1 & 0 & 0\end{pmatrix}^{-1}\cdot\begin{bmatrix}-\frac{1}{2} & \frac{1}{2} & -\frac{1}{2}\\ 0 & 0 & 1\\ 1 & 0 & 0\end{bmatrix}*$$

$$L=\begin{pmatrix}1 & 0 & 0\\ 0.5 & 0.5 & 0.5\\ 0 & 0 & 1\end{pmatrix}$$

The geometry vector G for the equivalent Zed spline is:

$$G:=L\cdot\begin{pmatrix}1\\3\\.5\end{pmatrix}$$

$$G=\begin{pmatrix}1\\2.25\\0.5\end{pmatrix}$$

Looking back at the definition of the Zed spline, this defines a spline that goes through (0,1) and (1,2.25), which is correct, and has a slope of .5 at x=0, which is also correct.

Frequently used uniform cubic splines

A certain number of uniform cubic splines very useful for various reasons, mostly modelling curves. Some of these will be described here. The Hermite spline is defined by four two control points P1 and P2 and 2 slopes vector S1 and S2. Bézier splines are defined by 4 control points P1 through P4 and uniform nonrational B-splines are also defined by 4 control points, with different constraints however.

A few general notes before we dive into the main material. These splines are cubics, hence they are of the form $y=Ax^3+Bx^2+Cx+D$ and the derivative of such a cubic is $y'=3Ax^2+2Bx+C$. Often, constraints will be put on y(x) or y'(x) for some x, which will then be used to figure out the basis matrix for the spline.

Hermite splines

The hermite spline is defined by two control points P1 and P2, and two slopes s1 and s2 as follows:

1- The spline interpolates through P1

2- The spline interpolates through P2

3- The slope at P1 is s1

4- The slope at P2 is s2

Since the spline is uniform, P1=(0,y1) and P2=(1,y2). The geometry vector G is (y1,y2,s1,s2). We re-write the four constraints using that:

1- y1=D

2- y2=A+B+C+D

3- s1=C

4- s2=3A+2B+C

We re-write in matrix form and solve for (A,B,C,D):

$$\begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix}^{-1} \cdot \begin{bmatrix} y1 \\ y2 \\ s1 \\ s2 \end{bmatrix}$$

$$\begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix} = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} y1 \\ y2 \\ s1 \\ s2 \end{bmatrix}$$

$$M_{hermite} = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$M_{hermite}$ is the basis matrix for hermite splines.

Bézier splines

Named after the French dude Pierre Bézier (for French people are dudes). This spline is defined by control points P1, P2, P3 and P4 as follows:

1- The spline interpolates through P1

2- The spline interpolates through P4

3- At P1, the slope of the spline is the slope of the P2-P1 vector

4- At p4, the slope of the spline is the slope of the P4-P3 vector

We re-write as mathematical constraints, noticing that P1=(0,y1), P2=(1/3,y2), P3=(2/3,y3), P4=(1,y4), the slope of P2-P1 is 3(y2-y1) and the slope of P4-P3 is 3(y4-y3). The geometry vector is G=(y1,y2,y3,y4).

1- y1=D

2- y4=A+B+C+D

3- 3(y2-y1)=C

4- 3(y4-y3)=3A+2B+C

3 and 4 can be somewhat simplified:

3- y2=C/3+D

4- y3=B/3+2C/3+D

We re-write this in matrix form and solve for (A,B,C,D):

$$\begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & \frac{1}{3} & 1 \\ 0 & \frac{1}{3} & \frac{2}{3} & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}^{-1} \cdot \begin{bmatrix} y1 \\ y2 \\ y3 \\ y4 \end{bmatrix}$$

$$\begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} y1 \\ y2 \\ y3 \\ y4 \end{bmatrix}$$

$$M_{bézier} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Convex hull

The convex hull of a set S is the smallest convex set C such that $S \subseteq C$. At this point, we need a strict definition of convex sets, then we can prove existence and uniqueness of convex hulls.

We assume a definition of line segment between x and y, such a definition can be made strict in the context of vector spaces over the reals. Then, L is defined simply as

$L(x,y)=\{tx+(1-t)y | t \in [0,1]\}$

The "natral definition" of "L is the shortest path between x and y" works well when the integral is well defined only.

Definition: A set C is said to be convex if, for ALL x,y in C, $L(x,y) \subseteq C$. (Ie, C is convex if, for all pair of points, the line segment between them is contained in C.)

Definition: arbitrary intersection $D=\cap_{i \in I} S_i$. D is the set such that $d \in D \Leftrightarrow \forall i \in I$, $d \in S_i$. Given S_i and a non-empty I, D is unique. (Proof: assume that $D1=\cap_{i \in I} S_i$, $D2=\cap_{i \in I} S_i$, $D1 \neq D2$. Assume without loss of generality that d is in D1 but not in D2. Since d is in D1, then d is in all S_i. Therefore, d has to be in D2, contradiction. QED.) Such a set is minimal in the sense that $D \subseteq S_i$ for all i in I.

Definition: Convex hull. First assume the universe U is convex (this is true for R^n). Then, the convex hull C of an arbitrary set S is defined as $\cap C_\alpha$, where $K=\{C_\alpha\}$ is the set of all convex supersets of S. Since S is in U and U is convex, U is an element of K, thus the intersection exists and is unique. It is also minimal. All we have to prove is that C is convex. But this is trivial, as we are about to show. Take an arbitrary pair of points x, y in C. We have to prove that L(x,y) is a subset of C. Since x and y are in C, x and y have to be in C_α for all α. Since C_α is convex, this means that L(x,y) is in C_α for all α. Therefore, L(x,y) is in C. This is true for any x,y in C, thus C is convex.

As an example, a triangle is its own convex hull. For a concave polygon, the convex hull is the smallest convex polygon that completely includes the concave polygon.

Now on to *convex sums*. A convex sum is a weighted sum $\sum_{1\le i\le n} w_i y_i$ such that $\sum_{1\le i\le n} w_i y_i=1$ and w_i is non-negative. A convex sum is so called because the resulting sum is in the convex hull of its control points y_i, as we are about to prove. We will here use a proof by induction. Let us first articulate the proposition we are about to prove.

Pi="The sum $S_i = \sum_{1\le j\le i} w_j y_j$ where the weights w_j are positive and sum to 1 lies within the convex hull of the control points y_j."

P1 translates into $w_1 y_1$, where w_1 is positive and $w_1=1$, is in the convex hull of y_1, which trivially true.

The next step is to demonstrate that $P_i \Rightarrow P_{i+1}$. Now let us examine P_{i+1}. $\sum_{1\le j\le i+1} w_j y_j = \sum_{1\le j\le i} w_j y_j + w_{i+1} y_{i+1}$. Let $K=\sum_{1\le j\le i} w_j$. We know that $K<1$ since all weights are positive and add up to 1. We can scale up S_i by 1/K (call that Q). Then, by hypothesis of P_i, Q is in the convex hull of the y_1 through y_i. However, $S_{i+1}=KQ+w_{i+1}y_{i+1}$ and $K+ w_{i+1}=1$. Thus S_{i+1} is on the line between Q and y_{i+1}. Since both Q and y_{i+1} are in the convex hull of y_1 through y_{i+1}, the line between Q and y_{i+1} is in the convex hull. Thus S_{i+1} is in the convex hull. By induction, Pi is true for all i, which completes the proof.

Bernstein polynomials

The nth degree Bernstein polynomial, $B_n(x)$ is defined by n+1 control points y0, y1, y2, ..., yn as follows:

$B_n(x)=\sum_{0\le i\le n} C(i,n)x^i(1-x)^{n-i}y_i$ where $C(i,n)=n!/[i!(n-i)!]$

n! is the factorial of n, is $1\times2\times3\times4\times...\times n$. As an example, 6!=720. As per the definition above, C(3,5), for example, would be 5!/(3!2!)=10.

At this point in time, it is useful to view the spline as a weighted sum of its control points. Looking at $B_n(x)$ as above, we can see that for a fixed x, we are in effect taking a weighted sum of the y_i's: $\sum_{0\le i\le n} w_i y_i$. Let us now examine the weights.

