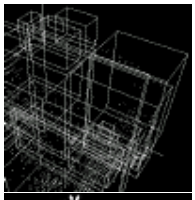
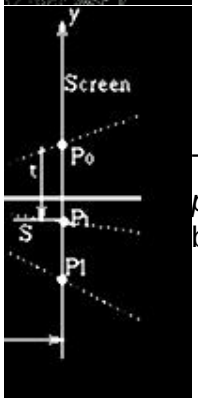


Few maths, lotsa demos

prev main next



This page is a gathering of some mathematical stuff I found useful while coding demos. No code, *a priori*. You might find that rather simplistic. I but agree: go read the Foley Van-Dam, or the Watt & Watt books instead...:)

Notations and definitions

- All *vectorial* quantities/fields will be typed in CAPITAL, whereas *scalar* in lower case.
- Vector components will be accessed subscripted, when not independantly defined. For instance: *Vo = (Vo_x, Vo_y, Vo_z)*, or *(xo, yo, zo)* when defined.
- Color vectors will be denoted with angles, their components being r,g,b: *[C];=(Cr,Cg,Cb)*
- *.* stands for the scalar product if concerning vectorial fields, multiplication otherwise.
- ^ is the vectorial product between two vector. Namely A^B is:

  (Ay.Bz - Az.By)
  (Az.Bx - Ax.Bz)
  (Ax.By - Ay.Bx)

  Used between scalar, it's the power raising operator. With matrices, *M^-1*is this matrix's inverse.
- We'll be here mostly dealing with polygons, and triangles in particular. For such a triangle defined by its 3 vertices *(Vo,V1,V2)*, we will defined *dF1* to be *dF1 = F1 - Fo*, for *any* field *F* defined over this triangle, and having values *(Fo,F1,F2)* at the vertices *(Vo,V1,V2)*.
  Similarly, *dF2 = F2-Fo* and *dF12 = F2-F1* (less used, however).
  For instance, the vector *dV1* is *V1-Vo*, an edge of the triangle, having coordinate *(dx1,dy1,dz1)*, i.e *(x1-xo, y1-yo, z1-zo)*.
- At a point *Po* of an object's surface, we'll define several vectors most models will use:

  * The normal ***N***, whose euclidian norm is *1*, pointing *OUTSIDE* the object. This vector is the one perpendicular to the plane best approximating the object in the neighbourhood of *Po*. For a surface given by *f(x,y,z)=0*, it is given by *(df/dx, df/dy, df/dz)*. For a triangle defined by its 3 vertices, this normal is given by the normalized vectorial product *dV1 ^ dV2*.

  * The view vector *V*, also normalized to *1*. It is given by *V = Vo - Po*, *Vo* being the observer's (camera) position.

  * The light vector *L*, normalized to *1*. It is given by *L = Lo - Po*, *Lo* being the light's position.

- With this three vectors, one defines 2 new useful ones:

  * The reflected vector *R*. It is the symetrical of *V* around *N*:

  R = 2 (N.V) N - V

  It is automatically normalized to *1*if *V*and *N*are. It is belonging to the same half-space (bounded by the tangent plane *Tp*orthogonal to *N*and passing throught *Po*) as *V*. Thanks to refraction laws :). Of course, we have *N.V = N.R*.

  * The transmitted vector *T*. It is given by the refraction law: *n1.sin(a1) = n2.sin(a2)*. *n1* and *n2* are the refraction indices (>=*1*) of the media *1* and *2*, *a1* and *a2* are the angle of

entering and transmitted light rays with respect to *N*.
Taking *V* as entering vector and belonging to medium *1*, we have:

$$T = [\, n(N.V) - \cos(a2) \,]\, N - n\, V$$

where: $n = n1/n2 = \sin(a2)/\sin(a1)$. You'll get $\cos(a2)$ from that...

It is automatically normalized to *1* if *V* and *N* are. It is belonging to the other half-space than *V*.

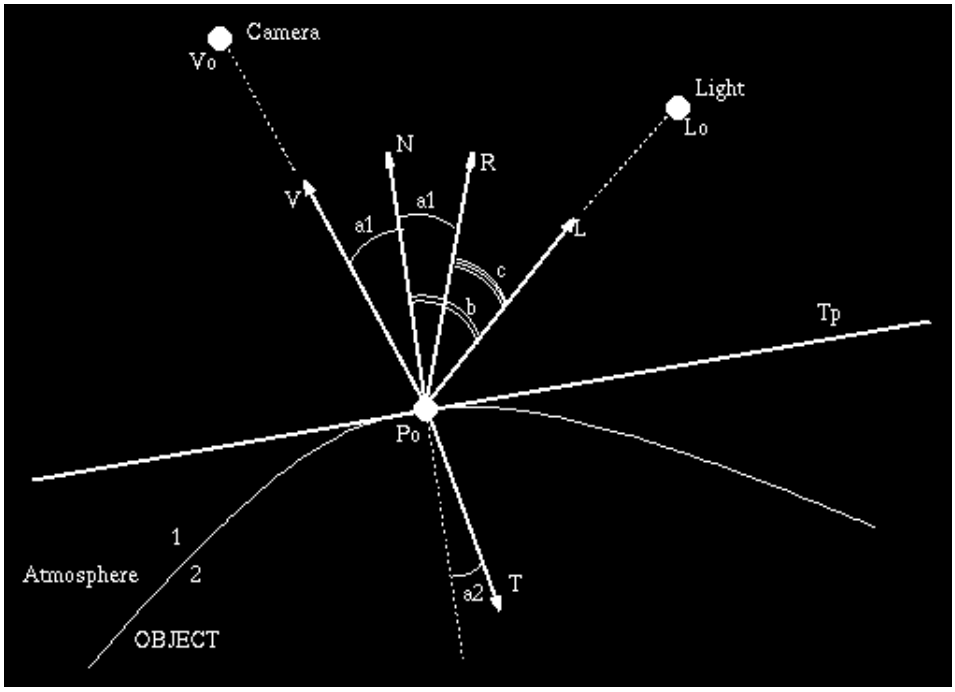


Fig1: all what we've just defined...

3D and Perpective transformations

- Scale-Rotation-Translation
  Having an object's point *Po* (a poly vertex, e.g.) defined in the basic frame, one wants to transform it in the world's referential. Namely: apply a scale transformation, then a rotation, and eventually a translation to this object. Well, just compose the corresponding matrices and apply the resulting one to the original vector !
  What are this matrices ? One can use 4x4 matrices, using generalized -four dimensional- coordinates, but it's mostly useful when one want to consider perspective transformation (see below) as a matrix like others. That we won't, since perspective transform is more efficiently coded (methink :) when considered independantly from 3D transformations (making demo is *not* writting a general renderer - yet - :).
  So, rotation of an angle ax around the X-axis, followed by an rotation of ay around the Y-axis, plus a rotation of az around the Z-axis is given by the application of the following matrix:

  $Rz(az) \circ Ry(ay) \circ Rx(ax) =$

$$\begin{matrix} cz.cy, & cz.sy.sx - sz.cx, & cz.sy.cx + sz.sx \\ sz.cy, & sz.sy.sx + cz.cx, & sz.sy.cx - cz.sx \\ -sy, & cy.sx, & cy.cx \end{matrix}$$

  where

cx = cos(ax), sx = sin(ax)
cy = cos(ay), sy = sin(ay)
cz = cos(az), sz = sin(az)

- Some useful facts about matrix:
  * Applying a matrix to a vector is done by multiplying *on left*: P = M.Po
  * Taking the transpose of a matrix product reverses the order: *t(A.B.C) = tC.tB.tA*
  Same thing for inversion: *(A.B)^-1 = B^-1 . A^-1*...
  * Rotations in 3D do *not* commute. Rx o Ry o Rz != Rx o Rz o Ry. It would be too easy :)
  * Inversion of a rotation matrix is easy, thanks to isometry property: just take the transpose !
  * Always the same ol' dilemna with contra/co-variant tensors:
  Normals are *axial* vectors (they can be written like a vectorial product). Hence if you're transforming point *Po* with the matrix *M*, the normal attached to this point must be transformed with *t(M^-1)*. Lucky you, if *M* is only a rotation matrix, *t(M^-1) = M*. But be careful when scaling are concerned ! Moreover, transforming a normal does not deal with the translational components of the object !

  A bit of explanation about where this comes from: take any vector *Vo* orthogonal to a normal *No*. Transformed by *M*, the vector will be sent to *V = M.Vo*. If one wants the tangential property *Vo.No = 0* to still hold after the transformation (i.e. that *N.V=0*), what should be the matrix being applied to *No* ? Yep, *t(M^-1)*.

Digression about quaternions

Quaternions are useful to represent and interpolate rotations in 3D space.

First, what is a quaternion ?
It's a generalization of complex numbers in 4 dimensions. The quaternion space has 4 vectors as basis: 1, i, j and k. Whereas the additive rule is 'classical', the multiplication follows the rules: *ij = -ji = k*, plus the corresponding relation after cyclic permutation of (i,j,k). Hence, multiplication is *not* commutative with quaternion.

You can represent a quaternion with a scalar s and a tri-dimentional vector **V** with: *q=s+i.Vx+j.Vy+k.Vz*, hence notated *q = (s,**V**)*.

Now, you can check that a multiplication of 2 quaternions *q1 = (s1,**V1**).* and *q2 = (s2,**V2**).* is: *q1.q2 = (s1.s2 - V1.V2, s1 V2 + s2 V1 + V1 x V2)*. This notation has the advantage of using the 3d operation 'dot product' and 'cross product' (and making appear clearly that the non-commutative-ness is due to the presence of this cross product).

The conjugate of a *q=(s,V)* is *!q=(s,-V)* and its norm is *|q|^2 = q.!q*.

Now, the main trick:
Take any 3d-vector **V**, make it quaternionic: *v=(0,**V**)*. Now take the quaternion *r=(cos t/2, sin t/2.N)*, whose norm is *1*. *t* is an angle, and *N* a unit 3d axis.

Now, compute *v'=r.v.r^1*. You'll realize (the expression is heavy... I won't write it here :) that *v'* has the form: *v'=(0,V')* where *V'* is **the vector deduce from R by a rotation of axis *N* and of angle *t*** !!

Hence, you can *also* represent a rotation with a quaternion (*r*) which accounts for 4 parameters (3 if you recall that *r* has norm equal to 1), instead of usual representation. What's the advantage ? Since this quaternion are embedded in a four-dimensional space, it allows much more freedom while interpolating the corresponding rotation.

The correspondance is done as follow: a unit quaternion *r=(W,X,Y,Z)* correspond to the following rotation matrix:

1-2Y^2-2Z^2,	***2XY-2WZ,***	***2XZ+2WY***
2XY+2WZ,	***1-2X^2-2Z2,***	***2YZ-2WX***
2XZ-2WY,	***2YZ+2WX,***	***1-2X^2-2Y^2***

So you can convert easily between quaternion,rotation, and rotation matrix... Here are some additional infos about quaterions.

So, if you want an object's rotation to be interpolated from, say, *R1* to *R2*, just transform these rotation into corresponding quaternions *r1* and *r2*, interpolate in quaternion space between these two, and transform back this interpolated quaternion into a rotation matrix: you'll then have the intermediate object's positionning.
Beware, you just can't simply interpolate linearly !! The norm of the interpolated quaternion between *r1* and *r2* *must* be 1 if you want it to represent a intermediate rotation. This is tricky.