A weight $w_i=C(i,n)x^i(1-x)^{n-i}$ as above can be interpreted probabilistically using the binomial distribution. If we are repeating an experiment n (independant) times, each time with probability of success x, then w_i is the probability of the experiment succeding exactly i times. Then, it becomes obvious that $\sum_{0\le i\le n} w_i = 1$ (the sum is simply the sum of the probabilities of all possible outcomes, which has to be one).

Going back to $B_n(x)$, we can now say that the Bernstein polynomial is a *convex sum* of its control points. A convex sum is when the sum of the weights is one ad each weight is positive (as is the case here). It is so called because then the result will be in the convex hull of the control points. Of particular interest to us is the $B_3(x)$ polynomial. It can be written down as follows:

$$B_3(x) = y_0(1-x)^3+y_1 3x(1-x)^2+y_2 3x^2(1-x)+y_3x^3$$

$$=y_0(1-3x+3x^2-x^3)+y_1(3x-6x^2+3x^3)+y_2(3x^2-3x^3)+y_3x^3$$

$$=x^3(-y_0+3y_1-3y_2+y_3)+x^2(3y_0-6y_1+3y_2)+x(-3y_0+3y_1)+y_0.$$

It is obvious from this last form that we have just found an equivalent definition to the Bézier spline. With the work we have just accomplished, we have demonstrated that a Bézier spline lies within the convex hull of its control points, which is an important property, as we will see later.

Uniform nonrational B-spline

The uniform nonrational B-spline is defined as a cubic polynomial spline which has C2 continuity everywhere even when several B-spline segments are put in sequence. The term nonrational refers to the fact that we are dealing here with a conventional polynomial, as opposed to a quotient of polynomials. A rational spline is defined by the quotient of two polynomials.

The B-spline is defined by 4 control points P1, P2, P3, P4. However, let us examine exactly how we should use these to make certain that consecutive B-splines are C0, C1 and C2 continuous. That is, say we have 6 points P0, P1, P2, P3, P4 and P5. Then, the B-spline defined by P1, P2, P3 and P4 is the one we are interested in, let's call it segment S2. However, there is a spline segment defined by P0, P1, P2, P3, which we will call S1. We want S1 and S2 to have C0, C1 and C2 continuity at their joint. Furthermore, there is a spline defined by P2, P3, P4 and P5, which we name S3, and we want S2 and S3 to have C0, C1 and C2 continuity at their joint.

This section in progress, I can't find a justification for the B-spline basis matrix.

Catmull-Rom splines: a non-uniform type of spline

This spline type is defined as follows (given four control points P_0 through P_3 (x_0,y_0), (x_1,y_1), (x_2,y_2) and (x_3,y_3), x_i are increasing):

1- The spline interpolates through (x_1,y_1).

2- The spline interpolates through (x_2,y_2).

3- The spline's slope at x_0 (the second control point) is whatever slope the line from the first to the third control point is.

4- The spline's slope at x_1 (the third control point) is whatever slope the line from the second to the fourth control point is.

The spline segment thus defined is for the $[x_1,x_2]$ segment. If a similar spline is defined with the control points P_1, P_2, P_3, P_4, then they will join at P_2 and their first derivative will agree, giving it a smooth appearance. Writing the restrictions expressly in equation form, we get:

1- $y_1=Ax_1^3+Bx_1^2+Cx_1+D$.

2- $y_2=Ax_2^3+Bx_2^2+Cx_2+D$.

3- $y'(x_1)=3Ax_1^2+2Bx_1+C=(y_2-y_0)/(x_2-x_0)$.

4- $y'(x_2)=3Ax_2^2+2Bx_2+C=(y_3-y_1)/(x_3-x_1)$.

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 conics 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 as the point is the smallest entity that can be displayed on the display device. These are not necessarily 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 believe) 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 a "strange" 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. (High end platforms today use 16 bits per component, including a so-called alpha channel, for a possible total of 64 bits, but this is to minimize roundoff when combining several images together, for instance.) 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**.

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** and **HSV 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, Magenta 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 800x600 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 800 pixels would be the next 800 bytes. Then next row of 800 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*800. 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*800. 800 is 512+256+32, thus we have A+x+y(512+256+32) = A+x+512y+256y+32y. However, $512=2^9$, $256=2^8$ and $32=2^5$ Thus, the pixel memory location is $A+x+2^9y+2^8y+2^5y$. 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+y<<5.

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 preferred 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.

This algorithm has the nice property of being a special case of forward differencing. It's also fast and has no comparisons in the so-called inner loop. (Comparisons have a tendency to flush prefetch queues in CPU's, which results in relatively long delays in the inner loop).

However it has the disadvantage of accumulating roundoff error at each new pixel. This should not be a problem in general, but when utmost precision is needed, an alternate algorithm which does not accumulate error might be considered.

The algorithm works without any error accumulation (the only error is the roundoff to the nearest pixel). The idea is as follows. We first observe that the slope m is a rational of the form a/b. Let's assume m is positive and less than 1. We can make a special case for each of the 8 octants such that this is true.

Next, let's assume y_0 is of the form N+c/b, where c is some integer between 0 and b. Then, when adding the slope to the current y, we get N+(c+a)/b. However, now we need to check whether c+a is more than b. If it is, then we rewrite as: N+(c+a)/b=N+(c+a-b+b)/b=N+1+(c+a-b)/b. This means that whenever c+a is more than b, we subtract b from it and add 1 to N. The pixel coordinate in y that we actually draw is N. (This implies that we're truncating y. If we want to round off rather, we can add 0.5 to the original y_0, which will have the net effect of rounding to the closest integer. The denominator can be doubled to avoid roundoff in the .5). Pseudocode for this follows (integer endpoints are assumed, this can be generalized to rational endpoints of course).

```
Let (x0,y0) and (x1,y1) be the endpoints of the line segment, such
that (x1,y1) is in the first octant relative to (x0,y0).
Let a=2*(y1-y0)
Let b=2*(x1-x0)
Let N=y0
Let c=(x1-x0)

for x varying from x0 to x1 by steps of one, do
      putpixel(x,N)
      add a to c
      if c>=b then
            subtract b from c
            add 1 to N
      end if
end for
```

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. This is called the **even-odd rule** by the industry. It is recommended that you try this with the star above and note 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 segment 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 empty
```

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 it is hard to add any polygons to a scene thus rendered, or to deform it in a nonlinear way. 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 algorithms 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 within 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 algorithms is not the subject 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 **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.

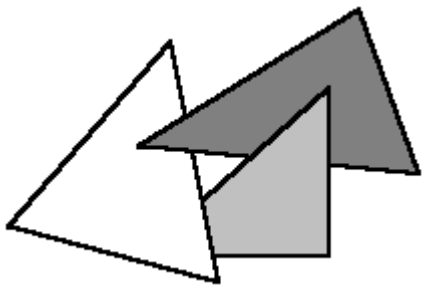
```
Let i[0..256] be 257 indices, initialized to 0
```

```
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+1]
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 include the **quick sort**, **heap sort**, **insertion sort** and **merge sort**. These will not be discussed here, they each have their advantages and drawbacks (for a full discussion, see [2]).

Painter's algorithm and depth sorting

As was previously mentioned, 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 whether 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 whether 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**. Cutting up the triangle into pieces (clipping) will be described later.

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 number 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 (we did this in the perspective chapter). 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 whether 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

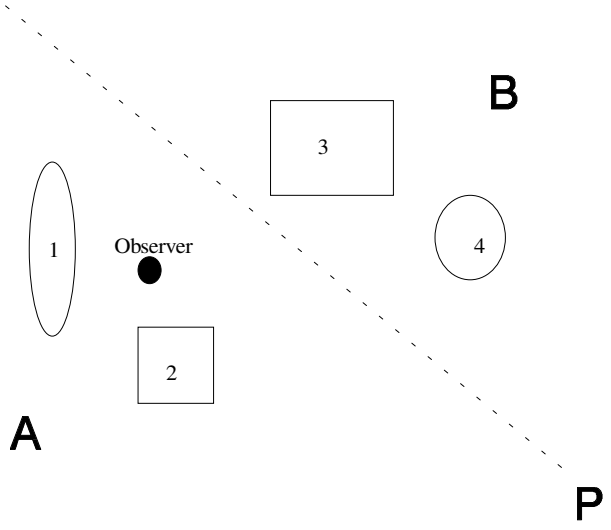
M=A/(Dp), N=B/(Dq), K=C/D

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.

Note that visible surface determination can be performed in a clever way using **A-Buffering** (described in the **antialiasing** section).

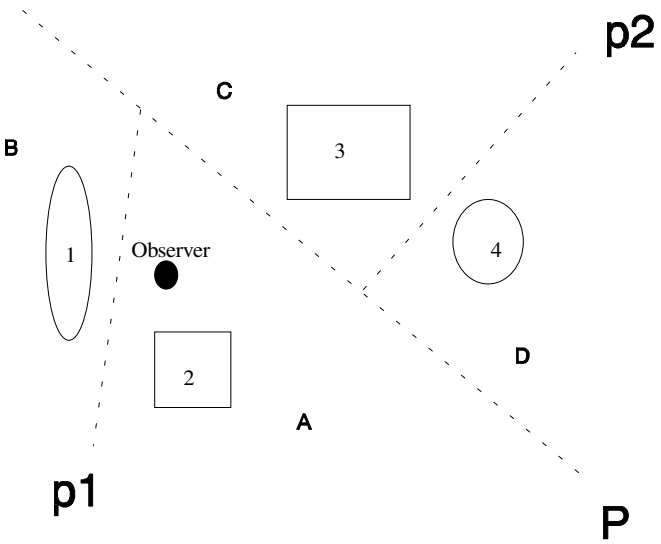
Binary Space Partitioning

Let us assume we have the description of a scene. Let us cut the scene with a plane P. (That is, choose a plane that splits the scene into two halves.) The crucial point in BSP is that anything on the same side of the plane P as the observer can **not** be obscured by anything on the other side of the plane P. Therefore, if we can split the space with a plane P in a side A, in which the camera lies, and a side B, which is the other side, then we can draw everything in B then everything in A, and the drawing order between objects in B and A will be correct. Note that we still need to somehow determine what is the correct order within B and A.

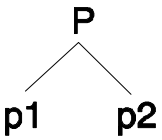


In the example above, we show the partitioning plane P with a dotted line, objects are rectangles and ellipses, and the two areas are marked A and B. The observer is the dark spot marked "observer". In this case, objects 1 and 2 are on the "A" side of the P plane, therefore it is impossible for them to be obscured by any object in the B side (namely objects 3 and 4). This is true *no matter where in A the observer lies*. The observer can be anywhere in the A region and objects 1 and 2 will never be obscured by objects 3 and 4 for the observer.

However this is still not sufficient. We still need to have a drawing order for objects in the A region and objects in the B region. Therefore, we recurse the algorithm and split the A region into two sub regions for which the ordering becomes unambiguous and similarly for the B region, as seen below:



Now there are four regions, A, B, C and D. Space is partitioned at the root by P. Then, the two resulting subspaces are partitioned by p1 and p2. This can be represented by the following tree:



When we want to get a drawing order for objects 1, 2, 3 and 4, we traverse the tree as follows:

a) Start with the root P, find on which side of the plane the observer lies. That is the AB side. The opposite side is the CD side. So first, draw the CD side, then the AB side.

b) Draw CD side. Find on which side of p2 the observer lies. The observer is on the C side of p2, the opposite side is D. So first draw D then draw C.

c) The D side has only one object, draw it (so we draw object 4 first)

d) The C side has only one object, draw it (the second object we draw is object 3)

e) Draw the AB side. Find on which side of p1 the observer lies. The observer is on the A side of p1, so first draw the B side then the A side.

f) The B side has only one object, so draw it (the third object we draw is object 1)

g) The A side has only one object, so draw it (the last object we draw is object 2)

The drawing order generated by this algorithm is therefore 4,3,1,2. This ordering is correct.

Sometimes it might be impossible to find a plane that neatly splits space into two sections. When this happens, you can just pick any plane and slice objects apart with it. That is, if a plane intersects an object, slice the object into two sub-objects that do not intersect the splitting plane.

If we have a planar object that is exactly on the splitting plane, then the drawing order can be tweaked slightly to draw it:

1) Draw everything that's on the opposite side of the plane from the observer as usual

2) Draw everything that's on the plane

3) Draw everything that's on the same side of the plane as the observer as usual

If the observer is on the splitting plane, the drawing order is not important.

For polyhedral objects, this can be used efficiently as follows. Instead of arbitrarily picking planes and splitting space, pick a polygon's plane as partitioning plane.

When generating a BSP, you get a binary tree representation of your scene. If you used the planes of the polygons as partitioning planes, you have an additional mild bonus. The leafs of the tree are either inside the polyhedron or outside. That is, the regions in space described by the leafs of the tree are either totally outside the polyhedron or totally inside it. This can be used for simple collision detection of polyhedron with points, even though it is not always efficient to do so. (Efficient collision detection is beyond the scope of this text. For a good starting point, see [4]).

All this brings up the subject of generating an optimal BSP tree. There are many problems with that. The first is the definition of optimal. In the collision detection case (which is not too efficient anyway, but deserves mention), optimal often means shallowest tree. For visible surface determination, minimizing the number of triangle clipping is important, as well as minimizing the number of nodes (which is a consequence of clipping). In both cases, the problem turns out to be extremely hard (NP-Hard). In the "minimizing the tree depth" field, a greedy algorithm that picks the plane that splits space as evenly as possible might do well. However, when trying to minimize the number of polygon clips, it's harder to get good heuristics.

Merging BSP trees is also a very tough problem. Basically it's at least as hard as generating a new BSP tree from scratch. However, it is possible to cut corners.

If we have a very large maze-like scene (for example), and a small object navigating through it, we can do as follows. We treat the small object as a point, just like the observer, and traverse the BSP tree of the maze scene to find where the "punctual" object belongs in the BSP tree. Once we have found it, we insert the "punctual" object's BSP tree at that point in the maze BSP tree. This will work relatively well so long as the small object dos not come close enough to corners in walls to cause ambiguities in the display ordering.

It is also possible to insert several objects this way in the maze BSP.

This algorithm can be very efficient if we have many objects spread over a large area where inter object ordering can be determined easily, but the objects themselves are complex so intra object visual surface determination is nontrivial.

Note that a BSP can be processed through an affine transform and still remain valid if proper care is taken. This means that we can move a binary space partitioned object around, or we can move the observer in the BSP object (these two are equivalent anyway) without fear of the algorithm crumbling. Therefore, it is possible to have a few flying BSP objects at the same time, for instance.

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 brain and eyes. 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 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 $\cos\theta$, 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. Theoretically, 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. The reflection vector R is the direction in which the light should be most reflected by the surface. R can be shown to be

$$R = 2A(A \bullet B) - B$$

(remember that A is the plane normal and B the light vector and note that A and B should be normalized).

Let V be the vector from the observed point to the eye, normalized. Then, a function of R•V can be added to the calculated intensity (before fallout due to distance is taken into account), which will add a specular like highlight to the shading. Mr. Phong Bui-Tuong used the following specular component:

$k \times (R \bullet V)^{\alpha}$

where α is the so-called **specular reflection exponent** and k is the **specular reflection coefficient**. The larger the α, the more punctual the reflection. The larger the k, the more intense the reflection. Values of α around 1-5 yield almost no specular reflection, while a value of 300 or more yields a sharp spot. Note that if a too large k is chosen, the image will look washed out or overexposed.

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 computations which will not be discussed here.