Space transforms

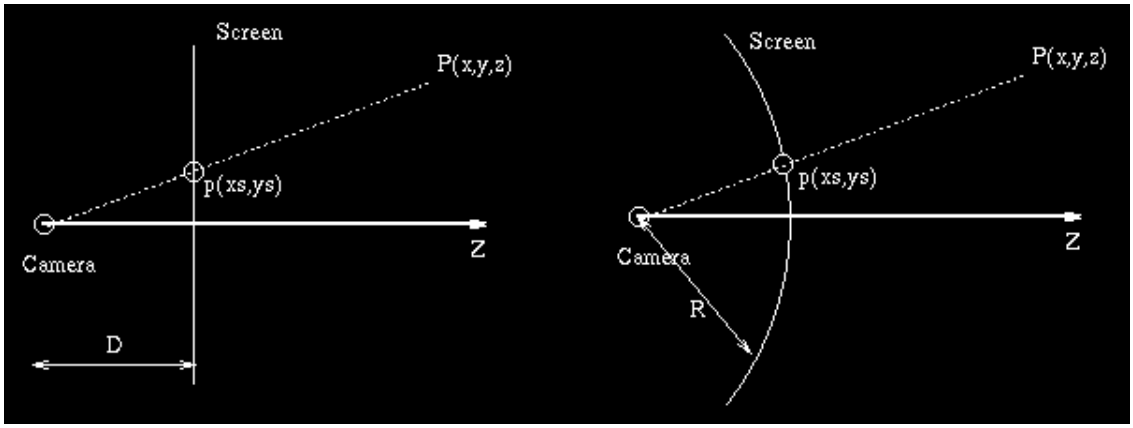


Fig2: Planar/spherical perspective

- Planar perspective transform:
  Perspective transformation is the fact of projecting a 3-dimensional points *P* on a 2-D dimensional surface -the screen-. You draw the line between *P* it and the camera's position (assuming this camera is centered at the origin, looking along the Z-axis), and find out where this line intersects a plane perpendicular to the Z-axis, at a distance *z = D* from the camera. It's:
  (x, y, z) => (xs = D.x/z, ys = D.y/z)
  ...and you scale that to fit your (real) screen size...
- Spherical perspective transform: If you want to project points on a spherical screen (fish-eye effect) of radius *R*, just take:
  (x, y, z) => (xs = R/pi.atan(x/z), ys = R/pi.atan(y/z))

  (for numerical reason, I but recommend making two different cases when x/z>1 and x/z<1. And use atan2() instead of atan(), just to be safe with signs)
- Cylindrical perspective transform: Just mix the two above :)

Lighting models

So, you've got an object's point *Po*, illuminated with a light located at *Lo*, and viewed from position *Vo* (see Fig1). What should be this point's color ? There are many possible models, here are some examples:

- Ambient light: just take a constant color [Co], supposed to be the object's one.
  It models the ambient light of the environment the object is put in. Radiosity tries to model this more accuratly, but shouldn't be considerated further here (for CPU's sake :)

- Diffuse lighting: A basic diffuse color [Cd], but scaled (i.e. each R,G,B channels. That's what's great with R,G,B model) accordingly to the diffuse illumation coefficient *cos(b) = (N.L)*.
  One can also use raised to certain power *nd*:

  [C] = [Cd].(N.L)^(nd)

- Phong model: this takes in account a certain reflective property of the material, hence reflecting all the more light than it's in the reflected direction.

  [C] = [Cp].(R.L)^(n)

  Here, *n* accounts approximatively to the light's size throught the highlight this model produces on the object's surface. Take *n ~= 30*, and you'll have a flat hightlight. Take it *~= 200*, and you'll have a tiny one...

- Other models: some more physically-based models are available (Blinn, Cook-Torrance,...), but of little use when demos are at stake, since accuracy is less the matter than natural-looking/speed considerations. With the 3 above, plus some reflection/env/texture mapping, most effects will be great. Just adapt the [Co], [Cd], [Cp] coeffs...

Miscellaneous poly mapping

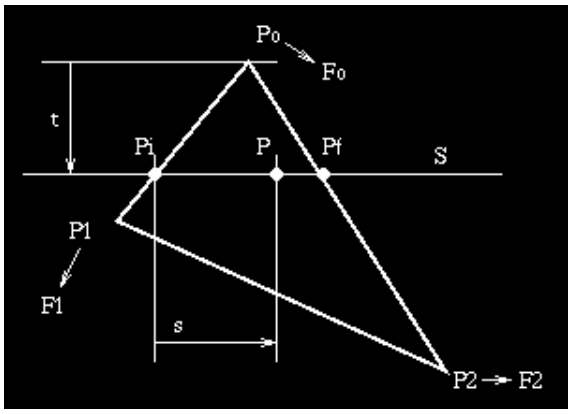


Fig3: The poly we want to scan convert

Consider a polygon defined by its vertices *(Vo,V1,V2)*, projected on the screen to points *(Po,P1,P2)* (*Vi* are three-dimensional points, *Pi* two-dimensional ones...). We also have a field *F* defined at the vertices throught its values *(Fo,F1,F2)* and we want to interpolate it over the whole polygon, under a scan-line basis. *F* can be the color, the diffuse component, the U coordinate in a texture, anything...

We have sorted (3 lines of code) the vertices with respect to their *y* coordinates. We now have to process the triangle between its vertices *Po* and *P1*, and then the part between vertices *P1* and *P2*, which is done almost the same way the first is...

- Basic interpolation:
  It consists of linearly interpolating *F* with respect to the *SCREEN* coordinates *(Po,P1,P2)* of the projected vertices. Perspective correction will consist in the same, but applied to the *SPACE*

coordinates *(Vo,V1,V2)*.

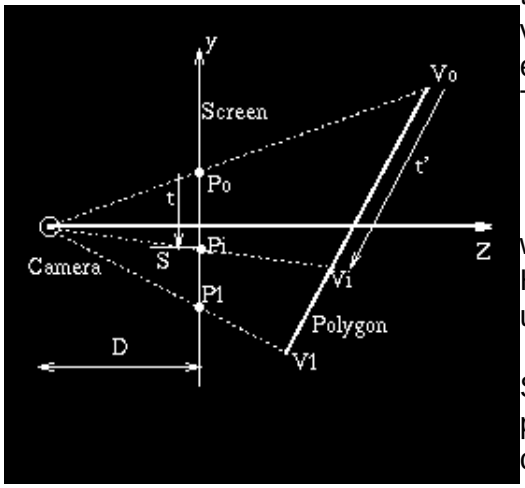
Our scan-line *S* will go from *Po* to *P1* as the variable *t* goes from *0.0* to *1.0*. Actually, *t = (y-yo)/(y1-yo)*. It intersects the edges of the triangle on the initial point *Pi* and the final one *Pf* (we're scanning from left to right. That's the opposite of what a code should do, but we're not coding :). The field values at *Pi* and *Pf* are:

$$Fi = Fo + t*dF1$$
$$Ff = Fo + t*dF2$$

Now, we're going from *Pi* to *Pf*, interpolating along the X-axis with a variable *s = (x-xi)/(xf-xi)* also going from *0.0* to *1.0*. The desired field value is just:

$$F = Fi + s*(Ff-Fi)$$

- Perspective correction (planar projection only):



The problem with the previous scheme comes from *Pi*, the interpolated point between the *screen* points *Po* and *P1*. If we re-project *Pi* on the *real* poly, we obtain *Vi*. But... the equivalent of the ratio *t = (y-yo)/(y1-yo)* for the space vertices is *t' = (Vi_y-Vo_y) / (V1_y-Vo_y)*, which is not egal to *t*.
The real relation between *t* and *t'* is:

$$t' = t / (1 + (1-t).e)$$

where *e = (V1_z - Vo_z)/Vo_z*.
Hopefully, *e* is generally small. It is assumed null for the usual, linear, mapping.

So, are we left with an awful additional div/pixel to do perspective correction ? Not really: since *e* is small, you can approximate the div with:

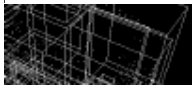
$$1 / (1 + (1-t).e) \sim= 1 - (1-t).e$$

which can be incrementally coded...

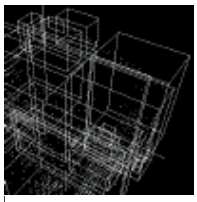
Useful resources

Here some rather old but still useful resources I've substracted from Internet oblivion:

- The quaternion/matrix FAQ
- fatmap.txt Made by Mats Byggmastar, it's about coding UV mapper efficiently...
- The comp.graphics.algorithms FAQ. Always a good basis.
- OPTIMIZE.TXT A text (originally by Michael Kunstelj) about what's going on in a CPU, Penti*m especially.
- DJASM.TXT What's the AT&T syntax for ASM, by avly@remus.rutgers.edu.



skal@planet-d.net



Matrix and Quaternion FAQ

prev main next

The Matrix and Quaternions FAQ
==========

Version 1.2 2nd September 1997

This FAQ is maintained by "hexapod@netcom.com". Any additional suggestions or related questions are welcome. Just send E-mail to the above address.

Feel free to distribute or copy this FAQ as you please.

Contributions

Introduction I1: steve@mred.bgm.link.com

Introduction
==========

I1. Important note relating to OpenGL and this document

Questions

BASICS
==

ARITHMETIC
==

DETERMINANTS AND INVERSES
==

TRANSFORMS
==========

QUATERIONS
==========

Introduction
============

I1. Important note relating to OpenGl and this document

In this document (as in most math textbooks), all matrices are drawn in the standard mathematical manner. Unfortunately graphics libraries like IrisGL, OpenGL and SGI's Performer all represent them with the rows and columns swapped.

Hence, in this document you will see (for example) a 4x4 Translation matrix represented as follows:

```
        | 1  0  0  X |
        |            |
        | 0  1  0  Y |
    M = |            |
        | 0  0  1  Z |
        |            |
        | 0  0  0  1 |
```

In Performer (for example) this would be populated as follows:

```
M[0][1] = M[0][2] = M[0][3] =
M[1][0] = M[1][2] = M[1][3] =
M[2][0] = M[2][1] = M[2][3] = 0 ;
M[0][0] = M[1][1] = M[2][2] = m[3][3] = 1 ;
M[3][0] = X ;
M[3][1] = Y ;
M[3][2] = Z ;
```

ie, the matrix is stored like this:

```
        | M[0][0]  M[1][0]  M[2][0]  M[3][0] |
        |                                    |
        | M[0][1]  M[1][1]  M[2][1]  M[3][1] |
    M = |                                    |
        | M[0][2]  M[1][2]  M[2][2]  M[3][2] |
        |                                    |
        | M[0][3]  M[1][3]  M[2][3]  M[3][3] |
```

OpenGL uses a one-dimensional array to store matrices - but fortunately, the packing order results in the same layout of bytes in memory - so taking the address of a pfMatrix and casting it to a float* will allow you to pass it directly into routines like glLoadMatrixf.