Smooth shading

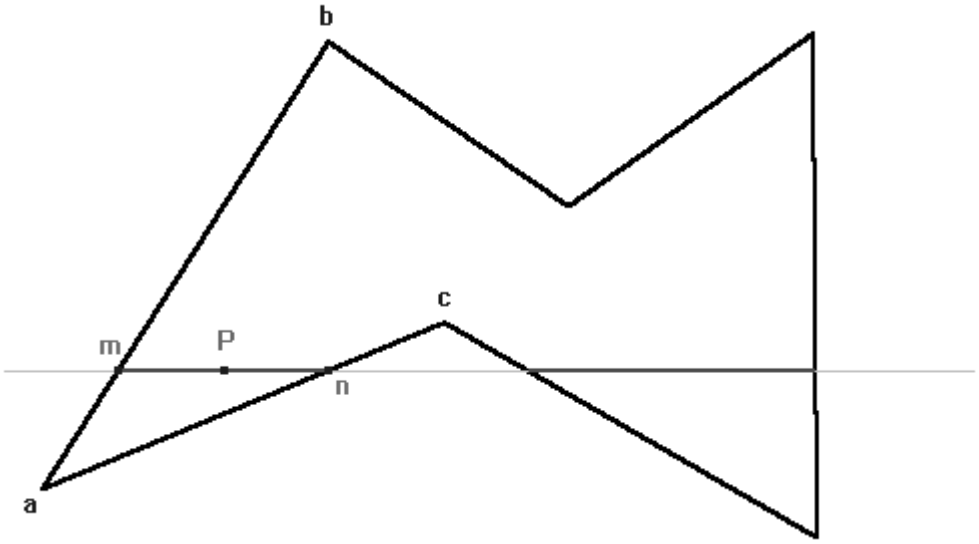
The simplest form of polygon shading calculates one value of intensity and uses that value across the whole polygon.

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 intensity 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.



As can be seen above, intensities are calculated for all vertices, particularly vertices a, b and c. Then, intensity is linearly interpolated 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.

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. Specifically, we know that:

Ax0+By0+C=i0

Ax1+By1+C=i1

Ax2+By2+C=i2

Which can be represented in matrix form as:

$$\begin{pmatrix} x0 & y0 & 1 \\ x1 & y1 & 1 \\ x2 & y2 & 1 \end{pmatrix} \cdot \begin{pmatrix} A \\ B \\ C \end{pmatrix} = \begin{pmatrix} i0 \\ i1 \\ i2 \end{pmatrix}$$

or,

XK=G, where

$$X = \begin{pmatrix} x0 & y0 & 1 \\ x1 & y1 & 1 \\ x2 & y2 & 1 \end{pmatrix} \qquad K = \begin{pmatrix} A \\ B \\ C \end{pmatrix} \qquad G = \begin{pmatrix} i0 \\ i1 \\ i2 \end{pmatrix}$$

Therefore, we have that $K=X^{-1}G$, which solves for K.

As a special note, it should be remembered that a similar process can be used for any type of linear interpolation across the surface of a polygon.

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 (not to be confused with the **Phong illumination model**) 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 a side note, the interpolation of x, y and z can be done as we just saw for Gouraud shading.

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 some nice metal/chrome/mirror effects.

A note on the "mirror" effect. If we imagine that we have a sphere centered on our object with an extremely large radius, and the in side of the sphere is paved with a texture (example: stars & stellar objects) and the object has a mirror surface, then the environment (textured sphere) should be reflected on it. The perspective calculations and other things make this complex. However, we can simplify things this way. We assume that the vector from eye to object (eye vector) is constant over all of the surfaces of the object (which is normally true only in parallel projections, but will be almost true if the object has little perspective distortion). Second, we assume that the sphere has a large radius enough that the point of the sphere which is reflected by a point of the object only depends on the eye vector and the surface normal.

In this case, we can interpolate the surface normal Phong-style, then use that to compute the reflected point from the sphere using the eye vector. However, the computations are still quite expensive. We can simplify them by using the hack where we interpolate x and y linearly and then set $z=(1-x^2-y^2)^{1/2}$. Then, the normal vector of a point on the surface is entirely determined by x and y. In this case, the reflected point on the sphere depends on x, y and the eye vector. What we can do is assume a fixed eye vector, then make a lookup table (which is then only dependant on x and y, which is manageable) to find what point on the sphere it reflects to.

Hence this simplifies to interpolating the (x,y) component of a surface normal across the screen, then looking it up in a lookup table that contains the color of the corresponding reflected point on the sphere.

Texture mapping & variants on the same theme

Texture mapping is the process by which we give a polygon its own planar coordinate system, with two base vector that lie in the polygon's plane, and a vector for the position of a point in the plane. Specifically, if u and v lie in the plane of the polygon, and w is a point in the plane of the polygon (for example, a vertex), then the plane equation for the polygon can be written as:

au+bv+w

where (a,b) are the texture coordinates on the polygon. Once we have (a,b), we can assign different properties to different (a,b) pairs. For example, we can make the color of the polygon a function of (a,b), which corresponds to classical texture mapping. Another thing we can do is perturb the surface normal of the polygon with some function of (a,b), which corresponds to bump mapping.

As it is, the Phong shading approximation we saw in the last bit of the preceding section is essentially a texture mapping trick.

Note that it is possible to have several different coordinate systems for the same polygon, if several different textures have to be applied (ie, one for the actual texture mapping, another one for the phong shading, another one for bump mapping, and who knows what else).

What we have said in this section up to now is (relatively) independent of the projection used. Now we will consider the type of projection used.

In a parallel projection, linear interpolation, just like we did for Gouraud, across the projected surface is correct (so long as the surface is planar). However, when perspective projecting, linear interpolation is generally wrong. For an elaborate discussion of the texture mapping equation in the perspective projection case, see the perspective chapter.

If the plane of the perspectively projected polygon is perpendicular to the z axis, then linear interpolation is exact. As the angle between the plane of the polygon and the z axis moves away from 90 degrees, linear interpolation becomes more and more wrong. If the polygons are small enough on screen, the perspective distortion might now show, but for larger and more angled polygons, it is quite apparent.

Linear interpolation may suffice for some purposes on low end platforms and games, but a correction for perspective will definitely be needed for more serious applications, as discussed in the perspective chapter.

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.

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, $a \oplus b$ 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, $a \ominus b$ is defined to be (a-b)%n. As an example, let's assume we are working in modulo 8 space. Then,

$3 \oplus 2 = 5\%8 = 5$ $5 \oplus 6 = 11\%8 = 3$ $4 \ominus 3 = 1\%8 = 1$ $4 \ominus 7 = -3\%8 = 5$

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.

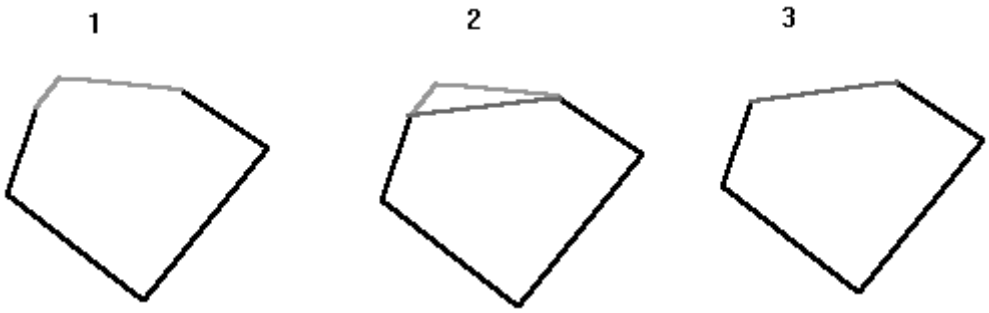
The point of that first step was to make at least one correctly oriented normal. Then, start following the edges and generate correctly oriented normals as follows.

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

$$A(u,v)=1/2\times[(u0-u1)(v0-v2)-(v0-v1)(u0-u2)]$$

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

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•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.

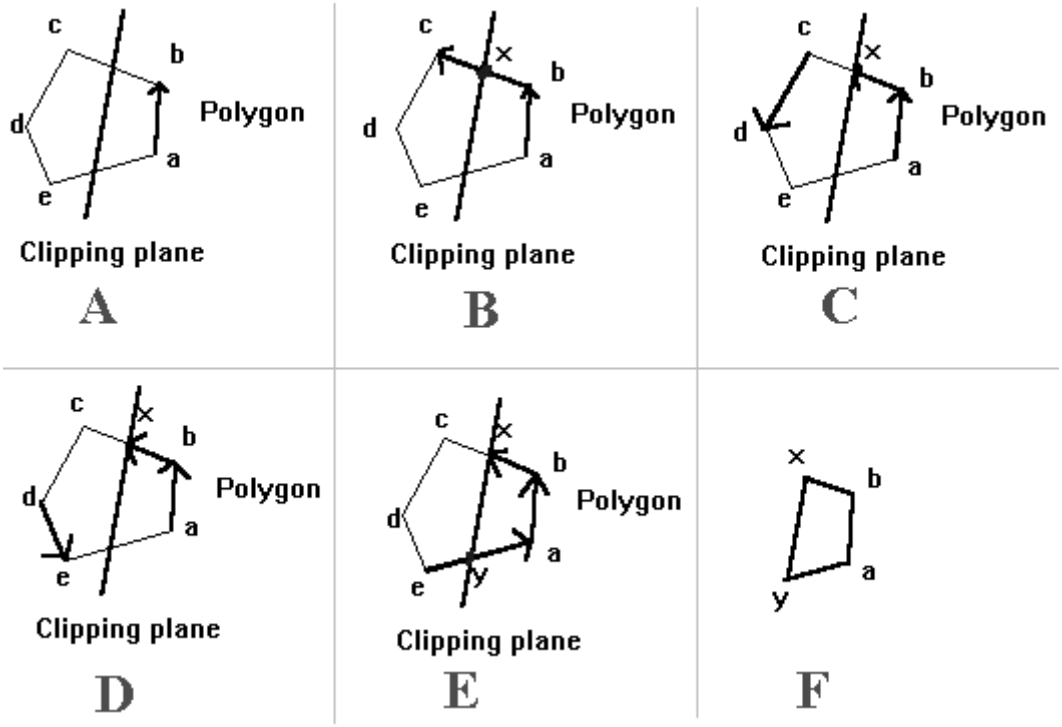
One very good way of propagating the normals is to start with one of the initial faces for which we generated the normal, and then do a depth first search through connected faces. The depth first search is elementary and will not be discussed here because it is not absolutely necessary, though it will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

Quaternions

Introduction

In this chapter, a good understanding of basic mathematics is assumed. I also have been extremely lazy and I'm not giving any justifications whatsoever. Someday, hopefully, this will be righted. It is possible to do very much everything that is described here using only matrices with a good knowledge of linear algebra. For instance, as a first step, we can find a matrix that rotates about an arbitrary unit vector by a specific angle. Then, if we want to interpolate linearly between two orientations, we can proceed as follows. First find the matrix M that expresses the transform from the first orientation to the second orientation. Then find the eigenvector of that matrix, this is the desired axis of rotation. Then linearly interpolate the angle from 0 to whatever angle the two orientations make.

Let us recall **complex numbers**, which have the form a+bi, where $i^2=-1$. These complex numbers can also be represented in polar or **exponential form**. I will interest myself mainly with the latter. The exponential form of a complex number is $Z=r\times\exp(i\theta)$, where (r,θ) are the polar coordinates of Z in the complex plane. All points in the plane can be represented by a complex number. Let B be a unit quaternion (that is, of the form $B=\exp(i\theta_0)$). Recall that, when multiplying two complex numbers, we multiply the modules and add the angles. Let us multiply Z by B. Then, we get $ZB=r\times\exp(i\theta+\theta_0)$, or Z rotated by θ_0.

The morale of this story is that unit complex numbers can be used to represent rotations in the plane.

We will now develop a similar tool for modeling rotations in 3d. A quaternion Q will be of the form Q=W+Xi+Yj+Zk, where 1, i, j and k are **linearly independent** quantities (and i, j and k are linearly independent imaginary numbers). Then, we will define quaternion multiplication, by using the following basic rules:

$i^2=j^2=k^2=-1$, ij=-ji=k, jk=-kj=i, ki=-ik=j.

It will so turn out that, similarly to unit complex numbers represent rotations in the plane, unit quaternions will represent rotations in 3d.

Obviously, this is a lot of work for "just another representation of rotations in 3d", especially where matrices work fine. Indeed, even if one is using quaternions to represent orientations, one will typically convert to a matrix when actual transformations are required. The main advantage of quaternions is for interpolating between two orientations in a useful manner, and mayhaps also for a compact representation of orientations.

Preliminaries

Several notations can be used to write quaternions. The following are all equivalent:

Q=W+Xi+Yj+Zk

Q=<W, A> with A=(X,Y,Z) (a real number W, with a 3d vector (X,Y,Z))

Q=(W,X,Y,Z) (a 4d vector)

We can define a **multiplication** operation that respects the following:

$i^2=j^2=k^2=-1$, ij=-ji=k, jk=-kj=i, ki=-ik=j.

<a1, v1>×<a2,v2>=<a1a2-v1•v2 , s2v1+s1v2+v1×v2>

Of course, a1 and a2 are real numbers and v1 and v2 are 3d vectors.

The **norm** or **module** of a quaternion Q=(W,X,Y,Z) is defined as the euclidian norm of vector (W,X,Y,Z). A **unit quaternion** is a quaternion for which its norm is 1.

The **conjugate** of quaternion Q=<W,A> is defined as Qc=<W,-A>.

If we have the quaternion

Q=(0,v)=(0,X,Y,Z)

and the unit quaternion

U=(cos(θ/2),sin(θ/2)×V)

where V is a unit vector, and

Q'=(0,v')=U×Q×Uc

then it can be shown that v' is v rotated about V by an angle of θ.

Note that this implies that the two unit quaternions Q and -Q represent the same rotation.

Conversion between quaternions and matrices

The quaternion Q=(W,X,Y,Z) (assuming Q is unit) is equivalent, when interpreted as a rotation, to the matrix:

$$\begin{bmatrix} 1-2\cdot Y^2-2\cdot Z^2 & 2\cdot X\cdot Y-2\cdot W\cdot Z & 2\cdot X\cdot Z+2\cdot W\cdot Y \\ 2\cdot X\cdot Y+2\cdot W\cdot Z & 1-2\cdot X^2-2\cdot Z^2 & 2\cdot Y\cdot Z-2\cdot W\cdot X \\ 2\cdot X\cdot Z-2\cdot W\cdot Y & 2\cdot Y\cdot Z+2\cdot W\cdot X & 1-2\cdot X^2-2\cdot Y^2 \end{bmatrix}$$

By examining the above matrix, it is easy to find, given an orthonormal orientation matrix, the corresponding quaternion. First, compute W.

$m_{11}+m_{22}+m_{33}=3-4X^2-4Y^2-4Z^2$

$(m_{11}+m_{22}+m_{33}+1)/4=1-X^2-Y^2-Z^2$

But $|(W,X,Y,Z)|=1$ hence $W^2+X^2+Y^2+Z^2=1$ hence

$W=[(m_{11}+m_{22}+m_{33}+1)/4]^{1/2}$

Then, compute X, Y and Z.

$X=(m_{32}-m_{23})/4W$

$Y=(m_{13}-m_{31})/4W$

$Z=(m_{21}-m_{12})/4W$

Orientation interpolation

Often, we might only have an initial orientation, and a desired orientation at some point in the future, and no data in between. In this case, it might be desirable to interpolate the orientations between these two key orientations. Assuming no other data is available, quaternion interpolation is very appropriate.

This could be done with **Euler angles**. However, several problems arise when we do so. We can very difficultly control the exact path of the rotation, or the speed for that matter. We also get a problem called "**Gimbal lock**", where the object appears to stop turning for a brief moment, and then starts again in an odd direction.

Using quaternions, we get none of these problems. Speed of rotation can be made constant, and the path of the rotation will be that of the shortest arc. However, we get numerical problems when the rotation is close to 180 degrees. Also, we will need to interpolate through more than 2 quaternions if we want to rotate by more than 180 degrees.

The idea is to picture the unit quaternions as being on the unit **hypersphere** in 4d. Then, we can find the shortest arc over that hypersphere between these two quaternions, and interpolate linearly the angle along that arc.

Given two unit quaternions, q1 and q2, let us first find the angle between them. By definition of angle as a function of dot product, we have:

$\phi=\text{acos}(q1 \bullet q2)$

Then, let us have a parameter t that varies from 0 to 1. Then, the quaternion given by:

$$q(t)=\frac{\sin((1-t)\cdot\phi)}{\sin(\phi)}\cdot q1+\frac{\sin(t\cdot\phi)}{\sin(\phi)}\cdot q2$$

will give uniformly distributed orientations between q1 and q2 as t varies from 0 to 1. In particular, q(0)=q1, q(1)=q2. This is called **spherical linear interpolation**, or **SLERP**.