In the code snippets scattered throughout this document, a one-dimensional array is used to store a matrix. The ordering of the array elements is transposed with respect to OpenGL.

```
       This Document                  OpenGL

      | 0  1  2  3  |            | 0  4  8  12 |
      |             |            |             |
      | 4  5  6  7  |            | 1  5  9  13 |
  M = |             |        M = |             |
      | 8  9  10 11 |            | 2  6  10 14 |
      |             |            |             |
      | 12 13 14 15 |            | 3  7  11 15 |
```

Basics

======

Q1. What is a matrix?

A matrix is a two dimensional array of numeric data, where each row or column consists of one or more numeric values.

Arithmetic operations which can be performed with matrices include addition, subtraction, multiplication and division.

The size of a matrix is defined in terms of the number of rows and columns.

A matrix with M rows and N columns is defined as a MxN matrix.

Individual elements of the matrix are referenced using two index values. Using mathematical notation these are usually assigned the variables 'i' and 'j'. The order is row first, column second

For example, if a matrix M with order 4x4 exists, then the elements of the matrix are indexed by the following row:column pairs:

```
      | 00 10 20 30 |
  M = | 01 11 21 31 |
      | 02 12 22 32 |
      | 03 13 23 33 |
```

The element at the top right of the matrix has i=0 and j=3
This is referenced as follows:

```
  M    = M
   i,j    0,3
```

In computer animation, the most commonly used matrices have either 2, 3 or 4 rows and columns. These are referred to as 2x2, 3x3 and 4x4 matrices respectively.

2x2 matrices are used to perform rotations, shears and other types of image processing. General purpose NxN matrices can be used to perform image processing functions such as convolution.

3x3 matrices are used to perform low-budget 3D animation. Operations such as rotation and multiplication can be performed using matrix operations, but perspective depth projection is performed using standard optimised into pure divide operations.

4x4 matrices are used to perform high-end 3D animation. Operations such as multiplication and perspective depth projection can be performed using matrix mathematics.

Q2. What is the "order" of a matrix?

The "order" of a matrix is another name for the size of the matrix. A matrix with M rows and N columns is said to have order MxN.

Q3. How do I represent a matrix using the C/C++ programming languages?

The simplest way of defining a matrix using the C/C++ programming languages is to make use of the "typedef" keyword. Both 3x3 and 4x4 matrices may be defined in this way ie:

```
typedef float MATRIX3[9];
typedef float MATRIX4[16];
```

Since each type of matrix has dimensions 3x3 and 4x4, this requires 9 and 16 data elements respectively.

At first glance, the use of a single linear array of data values may seem counter-intuitive. The use of two dimensional arrays may seem more convenient ie.

```
typedef float MATRIX3[3][3];
typedef float MATRIX4[4][4];
```

However, the use of two reference systems for each matrix element very often leads to confusion. With mathemetics, the order is row first (i), column second (j) ie.

```
Mij
```

Using C/C++, this becomes

```
matrix[j][i]
```

Using two dimensional arrays also incurs a CPU performance penalty in that C compilers will often make use of multiplication operations to resolve array index operations.

So, it is more efficient to stick with linear arrays. However, one issue still remains to be resolved. How is an two dimensional matrix mapped onto a linear array? Since there are only two methods (row first/column second or column first/row column).

The performance differences between the two are subtle. If all for-next loops are unravelled, then there is very little difference in the performance for operations such as matrix-matrix multiplication.

Using the C/C++ programming languages the linear ordering of each matrix is as follows:

```
mat[0]  = M        mat[3]  = M
           00                 03


mat[12] = M        mat[15] = M
           30                 33
```

```
      |  0  1  2  3 |                 |             | 0 1 2 |
      |              |                 |             |       |
      |  4  5  6  7 |                 |             |       |
  M = |              |             M = | 3  4  5 |
      |  8  9 10 11 |                 |             |       |
```

```
        |            |             | 6 7 8 |
        | 12 13 14 15 |
```

Q4. What are the advantages of using matrices?

One of the first questions asked about the use of matrices in computer animation is why they should be used at all in the first place. Intuitively, it would appear that the overhead of for-next loops and matrix multiplication would slow down an application.

Arguments that resolve these objections can be pointed out. These include the use of CPU registers to handle loop counters on-board data caches to optimise memory accesses.

Advantages can also be pointed out. By following a mathematical approach to defining 3D algorithms, it is possible to predict and plan the design of a 3D animation system. Such mathematical approaches allow for the implementation of character animation, spline curves and inverse kinematics.

However, one objection that frequently comes up is that it would be quicker to just multiply each pair of coordinates by the rotation coefficients for that axis, rather than perform a full vector-matrix multiplication.

ie. Rotation in X transforms Y and Z
Rotation in Y transforms X and Z
Rotation in Z transforms X and Y

The argument to this goes as follows:

Given a vertex V = (x,y,z), rotation angles (A,B and C) and translation (D,E,F). A the algorithm is defined as follows:

```
---------------------------

sx = sin(A)             // Setup - only done once
cx = cos(A)
sy = sin(B)
cy = cos(B)
sz = sin(C)
cz = cos(C)

x1 =  x * cz +  y * sz  // Rotation of each vertex
y1 =  y * cz -  x * sz
z1 =  z

x2 = x1 * cy + z1 * sy
y2 = z1
z2 = z1 * cy - x1 * sy

x3 = x2
y3 = y2 * cx + z1 * sx
z3 = z2 * cx - x1 * sx

xr = x3 + D             // Translation of each vertex
```

```
  yr = y3 + E
  zr = z3 + F

  ---------------------------
```

Altogether, this algorithm will use the following amounts of processing time:

Set-up	Per-vertex
6 trigonometric functions	
6 assignment operations.	12 assignment
	12 multiplication
	9 addition

Assume that the same operations is being performed using matrix multiplication.

With a 4x4 matrix, the procesing time is used as follows:

Set-up	Change	Per-vertex	Change
6 trigonometric functions	0		0
18 assignment operation	-12	3 assignment	-9
12 multiplication	+12	9 multiplication	-3
6 subtraction	+6	6 addition	-3

Comparing the two tables, it can be seen that setting up a rotation matrix costs at least 12 multiplication calculations and an extra 18 assignment calls.

However, while this may seem extravagant, the savings come from processing each vertex. Using matrix multiplication, the savings made from processing just 4 vertices, will outweigh the additional set-up cost.

Q5. How do matrices relate to coordinate systems?

With either 3x3 or 4x4 rotation, translation or shearing matrices, there is a simple relationship between each matrix and the resulting coordinate system.

The first three columns of the matrix define the direction vector of the X, Y and Z axii respectively.

If a 4x4 matrix is defined as:

```
      | A B C D |
  M = | E F G H |
      | I J K L |
      | M N O P |
```

Then the direction vector for each axis is as follows:

```
   X-axis = [ A E I ]
```

Y-axis = [B F J]

Z-axis = [C G K]

Arithmetic

Q6. What is the identity matrix?

The identity matrix is matrix in which has an identical number of rows and columns. Also, all the elements in which i=j are set one. All others are set to zero. For example a 4x4 identity matrix is as follows:

$$M = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Q7. What is the major diagonal of a matrix?

The major diagonal of a matrix is the set of elements where the row number is equal to the column number ie.

M_{ij} where i=j

In the case of the identity matrix, only the elements on the major diagonal are set to 1, while all others are set to 0.

Q8. What is the transpose of a matrix?

The transpose of matrix is the matrix generated when every element in the matrix is swapped with the opposite relative to the major diagonal

This can be expressed as the mathematical operation:

$$M'_{ij} = M_{ji}$$

However, this can only be performed if a matrix has an equal number of rows and columns.

If the matrix M is defined as:

$$M = \begin{vmatrix} 0.707 & -0.866 \\ 0.866 & 0.707 \end{vmatrix}$$

Then the transpose is equal to:

$$T = \begin{vmatrix} 0.707 & 0.866 \\ -0.866 & 0.707 \end{vmatrix}$$

If the matrix is a rotation matrix, then the transpose is guaranteed to be the inverse of the matrix.

Q9. How do I add two matrices together?

The rule of thumb with adding two matrices together is:

"add row and column to row and column"

This can be expressed mathematically as:

$$R_{ij} = M_{ij} + L_{ij}$$

However, both matrices must be identical in size.

For example, if the 2x2 matrix M is added with the 2x2 matrix L then the result is as follow:

$$R = M + L$$

$$\begin{vmatrix} A & B & C \\ D & E & F \\ G & H & I \end{vmatrix} = \begin{vmatrix} A & B & C \\ D & E & F \\ G & H & I \end{vmatrix} + \begin{vmatrix} J & K & L \\ M & N & O \\ P & Q & R \end{vmatrix}$$

$$= \begin{vmatrix} A+J & B+K & C+L \\ D+M & E+N & F+O \\ G+P & H+Q & I+R \end{vmatrix}$$

Q10. How do I subtract two matrices?

The rule of thumb with subtracting two matrices is:

"subtract row and column from row and column"

This can be expressed mathematically as:

$$R_{ij} = M_{ij} - L_{ij}$$

However, both matrices must be identical in size.

For example, if the 2x2 matrix L is subtracted from the 2x2 matrix M then the result is as follows:

$$R = M - L$$

$$\begin{vmatrix} A & B & C \\ D & E & F \\ G & H & I \end{vmatrix} = \begin{vmatrix} A & B & C \\ D & E & F \\ G & H & I \end{vmatrix} - \begin{vmatrix} J & K & L \\ M & N & O \\ P & Q & R \end{vmatrix}$$

```
        |       |   |       |
        | G H I |   | P Q R |


        | A-J B-K C-L |
        |             |
      = | D-M E-N F-O |
        |             |
        | G-P H-Q I-R |
```

Q11. How do I multiply two matrices together?

The rule of thumb with multiplying two matrices together is:

"multiply row into column and sum the result".

This can be expressed mathematically as:

$$R_{ij} = \sum_{i=1}^{n} M_{ij} \times L_{ji}$$

If the two matrices to be multiplied together have orders:

M = AxB and L = CxD

then the two values B and C must be identical.

Also, the resulting matrix has an order of AxD

Thus, it is possible to multiply a 4xN matrix with a 4x4 matrix but not the other way around.

For example, if the 4x4 matrix M is defined as:

```
      | A B C D |
  M = | E F G H |
      | I J K L |
      | M N O P |
```

and a 4x2 matrix L is defined as:

```
  L = | Q R |
      | S T |
      | U V |
      | W X |
```

then the size of the resulting matrix is 2x4. The resulting matrix is defined as:

R = M x L

```
      | A B C D |   | Q R |
    = | E F G H | x | S T |
      | I J K L |   | U V |
      | M N O P |   | W X |
```

```
        | AQ+BS+CU+DW  AR+BT+CV+DX |
      = | EQ+FS+GU+HW  ER+FT+GV+HX |
        | IQ+JS+KU+LW  IR+JT+KV+LX |
        | MQ+NS+OU+PW  MR+NT+OV+PX |
```

Q12. How do I square or raise a matrix to a power?

A matrix may be squared or even raised to an integer power. However there are several restrictions. For all powers, the matrix must be orthogonal ie. have the same width and height

For example,

M^{-1} is the inverse of the matrix

M^{0} generates the identity matrix

M^{1} leaves the matrix undamaged.

M^{2} squares the matrix and

M^{3} generates the cube of the matrix

Raising a matrix to a power greater than one involves multiplying a matrix by itself a specific number of times.