Note that we didn't check if we were using the shortest arc or the longest on the great circle. In order to take the shortest path, we simply check that

$(q1-q2)\bullet(q1-q2)>(q1+q2)\bullet(q1+q2)$

If that is false, replace q2 by -q2 (this is allowed since Q and -Q represent the same rotation).

The above formula will have numerical difficulties when ϕ is close to 0 or π. When ϕ is close to 0, we can replace the SLERP with a simple linear interpolation. E.g. q(t)=(1-t)q1+tq2. When ϕ is close to π, we need to add more keyframes (intermediate quaternions).

Antialiasing

Introduction

In many places, we have approximated continuous phenomena by sampling it at discrete intervals, and then reconstructing an image with these samples. As an example, we have studied shooting one beam of light through the center of a pixel and see what it intersects and then coloring the whole of the pixel by whatever color corresponds to what it hits.

This might result in very inaccurate and sometimes unsightly pictures. As an example, take a black and white tiled floor for which the tiles are 0.25 meters wide, which we sampled at intervals of .5 meters. Then, we will have either all white samples or all black samples. This does not represent the color of the floor, which should either be a mix of black and white, or, in a certain sense, gray.

This is an example of aliasing in space, but numerous other types of aliasing exist. For example, say we are making an animation by generating several pictures of a scene with moving object and then displaying the still frames quickly (say, 24 frames per second). In our model, if an object moves fast enough, it might appear to jump around on the screen. For example, if the object is 1 centimeter big, but moves so fast that in 1/24th of a second (the duration of a frame), it's 5 centimeters away from where it was before, the object will appear to jump very drastically. Or even worse, an object could pass right in front of the camera in between frame a and b, but not be on neither frame a nor b.

In this section, we will introduce a few of the techniques that can be used to fix these problems.

Filtering

Filtering, in the way that we want to use it, is essentially a weighted average of a signal. Filtering theory is extensive, but we are not concerned too much by it here. I will just mention that it is possible to take the transform (Fourier, wavelet or other) of a signal and then remove very small amplitude components and/or "high frequency" components. This is what is traditionally viewed as filtering.

In our case a filter is represented by a filtering mask. We'll start with 1d cases. If we have a 1d signal (think of it as a single scanline in a pixmap, or maybe some sampled sound), then we have something like s_1,s_2,s_3,s_4, ..., s_n, where the s_i's are sampled data. Here, we have n samples. Then, we might want to create the samples s_1', s_2', s_3', ..., s_{n-1}', where $s_i'=(s_i+s_{i+1})/2$. This is an example of a filter, sometimes referred to as a "low-pass filter". This is another legitimate filter: $s_i'=.25s_i+.75s_{i+1}$. This particular filter is said to be *biased* towards s_{i+1}. In general, the class of filters as_i+bs_{i+1} can be represented by the vector equation $(a,b)\bullet(s_i,s_{i+1})$. (a,b) is said to be the filter mask. In the last example we gave, the filter mask is (.25, .75).

In general, filters are applied by taking the weighted sum of several samples. We normally like that the sum of the components of the filter mask be one, in which case the filter tends to not change the overall intensity of the image. If the sum of the components of the filter is not one, we get very different effect. As an example, the filter (-1,2,-1) is sometimes called the differential filter, for it approximates the derivative of a signal. It is generally not very useful as an approximation to the derivative, however it tends to highlight contrasts in a signal and can be used to help in edge detection, for example. This filter can be written as $s_i'=-s_i+2s_{i+1}-s_{i+2}$.

Two dimensional filters are generally more useful to computer graphics people. They are usually written in matrix form, $F=(f_{ij})$. This is an example filter, called the **box filter**.

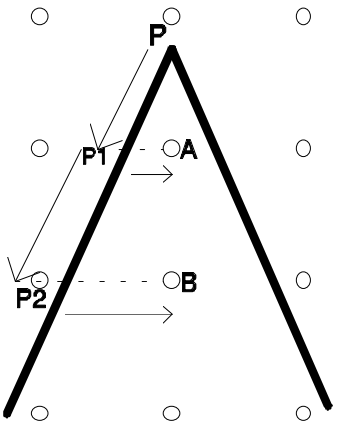
$$\begin{bmatrix} \frac{1}{16} & \frac{2}{16} & \frac{1}{16} \\ \frac{2}{16} & \frac{4}{16} & \frac{2}{16} \\ \frac{1}{16} & \frac{2}{16} & \frac{1}{16} \end{bmatrix}$$

```
q[i][j]=(p[i][j]+2p[i][j+1]+p[i][j+2]+

        2p[i+1]+4p[i+1][j+1]+2p[i+1][j+2]+

        p[i+2][j]+2p[i+2][j+1]+p[i+2][j+2])/16
```

Where q[i][j] is the filtered sample, and p[i][j]'s are the original samples.

These filters are made so as to remove very high frequency from images, which are usually poorly represented. (High frequency component here is taken in the Fourier series sense.)

Pixel accuracy



When applying any type of shading except flat shading to a polygon, pixel precision becomes an issue. We often calculate incrementally some value per edge (for example, in gouraud shading, we interpolate the shading linearly along the edge, and in texture mapping, (u,v) texture coordinates get interpolated along edges and across the polygon).

Referring to the figure above, an example will be given with Gouraud shading. The small circles represent the pixels, and the dark edges are the edge of a triangle. With Gouraud shading, we start at P and we are given some initial color. Then, we want to interpolate down the edge. However, it is important to notice that P is less than one scanline above the scanline of P1. Therefore, the vector drawn above from P to P1 is shorter than the vector from P1 to P2, also drawn above. This needs to be taken into consideration to get correct pixel accuracy. Furthermore, once we have the correct color at P1, we can't simply put that color into pixel A even though the distance between P1 and A is less than one pixel. We have to take into account the distance from P1 to A into our calculations. As a matter of fact, in this particular example, A is roughly halfway between the left and right edge so its color should be roughly the average color of the two edges.

Then, as we go to the next scanline, we need to perform our calculations starting from P1. Given the color at P1, we find the color at P2, and then we need to calculate the proper color for pixel B by taking into account the distance from P2 to B. This goes on for the whole triangle or polygon.

This example can also illustrate how edge pixels should be considered. First of all, edge pixels should be extremely rare. As can be seen in the example above, not one pixel is exactly on an edge. In the very rare case when a pixel is exactly on an edge, you can use a simple disambiguating rule to decide one it, such as "draw the pixel only if it's on a left edge, not on a right edge". Another way of doing this is of thinking of pixels as having irrational x coordinate and rational y coordinate so that edges with rational endpoints have no hope of going through them.

Sub-pixel accuracy

The other way of improving the quality of our images is by doing all calculations at some higher accuracy. The intuitive way of doing this is to create a high resolution image and scale it down to the display resolution afterwards, which ought to produce a better looking image.

It turns out that this works well, and is in fact hard to beat. However, it is rather expensive, both in rendering time and storage space, and we will attempt to look into alternate algorithms as well.

When scaling an image down, there comes the question of how pixels should be averaged. This boils down to picking a filter, as in the previous section, and applying it to each region of the pixmap that gets scaled down to one pixel. Note that the uniform filter (the filter in which each pixel has equal weight) might not, as could be thought at first, be the best choice. The box filter is already a better choice. However, a uniform filter is better than no filter, and one should be considered for a real-time system if it is easier to achieve than a generic filter.

An alternative to generating a fully blown high resolution picture is to adaptively increase the resolution. If you find that several small objects get drawn in the same pixel, subdivide that pixel into a small bitmap and calculate more precisely for that pixel. This can be recursed at will, for arbitrary precision. However, this process is still somewhat expensive, and has the disadvantage that it has to be more or less hard-wired all over the graphics engine. It is hard to perform this in real time.

A very attractive alternative is the so-called **A-buffer**.

For each pixel, a 4x8 subpixel grid is associated. However, instead of having full R,G and B components, they are merely bitmasks. Then, for all pixels, the following is done. A list of all polygons covering this pixel has to be generated. Then, the 4x8 bitmask of whatever is covered by a polygon in that pixel is generated. Then, using and's the amount of overlap can be determined, etc... and color can be computed using this (more on this later).

Let us use an example. Let's say that the following polygonal piece is in a pixel:

This is represented by the following bitmask:

```
00000000   0000 1111     111 00000
000 1 0000 = 000 11111 and 1111 0000
000 1 0000   000 11111     1111 0000
00 111 000   00 111111     11111 000
 (A)          (B)          (C)
```

Notice that (A) is the bitmask for the part of the triangle that covers the pixel, while (B) and (C) are the bitmasks for what's inside of the left and right edge, respectively. Thus, bitmasks are computed for each edge and are **and**'ed together to get the bitmask for the triangle. The bitmap can also be made by some exclusive or of bitmasks (this latter method might be a bit less useful).

If we then **or** all bitmasks for all polygonal pieces together, we get a bitmask that tells us what portions of the pixel are covered by any polygons. Then, using **and**, we determine which pieces of polygons overlap. We might also need the Z value of the pixel per polygon (not for each bit in the bitmask, only for the whole pixel), so that we can determine which bitmasks are in front of each other. What follows is a suggested algorithm for drawing a pixel:

```
Let B[i] be the bitmask of polygon i, i varies from 1 to n (number
of polygons covering the pixel).
Let C[i] is the color of the polygon i.
Let M be the following bitmask:
      11111111
      11111111
      11111111
      11111111
Let P be the pixel color, initialized to 0 (black) (either a color
vector, or a monochrome intensity)
Let K be the background pixel color
Assume polygons are sorted front to back (ie, polygon i is in
front of all polygons greater than i).
Let #X be the number of bits that are set in X

For j=1 to n, do
      foo=B[j] AND M
      P=P+C[j] * #foo / 32
      M=M XOR foo
End For
P=P+K * #M / 32
```

Note that 4x8 bitmasks have the nice property of being 32 bits numbers, which can be manipulated very easily on today's platforms. The algorithm can very easily be adapted to 4x4 bitmasks (for 16 bit machines, for example) or 8x8 bitmasks (for 64 bits machines for example).

Also note that this algorithm works relatively well if polygons are depth sorted front to back and then drawn. It can't really be drawn strictly back to front.

Time antialiasing, a.k.a. motion blur

We already mentioned the time aliasing effect. The most popular method of performing time antialiasing is still to generate several frames at very close interval and merge them together. Once more, different filters can be applied.

One has to be careful when performing motion blur. Visible motion blur is something that should only happen when things are moving faster than the frame rate allows us to see. For example, an animation going at 24 fps (frames per seconds) should not have a trail behind an object moving one millimeter per frame on the screen. Put in different terms, an animation going at 24 fps should not show in a frame an event that happened 0.25 seconds ago. This effect, called persistence, can be quite annoying, and though it can produce an interesting result, it is generally not interesting or realistic to do so.

Therefore, if we want to generate a time antialiased animation at 24 fps, we really need to generate something like an 120 fps animation, and then apply a filter to merge the frames four by four and get a 24 fps animation. This is, as can be seen, quite expensive. However, it tends to produce very nice looking pictures. If the frame rate isn't kicked high enough, a fast moving object will appear (in our example above) as 4 distinct though semitransparent images per frame. This is probably undesirable, but little can be done if objects are moving fast enough.

Other approaches have been attempted, mostly in raytracing, and will not be discussed here. One notable area that might be worth further interest is the extruding of polygons as volumes in 4d with time as the fourth dimension and trying to get the motion blur from that. This would have the advantage of being much more exact than anything presented here, all the while not having problems with very fast moving objects. However, even if the exact path of the object is known, extrusion along that path might be too expensive or difficult to perform, so linear extrusion might have to be considered. Nevertheless, I suspect this would produce very attractive results.

Mipmapping

Texture mapping was described previously. However, a naive approach was taken; we did not consider what happens when the texture is so scaled down that it takes several texels to cover a single pixel. In the context of this chapter, however, we would like to somehow "average" (more properly, filter) the texels that cover a single pixel to get a nicer looking picture. To illustrate the problem, if the texture is so shrunk that it barely covers one pixel, then the color of that pixel will be a more or less random point from the texture. Hence, if the texture is not very "smooth", this particular pixel can "blip", change color rapidly and scintillate. This is of course undesirable. Also, even though the texture seen from afar might be blue for the most part, a small reddish region might result in the whole pixel appearing red, which would be wrong.

Performing different approaches can be taken to this antialiasing problem. We can actually compute all the texels that fall within a pixel, then apply some filter, perhaps even based on the size of the texel in the pixel. These sort of approaches lend themselves very poorly to real-time applications, which is the main interest of this document, hence we will not approach them.

A real-time alternative is Mipmapping. The texture map is pre-filtered to different degrees, and at run time, we determine how much the texels get squished by the perspective transform, and the select the proper mipmap.

Uniform Mipmapping

Classically, this is how mipmapping is done. We start with a texture map. Say the texture map is of size 64x64. Then, mipmaps of size 32x32, 16x16, 8x8, 4x4, 2x2 and 1x1 are generated by recursively averaging 2x2 blocks of pixels. We start with the 64x64 texture map, then average each 2x2 block to get a 32x32 mipmap. Then we filter that mipmap again to get a 16x16 mipmap, and so on. The mipmaps can be labelled "mipmap #1" for the unfiltered texture map, "mipmap #2" for the first mipmap, and so on.

The mipmap should be chosen based on the amount of squishing we think the texels will undergo. If we feel that each pixel will cover about 4 texels (roughly a 2x2 block), then we should use the 32x32 mipmap. If the texmap is even more squished (perhaps 16 texels per pixel, 4x4) then the 16x16 mipmap should be used.

Let's look at memory requirements. Let K be the amount of memory used by the basic texture map. First we observe that averaging 2x2 block of pixels and making the first mipmap makes something which takes one fourth as much memory as the texture map. As a matter of fact, each time we generate another mipmap, the memory taken by the new mipmap is 1/4 that of the old. Hence the total memory is:

$K+K/4+K/16+K/64+K/256+...+K/(4^n)<K+K/4+K/16+...+K/(4^n)+...$

=K(1+1/4+1/16+1/64+.....)=4K/3

Hence, classical mipmapping takes but 1/3 more memory than straight texture mapping.

The problem of determining which mipmap to use is, however, not trivial. As mentioned previously, this should be a function of the "squishing" undergone by the texels. However, the squishing in the x direction can be vastly different from the squishing in the y direction. You can fight with this, or you can invent a newer type of mipmapping.

Nonuniform Mipmapping

Instead of scaling the texture map homogeneously (by the same factor in x and y), we can generate mipmaps with nonuniform scaling. This way, if there is a lot of "squishing" in a direction, but very little in the other, a relatively good mipmap can still be found.

Still using the 64x64 example above, we would generate mipmaps of the following sizes:

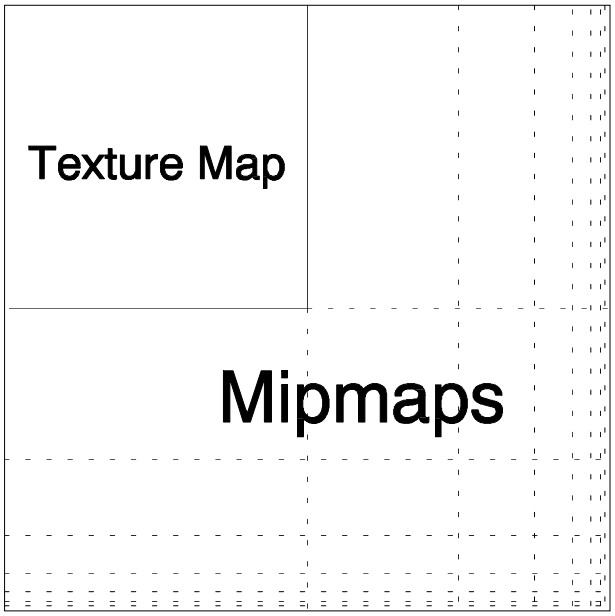
64x64	64x32	64x16	64x8	64x4	64x2	64x1
32x64	32x32	32x16	32x8	32x4	32x2	32x1
16x64	16x32	16x16	16x8	16x4	16x2	16x1
8x64	8x32	8x16	8x8	8x4	8x2	8x1
4x64	4x32	4x16	4x8	4x4	4x2	4x1
2x64	2x32	2x16	2x8	2x4	2x2	2x1
1x64	1x32	1x16	1x8	1x4	1x2	1x1

Note that the mipmaps can be indexed by a pair of number. For instance, the mipmap 64x64 can be identified to the pair (1,1), the mipmap 64x16 could be identified to the point (3,1), mipma 2x4 would be (5,6) and mipmap 1x1 would be (7,7).