For example,

$M^{2} = M \cdot M$

$M^{3} = M \cdot M \cdot M$

and so on.

Raising the identity matrix to any power always generates the identity matrix ie.

$I^{n} = I$

Q13. How do I multiply one or more vectors by a matrix?

The best way to perform this task is to treat the list of vectors as a single matrix, with each vector represented as a column vector.

If N vectors are to be multiplied by a 4x4 matrix, then they can be

treated as a single 4xN matrix:

If the matrix is defined as:

```
      | A B C D |
  M = | E F G H |
      | I J K L |
      | M N O P |
```

and the list of vectors is defined as:

```
      | x1 x2 x3 x4 x5|
  V = | y1 y2 y3 y4 y5|
      | z1 z2 z3 z4 z5|
      | 1  1  1  1    |
```

Note that an additional row of constant terms is added to the vector list, all of which are set to 1.0. In real life, this row does not exist. It is simply used to make the orders of the matrix M and the vector list V match.

Then the multiplication is performed as follows:

```
          M . V = V'

| A B C D |   | x1 x2 x3 x4 x5 |   | A.x1+B.y1+C.z1+D A.x2+B.y2+C.z2+D
... |
| E F G H | . | y1 y2 y3 y4 y5 | = | E.x1+F.y1+G.z1+H E.x2+F.y2+G.z2+H
... |
| I J K L |   | z1 z2 y3 y4 z5 |   | I.x1+J.y1+K.z1+L I.x2+J.y2+K.z2+L
... |
| M N O P |   | 1  1  1  1  1  |   | M.x1+N.y1+O.z1+P M.x2+N.y2+O.z2+P
... |
```

For each vector in the list there will be a total of 12 multiplication 16 addition and 1 division operation (for perspective).

If the matrix is known not to be a rotation or translation matrix then the division operation can be skipped.

Determinants and Inverses

Q14. What is the determinant of a matrix?

The determinant of a matrix is a floating point value which is used to indicate whether the matrix has an inverse or not. If negative, then no inverse exists. If the determinant is positive, then an inverse exists.

For an identity matrix, the determinant is always equal to one. Any matrix with a determinant of 1.0 is said to be isotropic.

Thus all rotation matrices are said to be isotropic, since the determinant is always equal to 1.0.

This can be proved as follows:

```
    | A B |   | cos X  -sin X |
M = |     | = |               |
    | C D |   | sin X   cos X |

D = AD - BC

D = (cos X . cos X) -  (-sin X . sin X)

        2          2
D = (cos X ) + (sin X)

        2       2
But, cos X + sin X = 1

Therefore,

D = 1
```

Q15. How do I calculate the determinant of a matrix?

The determinant of a matrix is calculated using Kramer's rule, where the value can be calculated by breaking the matrix into smaller matrices.

For a 2x2 matrix M, the determinant D is calculated as follows:

```
    | A B |
M = |     |
    | C D |

D = AD - BC
```

For 3x3 and 4x4 matrices, this is more complicated, but can be solved by methods such as Kramer's Rule.

Q16. What are Isotropic and Anisotropic matrices?

An Isotropic matrix is one in which the sum of the squares of all three rows or columns add up to one.

A matrix in which this is not the case, is said to be Anisotropic.

When 3x3 or 4x4 matrices are used to rotate and scale an object, it is sometimes necessary to enlarge or shrink one axis more than the others.

For example, with seismic surveys, it is convenient to enlarge the Z-axis by a factor or 50 or more, while letting the X and Y axii remain the same.

Another example is the implementation of "squash" and "stretch" with character animation. When a character is hit by a heavy object eg. an anvil, the desired effect is to character stretched out sideways and squashed vertically:

A suitable matrix would be as follows:

```
        |  2   0   0    0  |
    M = |  0   2   0    0  |
        |  0   0   0.5  0  |
        |  0   0   0    1  |
```

However, there is problem looming ahead. While this matrix will cause no problems with the transformation of vertex data, it will cause problems with gouraud shading using outward normals.

Because the transformation stage is implemented using matrix multiplication, both vertex data and outward normal data will be multiplied with this matrix.

While this is not a problem with vertex data (it is the desired effect) it causes a major headache with the outward normal data.

After raw multiplication, each outward normal will no longer be normalised and consequently will affect other calculations such as shading and back-face culling.

Q17. What is the inverse of a matrix?

Given a matrix M, then the inverse of that matrix, denoted as M^{-1}, is the matrix which satisfies the following expression:

$$M \cdot M^{-1} = I$$

where I is the identity matrix.

Thus, multiplying a matrix with its inverse will generate the identity matrix. However, several requirements must be satisfied before the inverse of a matrix can be calculated.

These include that the width and height of the matrix are identical and that the determinant of the matrix is non-zero.

Calculating the inverse of a matrix is a task often performed in order to implement inverse kinematics using spline curves.

Q18. How do I calculate the inverse of an arbitary matrix?

Depending upon the size of the matrix, the calculation of the inverse can be trivial or extremely complicated.

For example, the inverse of a 1x1 matrix is simply the reciprical of the single element:

ie. $M = | x |$

Then the inverse is defined as:

$$M^{-1} = \left| \frac{1}{x} \right|$$

Solving 2x2 matrices and larger can be achieved by using Kramer's Rule or by solving as a set of simultaneous equations.

However, in certain cases, such as identity or rotation matrices, the inverse is already known or can be determined from taking the transpose of the matrix.

Q19. How do I calculate the inverse of an identity matrix?

Don't even bother. The inverse of an identity matrix is the identity matrix. ie.

$$I \cdot I^{-1} = I$$

Any identity matrix will always have a determinant of +1.

Q20. How do I calculate the inverse of a rotation matrix?

Since a rotation matrix always generates a determinant of +1, calculating the inverse is equivalent of calculating the transpose.

Alternatively, if the rotation angle is known, then the rotation angle can be negated and used to calculate a new rotation matrix.

Q21. How do I calculate the inverse of a matrix using Kramer's rule?

Given a 3x3 matrix M:

$$M = \begin{vmatrix} A & B & C \\ D & E & F \\ G & H & I \end{vmatrix}$$

Then the determinant is calculated as follows:

$$\det M = \sum_{i=1}^{n} M_{0,i} * \text{submat}_{0,i}(M) * -1^{i}$$

where

$\text{submat}_{ij}(M)$ defines the matrix composed of all rows and columns of M excluding row i and column j. submat_{ij} may be called recursively.

If the determinant is non-zero then the inverse of the matrix exists. In this case, the value of each matrix element is defined by:

```
 -1      1                              i+j
M    = -----  *  det submat     M * -1
 j,i   det M               i,j
```

Q22. How do I calculate the inverse of a 2x2 matrix?

For a 2x2 matrix, the calculation is slightly harder. If the matrix is defined as follows:

```
    | A B |
M = |     |
    | C D |
```

Then the determinant is defined as:

```
det = AD - BC
```

And the inverse is defined as:

```
 -1     1   |  D  -B  |
M   =  ---  |         |
       det  | -C   A  |
```

This can be proved using Kramer's rule. Given the matrix M:

```
    | A B |
M = |     |
    | C D |
```

Then the determinant is:

```
                        0                              1
det =  M    * submat M    * -1 +  M    * submat M    * -1
        0,0           0,0           0,1           0,1

<=>    M    * M    * 1         +  M    * M    * -1
        0,0    1,1                0,1    1,0

<=>    A  * D                  +  B    * C    * -1

<=>    AD                      +  BC . -1

<=>    AD - BC

==============
```

And the inverse is derived from:

```
 -1                      0+0      -1
M    = det submat    * -1    <=> M    = M    *  1 <=> D
 0,0             0,0             0,0    1,1

 -1                      1+0      -1
M    = det submat    * -1    <=> M    = M    * -1 <=> C * -1
 0,1             1,0             0,1    1,0

 -1                      0+1      -1
M    = det submat    * -1    <=> M    = M    * -1 <=> B * -1
```

```
   1,0            0,1             1,0    0,1

   -1                      1+1     -1
  M    = det submat    * -1    <=> M    = M    *  1 <=> A
   1,1            1,1             1,1    0,0
```

Then the inverse matrix is equal to:

```
   -1    1  |  D  -C |
  M   = --- |        |
        det | -B   A |
```

Providing that the determinant is not zero.

Q23. How do I calculate the inverse of a 3x3 matrix?

For 3x3 matrices and larger, the inverse can be calculated by either applying Kramer's rule or by solving as a set of linear equations.

If Kramer's rule is applied to a matrix M:

```
      | A B C |
  M = | D E F |
      | G H I |
```

then the determinant is calculated as follows:

```
det M = A * (EI - HF) - B * (DI - GF) + C * (DH - GE)
```

Providing that the determinant is non-zero, then the inverse is calculated as:

```
   -1     1     |   EI-FH  -(BI-HC)   BF-EC  |
  M   = ----- . | -(DI-FG)   AI-GC  -(AF-DC) |
        det M   |   DH-GE  -(AH-GB)   AE-BD  |
```

This can be implemented using a pair of 'C' functions:

```
  ---------------------------------

  VFLOAT m3_det( MATRIX3 mat )
    {
    VFLOAT det;

    det = mat[0] * ( mat[4]*mat[8] - mat[7]*mat[5] )
        - mat[1] * ( mat[3]*mat[8] - mat[6]*mat[5] )
        + mat[2] * ( mat[3]*mat[7] - mat[6]*mat[4] );

    return( det );
    }

 ---------------------------------

 void m3_inverse( MATRIX3 mr, MATRIX3 ma )
   {
```

```
VFLOAT det = m3_det( ma );

if ( fabs( det ) < 0.0005 )
  {
  m3_identity( ma );
  return;
  }

mr[0] =    ma[4]*ma[8] - ma[5]*ma[7]   / det;
mr[1] = -( ma[1]*ma[8] - ma[7]*ma[2] ) / det;
mr[2] =    ma[1]*ma[5] - ma[4]*ma[2]   / det;

mr[3] = -( ma[3]*ma[8] - ma[5]*ma[6] ) / det;
mr[4] =    ma[0]*ma[8] - ma[6]*ma[2]   / det;
mr[5] = -( ma[0]*ma[5] - ma[3]*ma[2] ) / det;

mr[6] =    ma[3]*ma[7] - ma[6]*ma[4]   / det;
mr[7] = -( ma[0]*ma[7] - ma[6]*ma[1] ) / det;
mr[8] =    ma[0]*ma[4] - ma[1]*ma[3]   / det;
}
```

Q24. How do I calculate the inverse of a 4x4 matrix?

As with 3x3 matrices, either Kramer's rule can be applied or the matrix can be solved as a set of linear equations.

An efficient way is to make use of the existing 'C' functions defined to calculate the determinant and inverse of a 3x3 matrix.

In order to implement Kramer's rule with 4x4 matrices, it is necessary to determine individual sub-matrices. This is achieved by the following routine:

```
void m4_submat( MATRIX4 mr, MATRIX3 mb, int i, int j )
  {
  int ti, tj, idst, jdst;

  for ( ti = 0; ti < 4; ti++ )
    {
    if ( ti < i )
      idst = ti;
    else
      if ( ti > i )
        idst = ti-1;

    for ( tj = 0; tj < 4; tj++ )
      {
      if ( tj < j )
        jdst = tj;
      else
        if ( tj > j )
          jdst = tj-1;

      if ( ti != i && tj != j )
        mb[idst*3 + jdst] = mr[ti*4 + tj ];
```

```
      }
    }
  }

--------------------------
```

The determinant of a 4x4 matrix can be calculated as follows:

```
--------------------------

VFLOAT m4_det( MATRIX4 mr )
  {
  VFLOAT  det, result = 0, i = 1;
  MATRIX3 msub3;
  int     n;

  for ( n = 0; n < 4; n++, i *= -1 )
    {
    m4_submat( mr, msub3, 0, n );

    det     = m3_det( msub3 );
    result += mr[n] * det * i;
    }

  return( result );
  }

--------------------------
```

And the inverse can be calculated as follows:

```
--------------------------

int m4_inverse( MATRIX4 mr, MATRIX4 ma )
  {
  VFLOAT  mdet = m4_det( ma );
  MATRIX3 mtemp;
  int     i, j, sign;

  if ( fabs( mdet ) < 0.0005 )
    return( 0 );

  for ( i = 0; i < 4; i++ )
    for ( j = 0; j < 4; j++ )
      {
      sign = 1 - ( (i +j) % 2 ) * 2;

      m4_submat( ma, mtemp, i, j );

      mr[i+j*4] = ( m3_det( mtemp ) * sign ) / mdet;
      }

  return( 1 );
  }

--------------------------
```

Having a function that can calculate the inverse of any 4x4 matrix is an incredibly useful tool. Application include being able to calculate the base matrix for splines, inverse rotations and rearranging matrix equations.

Q25. How do I calculate the inverse of a matrix using linear equations?

If a matrix M exists, such that:

```
      | A B C |
  M = | D E F |
      | G H I |
```

then the inverse exists:

```
      | P Q R |
  M' = | S T U |
      | V W X |
```

and the following expression is valid:

$$M \cdot M^{-1} = I$$

$$\begin{vmatrix} A & B & C \\ D & E & F \\ G & H & I \end{vmatrix} \cdot \begin{vmatrix} P & Q & R \\ S & T & U \\ V & W & X \end{vmatrix} = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

The inverse can then be calculated through the solution as a set of linear equations ie.:

$$\begin{vmatrix} AP + BS + CV \\ DP + ES + FV \\ GP + HS + IV \end{vmatrix} = \begin{vmatrix} 1 \\ 0 \\ 0 \end{vmatrix} \quad \text{Column 0 (X)}$$

$$\begin{vmatrix} AQ + BT + CW \\ DQ + ET + FW \\ GQ + HT + IW \end{vmatrix} = \begin{vmatrix} 0 \\ 1 \\ 0 \end{vmatrix} \quad \text{Column 1 (Y)}$$

$$\begin{vmatrix} AR + BU + CX \\ DR + EU + FX \\ GR + HU + IX \end{vmatrix} = \begin{vmatrix} 0 \\ 0 \\ 1 \end{vmatrix} \quad \text{Column 2 (Z)}$$

Transforms

Q26. What is a rotation matrix?

A rotation matrix is used to rotate a set of points within a coordinate system. While the individual points are assigned new coordinates, their relative distances do not change.

All rotations are defined using the trigonometric "sine" and "cosine" functions.

For a two-dimensional coordinate system, the rotation matrix is as follows:

$$\begin{vmatrix} \cos(A) & -\sin(A) \\ \sin(A) & \cos(A) \end{vmatrix}$$

```
| sin(A)   cos(A) |
```

With the rotation angle A set to zero, this generates the identity matrix:

```
    |  1  0 |
I = |       |
    |  0  1 |
```

If the rotation is set to +90 degrees, then the matrix is as follows:

```
    |  0 -1 |
M = |       |
    |  1  0 |
```

If the rotation is set to -90 degrees, then the matrix is as follows:

```
    |  0  1 |
M = |       |
    | -1  0 |
```

Negating the rotation angle is equivalent to generating the transpose of the matrix.

If a rotation matrix is multiplied with its transpose, the result is the identity matrix.

Q27. How do I generate a rotation matrix in the X-axis?

Use the 4x4 matrix:

```
    |  1  0       0       0 |
M = |  0  cos(A) -sin(A)  0 |
    |  0  sin(A)  cos(A)  0 |
    |  0  0       0       1 |
```

Q28. How do I generate a rotation matrix in the Y-axis?

Use the 4x4 matrix:

```
    |  cos(A)  0  -sin(A)  0 |
M = |  0       1   0       0 |
    |  sin(A)  0   cos(A)  0 |
    |  0       0   0       1 |
```

Q29. How do I generate a rotation matrix in the Z-axis?

Use the 4x4 matrix:

```
    |  cos(A)  -sin(A)   0   0 |
M = |  sin(A)   cos(A)   0   0 |
    |  0        0        1   0 |
```

```
          |  0        0        0   1 |
```

Q30. What are Euler angles?

Euler angles are the name given to the set of rotation angles which specify the rotation in each of the X, Y and Z rotation axii.

These are specfied in vector format eg. |x y z| and can be stored as a VECTOR data structure.

For example, the set

| 0 0 0 | will always generate the identity matrix.

Other angles are represented as follows:

| 90 0 0 | is a rotation of +90 degrees in the X-axis.

| 0 90 0 | is a rotation of +90 degrees in the Y-axis and

| 0 0 90 | is a rotation of +90 degrees in the Z-axis.

Euler angles can be represented using a single vector data structure.

Q31. What are Yaw, Roll and Pitch?

Yaw, Roll and Pitch are aeronautical terms for rotation using the Euclidean coordinate system (Euler angles), relative to the local coordinate system of an aeroplane.

Imagine you are viewing an aeroplane from above and from directly behind.

The Z-axis is lined up with the tail and nose of the aeroplane. The X-axis runs from the tip of the left wing to the tip of the right wing.

The Y axis points straight up from the ground.

Pitch then becomes rotation in the X-axis, Yaw becomes rotation in the Y-axis and Roll becomes rotation in the Z-axis.

Q32. How do I combine rotation matrices?

Rotation matrices are combined together using matrix multiplication. As a result, the order of multiplication is very important.

Q33. What is Gimbal lock?

Gimbal lock is the name given to a problem that occurs with the use of Euler angles. Because the final rotation matrix depends on the order of multiplication, it is sometimes the case that the rotation in one axis will be mapped onto another rotation axis.

Even worse, it may become impossible to rotate an object in a desired axis. This is called Gimbal lock.

For example, assume that an object is being rotated in the order Z,Y,X and that the rotation in the Y-axis is 90 degrees.

In this case, rotation in the Z-axis is performed first and therefore correctly. The Y-axis is also rotated correctly. However, after rotation in the Y axis, the X-axis is rotated onto the Z-axis.

Thus, any rotation in the X-axis actually rotates the object in the Z-axis. Even worse, it becomes to rotate the object in the X-axis.

The only solution to this problem is to make use of Quaternions.

Q34. What is the correct way to combine rotation matrices?

Really, there is no "correct way" of combining rotation matrices. However, in order to be able to predict the result of combining matrices together, some organisation is required. This is also necessary if a full 3D matrix library is to be built.

The simplest way to rotate an object is to multiply the matrices using the order:

```
M = X.Y.Z
```

where M is the final rotation matrix, and X,Y,Z are the individual rotation matrices. This defines a rotation in the X-axis (pitch) first, followed by the Y-axis (yaw) and a final rotation in the Z-axis (roll).

However, whenever the view from the camera viewpoint is being evaluated, then the order and signs of the rotation is reversed.

For example, if you are standing up, and turn to your left, everything in your field of view appears to move towards the right.

However, someone else facing you will say that you turned towards their right.

Thus the view from the camera is modelled using the order:

```
M = -Z.-Y.-X
```

This is the inverse (or transpose) of the rotation matrix generated if the camera were being rendered as another object.

Q35. How do I generate a rotation matrix from Euler angles?

At first glance, the most obvious method to generate a rotation matrix from a set of Euler angles is to generate each matrix individually and multiply all three together ie.

```
m3_rotx( mat_x,     vec -> angle_x );
m3_roty( mat_y,     vec -> angle_y );
m3_rotz( mat_z,     vec -> angle_z );
m3_mult( mat_tmp,   mat_z, mat_y   );
```

```
  m3_mult( mat_final, mat_tmp, mat_x );
```

This set of calls could be placed in a separate routine eg.

```
  m3_fromeuler( MATRIX *mat_final, VECTOR3 *euler )
```

However, to perform this sequence of calls is very wasteful in terms of processing time. Given that each 4x4 rotation matrix is guaranteed to have 10 elements with value zero (0), 2 elements with value one (1) and four others of arbitary value, over 75% of every matrix operation is wasted. This does not include the set up and initialisation of each matrix.

Altogether, over 75% of all matrix operations are spent processing arithmetic expressions which lead to either zero or one.

A more efficient way must be found. Fortunately, there is another way of determining the final resulting matrix.

If all three matrices are combined in algebraic format, the following expression is defined:

```
  M = X.Y.Z
```

where M is the final matrix,
X is the rotation matrix for the X-axis,
Y is the rotation matrix for the Y-axis,
Z is the rotation matrix for the Z-axis.

Expanding into rotation matrices in algebraic format gives:

```
       |  1  0  0 |
  X  = |  0  A -B |
       |  0  B  A |

       |  C  0 -D |
  Y  = |  0  1  0 |
       |  D  0  C |

       |  E -F  0 |
  Z  = |  F  E  0 |
       |  0  0  1 |
```

where A,B are the cosine and sine of the X-axis rotation axis,
C,D are the cosine and sine of the Y-axis rotation axis,
E,F are the cosine and sine of the Z-axis rotation axis.

Then the expression:

```
  M  = X.Y.Z
```

can be split into two matrix multiplications:

```
  M' = X.Y

  M  = M'.Z
```

Evaluating M' first:

```
  M' = X.Y
```

```
          | 1  0  0 |   | C  0 -D |
    M' =  | 0  A -B | . | 0  1  0 |
          | 0  B  A |   | D  0  C |

          | 1.C + 0.0 +  0.D   1.0 + 0.1 +  0.0   1.-D + 0.0 +  0.C |
    M' =  | 0.C + A.0 + -B.D   0.0 + A.1 + -B.0   0.-D + A.0 + -B.C |
          | 0.C + B.0 +  A.D   0.0 + B.1 +  A.0   0.-D + B.0 +  A.C |
```

Simplifying M' gives:

```
          |  C     0  -D   |
    M' =  | -B.D   A  -B.C |
          |  A.D   B   A.C |
```

Evaluating M gives:

```
    M  = M'.Z

          |  C   0  -D  |   | E -F  0 |
    M  =  | -BD  A  -BC | . | F  E  0 |
          |  AD  B   AC |   | 0  0  1 |

          |   C.E + 0.F +  -D.0    C.-F + 0.E + -D.0     C.0 + 0.0 +  -D.1
|
    M  =  | -BD.E + A.F + -BC.0  -BD.-F + A.E + -BC.0  -BD.0 + A.0 + -BC.1
|
          |  AD.E + B.F +  AC.0   AD.-F + B.E +  AC.0   AD.0 + 0.0 +  AC.1
|
```

Simplifying M gives a 3x3 matrix:

```
          |  CE      -CF      -D  |
    M  =  | -BDE+AF  -BDF+AE  -BC |
          |  ADE+BF  -ADF+BE   AC |
```

This is the final rotation matrix. As a 4x4 matrix this is:

```
          |  CE      -CF      -D   0 |
    M  =  | -BDE+AF   BDF+AE  -BC  0 |
          |  ADE+BF  -ADF+BE   AC  0 |
          |  0        0        0   1 |
```

The individual values of A,B,C,D,E and F are evaluated first. Also, the values of BD and AD are also evaluated since they occur more than once.