It might appear on first look that this will require a lot of memory, however this is not as bad as it might first appears. A geometric demonstration is given below. This figure contains the texture map plus all the mipmaps listed above. As can be seen, the memory taken by the mipmaps and the texture map is four times the memory taken by the texture map alone.

In this case, the mipmaps are indexed by two indices. If no scaling occurs in the x direction, but scaling is roughly 1/2 in the y direction, we might want to use "mipmap (1,2)" (which is the 64x32 mipmap).

All in all, uniform mipmapping takes 4/3 the memory used by a texmap but has its problems, while nonuniform mipmapping makes an attempt to reduce these problems but takes 4 times as much memory as a simple texmap. Note, however, that we are fortunate enough that the amount of memory taken by nonuniform mipmapping is merely a constant times what is required for texture mapping.



Summed area tables

If we want to use a block filter (average all the texels that should go in a mipxel), we can use something perhaps more general, called a summed area table.

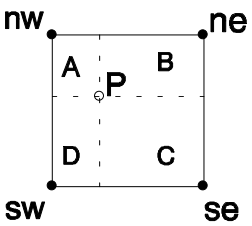
Let's say we have a texture map T[x][y]. We want to calculate the average of all pixels in the square delimited by say (p,q) and (r,s). (That is, the square [p,r)x[q,s).) One way is to pre-compute a summed area table of the same size as the original texture. This summed area table S is defined as follows. S[x][y] is the sum of all texels T[m][n] for m<x, n<y. Then it is easy to see that the sum of all texels in the (p,q)-(r,s) square is Q=S[r,s]-S[p,s]-S[r,q]+S[p,q]. Then the mean is M=Q/([r-p]x[s-q]).

This allows us to apply one very special type of filter to axis aligned boxes in the texture map relatively quickly. The probem is, most of the time, the texels that cover a pixel we are rendering do not form an axis aligned box in the texture map. There are also other issues: the block filter is not a tremendously attractive one, the filter should depend on the relative space taken up by each texel on screen, and so on.

Bi-linear interpolation

Now we have (more or less) taken care of the case where several texels are in the same pixel. But sometimes the opposite happens, and a texel gets stretched over several pixels. Of course, if we are using one of the smaller mipmaps and the texels cover many pixels it could mean that we should be using a larger mipmap. But when we get to the raw texture map and there are no more "larger" texture maps, we're stuck.

Bi-linear interpolation attempts to solve this problem. We will be using a linear polynomial of two variables (ie, a plane equation), thus the “Bi” of bi-linear. Typically, the texel coordinates will not be integer, as is depicted below.



The texels are nw, ne, sw and se (short for north west, north east and such). P is the actual texture coordinate for the current pixel. A, B, C and D are the area of the rectangles seen on the diagram above. The color we assign to the pixel will be:

A×nw+B×ne+C×se+D×sw

This is the "bi" part of bi-linear. Incidentally, this will also improve pictures when mipmapping is used.

It is possible to use something other than linear interpolation. For instance, bi-cubic interpolation is popular. Bicubic polynomials are often used. A grid of 4x4 texels will be used as control points for some uniform spline of two variables. Once the spline coefficients are known, the spline is evaluated at the intermediate point P (see the figure for bi-linear interpolation, above) and this value is used to shade the pixel.

Tri-linear interpolation

Tri-linear interpolation is bi-linear interpolation with an additional interpolation. First, a description for uniform mipmapping will be given, then this will be extended to nonuniform mipmapping and summed area tables.

When we use mipmapping, we need a function which tells us what mipmap to use. However, maybe this function tells us to use "mipmap #3.15", which we round to simply "mipmap #3". This means that we should use a mipmap somewhat in between 3 and 4, but more towards 3. Tri-linear interpolation simply interpolates linearly the mipmap between mipmap 3 and 4 (probably using, in our specific example of "mipmap #3.15", 85% of mipmap 3 and 15% of mipmap 4).

This will make the change in mipmaps quite smoother, which will have an important effect on animations in particular.

Using nonuniform mipmapping, we simply extend our idea to interpolate linearly between four mipmaps. Ie, if we want mipmap #3.2 in the x direction and #5.7 in the y direction, we mix in $(0.8\times0.3)=0.24$ of mipmap (3,5) with $(0.2\times0.3)=0.06$ of mipmap (4,5), $(0.8\times0.7)=0.56$ of mipmap (3,6) and lastly $(0.2\times0.7)=0.14$ of mipmap (4,7). As we can see, mipmap (3,6) is the dominant one, as is to be expected and mipmap (4,5) is barely used at all.

Glossary

Complex numbers: a number with a real and imaginary part, of the form Z=a+bi, where i is the imaginary part. We define i^2=-1. This way, we can define addition, multiplication, subtraction, and even inverse. Complex numbers can be compared to points in the plane. As such, they have a polar coordinate form. From this we can define the euclidian norm, or absolute value of a complex number. By using Euler's representation, we can put this form in exponential form. It is of note that multiplication by unit complex numbers represent rotations in the plane.

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. A stricter mathematical definition is used in the spline chapter. It is a generalization of our simpler definitions.

Concave: any non self-intersecting 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.

Euler angles: 3 angles used to represent a specific orientation. Can be used to represent any rotation, but is not very useful in practice for several reasons. First, the order in which the rotations are applied is important. Second, it is very hard to find the Euler angles for a given orientation. Third, the angles have very little physical meaning.

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, they are sometimes referred to as faces.

Mach banding: The human eye accentuates contrasts. Therefore, if two surfaces of slightly different colors lie next to each other, the boundary between the two will be clearly visible since it's an area of high contrast (contrast is, more or less, change of color over distance). This happens when we have too few colors to give a continuous texture to a surface, or when coloring adjacent faces of an object with uniform but slightly different shades of the same color.

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

Normalizing: making a vector V of unit length, by multiplying it by 1/|V|.

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 normally 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.

Quaternion: similar to a complex number, a quaternion has 1 real and 3 imaginary parts. It is generally written as Q=a+bi+cj+dk, where i, j and k are orthonormal imaginary parts. These imaginary components satisfy the following equalities: ij=-ij=k, jk=-jk=i, ki=-ik=j, $i^2=j^2=k^2=ijk=-1$. From that, addition, subtraction and multiplication operations can be defined. The useful thing about quaternions is, just multiplication by unit complex numbers represent rotations in 2d, multiplication by unit quaternions represent rotations in 3d.

Taylor series: A power series that approximates a function.

Texel: One unit in the texture map, the texture map equivalent of a pixel.

Texture mapping: Historically, this is the "sticking" of a "picture" on top of a polygon. A bitmap, called texture map, is mapped on a polygon or triangle. This has been generalized greatly however. Now, texture maps are sometimes replaced by a function of an x,y,z point in space. Also, the texture need no longer be a simple picture. The texture can be slight perturbations to the surface normal (bump mapping), a "transparency level" (alpha channel) or a number of other things.

Pixel: The smallest dot displayable by the hardware. Also used to describe one small square unit in a bitmap.

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.

Bibliography

[1] James Foley, Andries van Dam, Steven Feiner, John Hughes, *Computer Graphics Principles and Practice*, Addison-Wesley, 1990.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, *Introduction to Algorithms*, McGraw-Hill, 1989.

[3] S. D. Conte, Carl de Boor, *Elementary Numerical Analysis*, McGraw-Hill, 1980.

[4] Ming Chieh Lin, *Efficient Collision Detection for animation and robotics*, PhD Thesis at University of California at Berkeley, 1993.

[5] Barsky, B., *Computer Graphics and Geometric Modeling Using Beta-splines*, Springer-Verlag, New York, 1988.