Thus, the final algorithm is as follows:

```
    ----------------------

    A       = cos(angle_x);
    B       = sin(angle_x);
    C       = cos(angle_y);
    D       = sin(angle_y);
    E       = cos(angle_z);
    F       = sin(angle_z);

    AD      =   A * D;
    BD      =   B * D;
```

```
mat[0]  =   C * E;
mat[1]  =  -C * F;
mat[2]  =  -D;
mat[4]  = -BD * E + A * F;
mat[5]  =  BD * F + A * E;
mat[6]  =  -B * C;
mat[8]  =  AD * E + B * F;
mat[9]  = -AD * F + B * E;
mat[10] =   A * C;

mat[3]  =  mat[7] = mat[11] = mat[12] = mat[13] = mat[14] = 0;
mat[15] =  1;
```

Using basic matrix calculations, the operation count would reach 128 multiplications, 96 additions and 80 assignments operations.

Using the optimised algorithm, only 12 multiplications, 6 subtractions and 18 assignment operations are required.

So, it is obvious that by using the optimised algorithm, a performance achievement of 1000% is achieved!

Q36. How do I convert a rotation matrix to Euler angles?

This operation is the exact opposite to the one answered in the question above. Given that the rotation matrix is:

```
      |  CE      -CF      -D   0 |
M  =  | -BDE+AF   BDF+AE  -BC  0 |
      |  ADE+BF  -ADF+BE   AC  0 |
      |  0        0        0   1 |
```

where A,B are the cosine and sine of the X-axis rotation axis,
C,D are the cosine and sine of the Y-axis rotation axis,
E,F are the cosine and sine of the Z-axis rotation axis.

Then the entire process can be reversed:

```
angle_y = D = -asin( mat[2]);
C           =  cos( angle_y );
angle_y    *= RADIANS;

if ( fabs( angle_y ) > 0.0005 )
  {
  trx      =  mat[10] / C;
  try      = -mat[6]  / C;

  angle_x  = atan2( try, trx ) * RADIANS;

  trx      =  mat[0] / C;
  try      = -mat[1] / C;

  angle_z  = atan2( try, trx ) * RADIANS;
```

```
    }
  else
    {
    angle_x  = 0;

    trx      = mat[5];
    try      = mat[4];

    angle_z  = atan2( try, trx ) * RADIANS;
    }

  angle_x = clamp( angle_x, 0, 360 );
  angle_y = clamp( angle_y, 0, 360 );
  angle_z = clamp( angle_z, 0, 360 );

  ----------------------------------
```

Q37. How do I generate a rotation matrix for a selected axis and angle?

The only way to generate this type of rotation matrix is through the use of quaternion mathematics.

See question [Q47. How do I convert a quaternion to a rotation matrix?] for further details.

Q38. How do I generate a rotation matrix to map one vector onto another?

When developing animation software, a common requirement is to find a rotation matrix that will map one direction vector onto another.

This problem may be visualised by considering the two direction vectors to be attached at their starting points. Then the entire rotation space forms a unit sphere.

In theory, there are an infinite number of rotation axii and angles that will map one vector onto the other. All of these axii lie on the plane where all of the points are the exact same distance from both vectors.

However, only one solution is of practical interest. This is the path which covers the shortest angular distance between the two vectors.

The rotation axis to this path is calculated by taking the cross product between the two vectors:

$$Vaxis = Vs \times Vf$$

The rotation angle is calculated by taking the dot product between the two vectors:

$$Vangle = \cos^{-1}(Vs \cdot Vf)$$

One practical application of the solution to this problem is finding the shortest flight path between two cities. In this case, each city is represented as a direction vector generated from spherical coordinates. Since planet Earth is spherical, the desired flight path

is the shortest angular rotation between the two cities.

Q39. What is a translation matrix?

A translation matrix is used to position an object within 3D space without rotating in any way. Translation operations using matrix multiplication can only be performed using 4x4 matrices.

If the translation is defined by the vector [X Y Z], then the 4x4 matrix to implement translation is as follows:

```
      | 1  0  0  X |
      |            |
      | 0  1  0  Y |
  M = |            |
      | 0  0  1  Z |
      |            |
      | 0  0  0  1 |
```

If the vector is [0 0 0] then the vertex list will remain as before.

Q40. What is a scaling matrix?

A scaling matrix is used to enlarge or shrink the size of a 3D model.

If the scaling vector is [X Y Z] then the matrix to perform this is as follows:

```
      | X  0  0  0 |
      |            |
      | 0  Y  0  0 |
  M = |            |
      | 0  0  Z  0 |
      |            |
      | 0  0  0  1 |
```

If the scaling vector is [1 1 1], then this generates the identity matrix and vertex geometry will remain unchanged.

Q41. What is a shearing matrix?

A shearing matrix is used to make a 3D model appear to slant sideways. For example, "italic" text requires each character to slant towards the right.

In three dimensions six possible shearing directions exist:

- shear X by Y
- shear X by Z
- shear Y by X
- shear Y by Z
- shear Z by X
- shear Z by Y

All six shearing directions may be combined into a single matrix:

```
      | 1    Syx  Szx  0 |
      |                  |
      | Sxy  1    Szy  0 |
  M = |                  |
      | Sxz  Syz  1    0 |
      |                  |
      | 0    0    0    1 |
      |                  |
```

Where Sij implements a shear of I by J

Thus, Sxy shears X by Y

In theory, rotation in three dimensions may be considered a combination of six shearing directions.

Q42. How do I perform linear interpolation between two matrices?

Given two rotation matrices, the problem is to find a way of determining intermediate positions specified by a parametric variable t, where t ranges from 0.0 to 1.0

This can be achieved by converting the two matrices into either Euler angles or Spherical rotation angles (via quaternions) and a translation vector.
In either case, each matrix is converted into a pair of 3D vectors.

Interpolation between these two vectors can then be performed through the use of the standard linear interpolation equation:

```
    Vr = Va + t .(Vb - Va )
```

where Vr is the resulting vector
Va is the start position vector
Vb is the final position vector

This equation may be applied to both translation and rotation vectors.

Once determined, the resulting translation and rotation are then converted back into the desired intermediate matrix.

Q43. How do I perform cubic interpolation between four matrices?

Given four rotation or translation matrices, the problem is to find a way of determining intermediate positions specified by a parametric variable t.

This can be achieved by making use of cubic interpolation. As with linear interpolation, the four matrices are converted into their corresponding translation and rotation vectors (Again, either Euler angles or spherical rotation angles).

Each set of four vectors is then converted into a single geometry

vector G. Through the use of spline mathematics, this geometry vector is converted into an interpolation matrix M.

If the geometry vector is defined as:

```
       | x1 x2 x3 x4 |
   G = | y1 y2 y3 y4 |
       | z1 z2 z3 z4 |
```

Then multiplication by the base matrix:

```
       |  -4.5    9.0  -5.5  1.0  |
  Mb = |  13.5  -22.5   9.0  0.0  |
       | -13.5   18.0  -4.5  0.0  |
       |   4.5   -4.5   1.0  0.0  |
```

will generate the 3x4 interpolation matrix Mi:

```
  Mi = G .Mb
```

This can be implemented through a standard matrix-vector multiplication.

Interpolation can then be performed by the use of the parametric variable t:

```
      R  = Mi . t

                          |t^3|
  | xr |   | A B C D |   |t^2|
  | yr | = | E F G H | . |t  |
  | zr |   | I J K L |   |1  |
```

The result vector can then be converted back into a rotation or translation matrix.

It should be noted that the rotation paths that are generated may occasionally become rather loopy. This is normal, as the algorithm is trying to find the path with the least amount of rotation between all four vectors.

Of the two methods, spherical rotation angles will usually be seen to provide the cleanest interpolation paths for rotation.

Q44. How can I render a matrix?

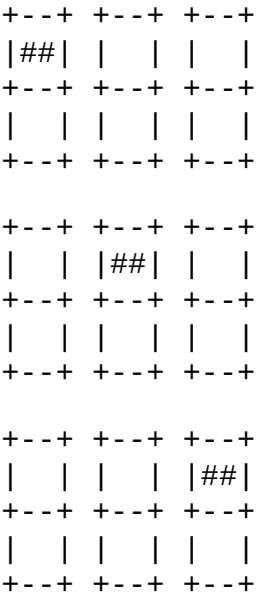
When using a graphics window for 3D animation, it is convenient to be able to view a rotation matrix concurrently with the animation.

However, displaying a rotation matrix as an array of numeric values does not provide a very meaningful context.

An alternative to rendering numeric data is to make use of graphical display methods such as bar-graphs.

Much like a graphic equalizer on a stereo, a rotation matrix may be displayed in a bar graph format. Each element of the rotation matrix is rendered as an individual bar-graph in the range -1 to +1.

A 3x3 matrix would look like the following:



In this case, the rotation matrix is the identity matrix, since each element in the major diagonal is +1, and all others are zero.

For added visual clarity, parameters which are negative may shaded in a different colour than those which are positive.

QUATERNIONS

Q45. What are quaternions?

Quaternions extend the concept of rotation in three dimensions to rotation in four dimensions. This avoids the problem of "gimbal-lock" and allows for the implementation of smooth and continuous rotation.

In effect, they may be considered to add a additional rotation angle to spherical coordinates ie. Longitude, Latitude and Rotation angles

A Quaternion is defined using four floating point values |x y z w|.

These are calculated from the combination of the three coordinates of the rotation axis and the rotation angle.

Q46. How do quaternions relate to 3D animation?

As mentioned before, Euler angles have the disadvantage of being susceptible to "Gimbal lock" where attempts to rotate an object fail due to the order in which the rotations are performed.

Quaternions are a solution to this problem. Instead of rotating an object through a series of successive rotations, a quaternion allows the programmer to rotate an object through a single arbitary rotation axis.

Because the rotation axis is specifed as a unit direction vector, it may be calculated through vector mathematics or from spherical coordinates ie (longitude/latitude).

Quaternions offer another advantage in that they be interpolated. This allows for smooth and predictable rotation effects.

Q47. How do I convert a quaternion to a rotation matrix?

Assuming that a quaternion has been created in the form:

Q = |X Y Z W|

Then the quaternion can then be converted into a 4x4 rotation matrix using the following expression:

$$M = \begin{vmatrix} 1 - 2Y^2 - 2Z^2 & 2XY - 2ZW & 2XZ + 2YW \\ 2XY + 2ZW & 1 - 2X^2 - 2Z^2 & 2YZ - 2XW \\ 2XZ - 2YW & 2YZ + 2XW & 1 - 2X^2 - 2Y^2 \end{vmatrix}$$

If a 4x4 matrix is required, then the bottom row and right-most column may be added.

The matrix may be generated using the following expression:

```
----------------

xx      = X * X;
xy      = X * Y;
xz      = X * Z;
xw      = X * W;

yy      = Y * Y;
yz      = Y * Z;
yw      = Y * W;

zz      = Z * Z;
zw      = Z * W;

mat[0]  = 1 - 2 * ( yy + zz );
mat[1]  =     2 * ( xy - zw );
mat[2]  =     2 * ( xz + yw );

mat[4]  =     2 * ( xy + zw );
mat[5]  = 1 - 2 * ( xx + zz );
mat[6]  =     2 * ( yz - xw );

mat[8]  =     2 * ( xz - yw );
mat[9]  =     2 * ( yz + xw );
mat[10] = 1 - 2 * ( xx + yy );

mat[3]  = mat[7] = mat[11 = mat[12] = mat[13] = mat[14] = 0;
mat[15] = 1;

----------------
```

Q48. How do I convert a rotation matrix to a quaternion?

A rotation may be converted back to a quaternion through the use of the following algorithm:

The process is performed in the following stages, which are as follows:

Calculate the trace of the matrix T from the equation:

$$T = 4 - 4x^2 - 4y^2 - 4z^2$$

$$= 4(1 - x^2 - y^2 - z^2)$$

```
= mat[0] + mat[5] + mat[10] + 1
```

If the trace of the matrix is greater than zero, then perform an "instant" calculation.

```
S = 0.5 / sqrt(T)

W = 0.25 / S

X = ( mat[9] - mat[6] ) * S

Y = ( mat[2] - mat[8] ) * S

Z = ( mat[4] - mat[1] ) * S
```

If the trace of the matrix is less than or equal to zero then identify which major diagonal element has the greatest value.

Depending on this value, calculate the following:

```
Column 0:
  S  = sqrt( 1.0 + mr[0] - mr[5] - mr[10] ) * 2;

  Qx = 0.5 / S;
  Qy = (mr[1] + mr[4] ) / S;
  Qz = (mr[2] + mr[8] ) / S;
  Qw = (mr[6] + mr[9] ) / S;

Column 1:
  S  = sqrt( 1.0 + mr[5] - mr[0] - mr[10] ) * 2;

  Qx = (mr[1] + mr[4] ) / S;
  Qy = 0.5 / S;
  Qz = (mr[6] + mr[9] ) / S;
  Qw = (mr[2] + mr[8] ) / S;

Column 2:
  S  = sqrt( 1.0 + mr[10] - mr[0] - mr[5] ) * 2;

  Qx = (mr[2] + mr[8] ) / S;
  Qy = (mr[6] + mr[9] ) / S;
```

```
Qz = 0.5 / S;
Qw = (mr[1] + mr[4] ) / S;
```

The quaternion is then defined as:

```
Q = | Qx Qy Qz Qw |
```

Q49. How do I convert a rotation axis and angle to a quaternion?

Given the rotation axis and angle, the following algorithm may be used to generate a quaternion:

```
sin_a = sin( angle / 2 )
cos_a = cos( angle / 2 )

qx    = axis -> x / sin_a
qy    = axis -> y / sin_a
qz    = axis -> z / sin_a
qw    = cos_a
```

Q50. How do I convert a quaternion to a rotation axis and angle?

A quaternion can be converted back to a rotation axis and angle using the following algorithm:

```
cos_angle  = qr -> qw;
angle      = acos( cos_angle ) * 2 * RADIANS;
sin_angle  = sqrt( 1.0 - cos_angle * cos_angle );


if ( fabs( sin_angle ) < 0.0005 )
  sa = 1;

axis -> vx = qr -> qx / sa;
axis -> vy = qr -> qy / sa;
axis -> vz = qr -> qz / sa;
```

Q51. How do I convert spherical rotation angles to a quaternion?

A rotation axis itself may be defined using spherical coordinates (latitude and longitude) and a rotation angle

In this case, the quaternion can be calculated as follows:

```
-----------------------
sin_a    = sin( angle / 2 )
cos_a    = cos( angle / 2 )

sin_lat  = sin( latitude )
cos_lat  = cos( latitude )

sin_long = sin( longitude )
cos_long = cos( longitude )
```

```
qx       = sin_a * cos_lat * sin_long
qy       = sin_a * sin_lat
qz       = sin_a * sin_lat * cos_long
qw       = cos_a
-----------------------
```

Q52. How do I convert a quaternion to spherical rotation angles?

A quaternion can be converted to spherical coordinates by extending the conversion process:

```
-----------------------
cos_angle  = q -> qw;
sin_angle  = sqrt( 1.0 - cos_angle * cos_angle );
angle      = acos( cos_angle ) * 2 * RADIANS;

if ( fabs( sin_angle ) < 0.0005 )
  sa = 1;

tx = q -> qx / sa;
ty = q -> qy / sa;
tz = q -> qz / sa;

latitude = -asin( ty );

if ( tx * tx + tz * tz < 0.0005 )
  longitude   = 0;
else
   longitude  = atan2( tx, tz ) * RADIANS;

if ( longitude < 0 )
  longitude += 360.0;
-----------------------
```

Q53. How do I use quaternions to perform linear interpolation between matrices?

For many animation applications, it is necessary to interpolate between two rotation positions of a given object. These positions may have been specified using keyframe animation or inverse kinematics.

Using either method, at least two rotation matrices must be known, and the desired goal is to interpolate between them. The two matrices are referred to as the starting and finish matrices(MS and MF).

Using linear interpolation, the interpolated rotation matrix is generated using a blending equation with the parameter T, which ranges from 0.0 to 1.0.

At T=0, the interpolated matrix is equal to the starting matrix.
At T=1, the interpolated matrix is equal to the finishing matrix.

Then the interpolated rotation matrix (MI) is specified as:

```
MI = F( MS, MI, T )
```

where F is a blending function.

The first stage in interpolating between the two matrices is to determine the rotation matrix that will convert MS to MF. This is achieved using the following expression:

$$T = Ms^{-1} \cdot Mf$$

where Ms is the start matrix,
Mf is the finish matrix,
and T is the intermediate matrix.

The next stage is to convert this matrix into a rotation axis and angle. This is achieved by converting the matrix into a quaternion and finally into the required rotation axis and angle.

In order to generate the interpolated rotation matrix, it is only necessary to scale the rotation angle and convert this angle and the rotation axis back into a rotation matrix.

Using a 4x4 matrix library, this is as follows:

```
m4_transpose(    mt, ms );               /* Inverse            */
m4_mult(         ms, mt, mb );           /* Rotation matrix    */
m4_to_axisangle( ms, axis, angle );      /* Rotation axis/angle */


for ( t = 0; t < 1.0; t += 0.05 )
  {
  m4_from_axisangle( mi, axis, angle * t ); /* Final interpolation */

  ... whatever ...

  }
```

where t is the interpolation factor ranging from 0.0 to 1.0

Q54. How do I use quaternions to perform cubic interpolation between matrices?

For some applications, it may not be convenient or possible to use linear interpolation for animation purposes. In this case, cubic interpolation is another alternative.

In order to use cubic interpolation, at least four rotation matrices must be known.

Each of these is then converted into a set of spherical rotations via quaternions and spherical rotation angles (ie. longitude, latitude and rotation angle).

These are then multiplied with the base matrix for a Cardinal spline curve. This interpolation matrix can then be used to determine the intermediate spherical rotation angles.

Once the interpolated coordinates are known (latitude, longitude and rotation angle), the interpolated rotation matrix can then be generated through the conversion to quaternions.

Using a 4x4 matrix library, the algorithm is as follows:

```
  ------------------------------------------------------------------------

  for ( n = 0; n < 4; n++ )
    m4_to_spherical( mat[n], &v_sph[n] );      /* Spherical coordinates
*/

  m4_multspline( m_cardinal, v_sph, v_interp ); /* Interpolation vector
*/

  ...

  v3_cubic( v_pos, v_interp, t );            /* Interpolation */

  m4_from_spherical( m_rot, v_pos );        /* Back to a matrix */

  ------------------------------------------------------------------------
```

Fast affine texture mapping (fatmap.txt)

by

Mats Byggmastar
a.k.a.
MRI / Doomsday
mri@penti.sit.fi

8 Jul. 1996 Jakobstad, Finland
19 Jun. 1996 Espoo, Finland

Read this today, it might be obsolete tomorrow.

Table of contents

1. About this document

This document tries to describe how to make fast affine texture mapping. The document describes both the general structure as well as the more critical parts such as inner loops. The information is aimed at both beginners and also at people who maybe already have a working texture mapper but are looking for more speed. The goal was to make a good document that would be useful today, not already be obsolete. So I'm giving you the best information I can possibly come up with.

You don't get the information for free though. You will have to invest some

of your own effort and actually learn what's going on in the inner loops and select the parts that will be most suitable for you.

The information is based on my own work and findings but also on information found on the net, especially articles posted to the newsgroup comp.graphics.algorithms and on ideas given to me by other coders. IRC channel #coders is usually a good place to get new ideas and help. Many of the coders there are willing to share ideas and answer decent questions.

I am not claiming that the methods described here are THE fastest methods of doing texture mapping. But these methods are what coders are using today and they ARE fast.

To get the most out of this document you should have a good understanding of 386+ Assembly and C. The asm code and optimizations are aimed especially for the Intel Pentium CPU.

Note that the C code given is only meant as some sort of pseudo code. C is most of the time much easier to read that asm. For your information I have the whole texture mapping function in asm. This is overkill, I know, but this way I get full control over the optimization. In C I can only _hope_ that the compiler makes the best code possible. I'm certain that a human brain still is better to optimize code than a compiler. I do not say this because I'm a true asm-freak. In fact, I had programmed C for a year before even considering learning asm.

I should say that I do not have a masters degree in computer graphics. I'm merely a 24 year old computer and telecom engineer (B.Sc.) that found interest in this area. I have never taken any computer graphics related course in school so if you think I misuses some expressions or terms, or even leave out some expressions or terms where I should use them, you might very well be right and I wrong.

Also I have to confess that I haven't read Chris Hecker's articles in Game Developer magazine (http://www.gdmag.com). People tell me that they are good. You should probably take a look at them also.

2. Disclaimer

Some parts of the technical discussion at the end of the document might not be 100% accurate as of the actual hardware in the Pentium. But from a programmers point of view the guidelines given should apply anyway.

When I state that a inner loop is e.g. 5 clock ticks per pixel, this don't mean that it will actually run at 5 clock ticks. This is just a theoretical minimum when I assume that the instructions pair as expected and there are no cache misses and no delays writing to RAM.

3. Definition of terms

Just so there won't be any confusion:

triangle side	Each triangle has two sides, the left side and the right side.
triangle edge	These makes up the outline of the triangle. Usually one interpolates variables along the triangle edges, both on the left and the right side of the triangle.
triangle section	A triangle is always made up of 3 sections. These are straight lines which makes up the triangle edges. When interpolating along the triangle edges we must calculate

	deltas for each of the 3 sections.
triangle x	The current x value of a triangle edge on the screen. There are two triangle x, one on the left side and one on the right side of the triangle. Between these is the current scanline.
u and v	The x and y components in the bitmap.
dudx and dvdx	Our constant deltas for u and v, du/dx and dv/dx. (constant texture gradients for u and v)

4. Assume the following

We are only drawing triangles. (This is no problem to me as 3D Studio only uses triangles anyway.) Well actually it doesn't have to be triangles, this also works on other types of polygons as long as the texture gradients are constant for the whole polygon surface.

You agree that a fractional part of 8 bit is enough for interpolating u and v on the scanlines of the triangle. Well actually a 16 bit fractional part is better but inner loops are usually much simpler to do if we only use 8 bits.

Bitmaps always has a width of 256 pixels and a maximum height of 256 pixels. In some of the inner loops we must also assume that the bitmaps are aligned on 64k.

The CPU is a Pentium and we are in 32 bit protected mode and flat memory model (TASM+WATCOM+PMODE/W).

5. The slow method

The slow method of doing texture mapping is to interpolate u and v (and triangle x) on both the left and right side of the triangle and then calculate du/dx and dv/dx for each scanline. Then interpolate u and v when drawing the scanline. To make this even slower you could first interpolate the u and v (and triangle x) on both sides of the triangle and store the values in a edge buffer. Then pick the values from the edge buffer and draw each scanline.

6. A faster method

Don't use a edge buffer as described in the slow method above. Calculate the edge deltas when you need them and interpolate along the edges at the same time you draw each scanline. It's just as simple to do it this way and a lot faster.

One important thing you should realize is that when texture mapping a triangle (or any type of polygon that has constant texture gradients), you are interpolating two variables, u and v, whose deltas are constant over the whole triangle. I repeat, the deltas are constant for the whole triangle. Make sure you understand this because this is the key to fast texture mapping (or any other type of linear shading for that matter). I guess that the correct term isn't constant deltas, rather constant gradients, but I like the term delta better.

Because the deltas (delta u and delta v) are constant, we only need to calculate them once for the whole triangle. No need to calculate them for each scanline. Also when interpolating u and v along the edges of the

triangle you only need to interpolate u and v on one side of the triangle.
Triangle x must be interpolated on both sides.

7. General structure of the texture mapping function

Here is the general structure of my texture mapping function. If you have Watcom C/C++ you can compile it as is. Just initialize VGA mode 0x13 and call it. I didn't want to include the clipping code as it only would make it more difficult to read. No kind of pre-stepping or any other type of compensation is presented here, this is just the bare bones of the function. It might look big (?) but it is pretty damn simple and efficient if I may say so myself.

You should call the function by passing a pointer to an array of 3 vertex structures and a pointer to the bitmap.

```
    extern char myimage[];  // 256x256 256 color bitmap
    vertex array[3];
    // fill in the values for each vertex in the array here
    DrawTextureTriangle(array, myimage);
```

Note that the function doesn't move the vertex data to some local variables, it uses pointers to each of the structures instead. This makes it extremely simple to later on add more variables in the vertex structure which you will be doing in the case of an environment-bump or Phong-texture-bump mapper. The same function structure can still be used, just add a few variables to the vertex structure, calculate 2 more deltas, interpolate 2 more variables along the left side and make a new inner loop.

```
// This is the only Watcom C/C++ specific part of the function. These
// instructions take a 26:6 bit fixed point number and converts it
// to 32:32 bit. Then divides it with another 16:16 bit fixed point
// number. The result is 16:16 bit. This must be done in asm where we
// can do 64/32 bit divides.

int shl10idiv(int x, int y);
#pragma aux shl10idiv = \
    " mov   edx, eax "\
    " shl   eax, 10  "\
    " sar   edx, 22  "\
    " idiv  ebx      "\
    parm [eax] [ebx] \
    modify exact [eax edx] \
    value [eax]

// sizeof(int) is 4

struct vertex
{
    int x,y;    // screen coordinates (integers)
    int u,v;    // vertex u,v (26:6 bit fixed point)
};

static vertex * left_array[3], * right_array[3];
static int left_section, right_section;
static int left_section_height, right_section_height;
static int dudx, dvdx;
static int left_u, delta_left_u, left_v, delta_left_v;
static int left_x, delta_left_x, right_x, delta_right_x;


inline int RightSection(void)
{
    vertex * v1 = right_array[ right_section ];
```

```
    vertex * v2 = right_array[ right_section-1 ];

    int height = v2->y - v1->y;
    if(height == 0)
        return 0;

    // Calculate the deltas along this section

    delta_right_x = ((v2->x - v1->x) << 16) / height;
    right_x = v1->x << 16;

    right_section_height = height;
    return height;                  // return the height of this section
}

inline int LeftSection(void)
{
    vertex * v1 = left_array[ left_section ];
    vertex * v2 = left_array[ left_section-1 ];

    int height = v2->y - v1->y;
    if(height == 0)
        return 0;

    // Calculate the deltas along this section

    delta_left_x = ((v2->x - v1->x) << 16) / height;
    left_x = v1->x << 16;
    delta_left_u = ((v2->u - v1->u) << 10) / height;
    left_u = v1->u << 10;
    delta_left_v = ((v2->v - v1->v) << 10) / height;
    left_v = v1->v << 10;

    left_section_height = height;
    return height;                  // return the height of this section
}

void DrawTextureTriangle(vertex * vtx, char * bitmap)
{
    vertex * v1 = vtx;
    vertex * v2 = vtx+1;
    vertex * v3 = vtx+2;

    // Sort the triangle so that v1 points to the topmost, v2 to the
    // middle and v3 to the bottom vertex.

    if(v1->y > v2->y) { vertex * v = v1; v1 = v2; v2 = v; }
    if(v1->y > v3->y) { vertex * v = v1; v1 = v3; v3 = v; }
    if(v2->y > v3->y) { vertex * v = v2; v2 = v3; v3 = v; }

    // We start out by calculating the length of the longest scanline.

    int height = v3->y - v1->y;
    if(height == 0)
        return;
    int temp = ((v2->y - v1->y) << 16) / height;
    int longest = temp * (v3->x - v1->x) + ((v1->x - v2->x) << 16);
    if(longest == 0)
        return;

    // Now that we have the length of the longest scanline we can use that
    // to tell us which is left and which is the right side of the triangle.

    if(longest < 0)
    {
        // If longest is neg. we have the middle vertex on the right side.
        // Store the pointers for the right and left edge of the triangle.
```

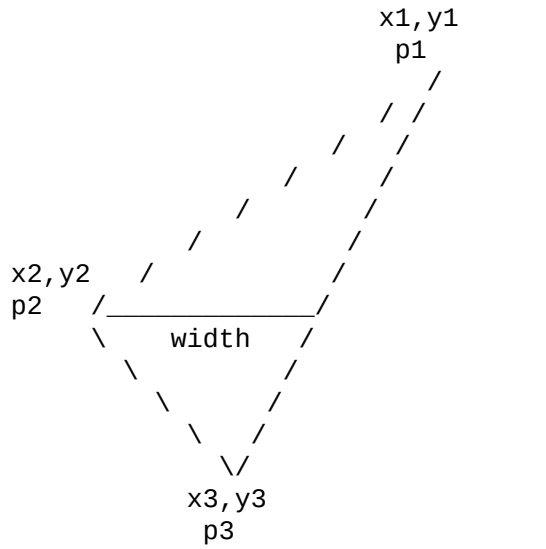
```
        right_array[0] = v3;
        right_array[1] = v2;
        right_array[2] = v1;
        right_section  = 2;
        left_array[0]  = v3;
        left_array[1]  = v1;
        left_section   = 1;

        // Calculate initial left and right parameters
        if(LeftSection() <= 0)
            return;
        if(RightSection() <= 0)
        {
            // The first right section had zero height. Use the next section.
            right_section--;
            if(RightSection() <= 0)
                return;
        }

        // Ugly compensation so that the dudx,dvdx divides won't overflow
        // if the longest scanline is very short.
        if(longest > -0x1000)
            longest = -0x1000;
    }
    else
    {
        // If longest is pos. we have the middle vertex on the left side.
        // Store the pointers for the left and right edge of the triangle.
        left_array[0]  = v3;
        left_array[1]  = v2;
        left_array[2]  = v1;
        left_section   = 2;
        right_array[0] = v3;
        right_array[1] = v1;
        right_section  = 1;

        // Calculate initial right and left parameters
        if(RightSection() <= 0)
            return;
        if(LeftSection() <= 0)
        {
            // The first left section had zero height. Use the next section.
            left_section--;
            if(LeftSection() <= 0)
                return;
        }

        // Ugly compensation so that the dudx,dvdx divides won't overflow
        // if the longest scanline is very short.
        if(longest < 0x1000)
            longest = 0x1000;
    }

    // Now we calculate the constant deltas for u and v (dudx, dvdx)

    int dudx = shl10idiv(temp*(v3->u - v1->u)+((v1->u - v2->u)<<16),longest);
    int dvdx = shl10idiv(temp*(v3->v - v1->v)+((v1->v - v2->v)<<16),longest);

    char * destptr = (char *) (v1->y * 320 + 0xa0000);

    // If you are using a table lookup inner loop you should setup the
    // lookup table here.

    // Here starts the outer loop (for each scanline)

    for(;;)
```

```
    {
        int x1 = left_x >> 16;
        int width = (right_x >> 16) - x1;

        if(width > 0)
        {
            // This is the inner loop setup and the actual inner loop.
            // If you keep everything else in C that's up to you but at
            // least remove this inner loop in C and insert some of
            // the Assembly versions.

            char * dest = destptr + x1;
            int u  = left_u >> 8;
            int v  = left_v >> 8;
            int du = dudx   >> 8;
            int dv = dvdx   >> 8;

            // Watcom C/C++ 10.0 can't get this inner loop any tighter
            // than about 10-12 clock ticks.

            do
            {
                *dest++ = bitmap[ (v & 0xff00) + ((u & 0xff00) >> 8) ];
                u += du;
                v += dv;
            }
            while(--width);
        }

        destptr += 320;

        // Interpolate along the left edge of the triangle
        if(--left_section_height <= 0)  // At the bottom of this section?
        {
            if(--left_section <= 0)     // All sections done?
                return;
            if(LeftSection() <= 0)      // Nope, do the last section
                return;
        }
        else
        {
            left_x += delta_left_x;
            left_u += delta_left_u;
            left_v += delta_left_v;
        }

        // Interpolate along the right edge of the triangle
        if(--right_section_height <= 0) // At the bottom of this section?
        {
            if(--right_section <= 0)    // All sections done?
                return;
            if(RightSection() <= 0)     // Nope, do the last section
                return;
        }
        else
        {
            right_x += delta_right_x;
        }
    }
}
```

8. Equations for the constant deltas

Sort the vertices in the triangle so that the topmost vertex is known as

x1,y1 and the bottom vertex is known as x3,y3. Like the drawing below.




```
  xn,yn     - x,y screen coordinates at vertex n (integers)
  pn        - Value of variable at vertex n to calculate the constant delta
              for. Note that this variable is assumed to have a 6 bit
              fractional part (26:6 bit fixed point).
  width     - Width of the longest scanline in the triangle
```

The reason why I have p as a 26:6 bit fixed point and not 16:16 or 24:8 bit fixed point is just for being able to store u and v with a little higher precision in the 3D structure and still use only words to save space.

Sorting 3 vertices is no more that 3 compares. Another thing: Don't load all x,y,u and v values of the vertices into registers. Use pointers to the vertex structures instead. This will also make it easier when you later on implement your Phong-texture-bump mapper. Something like this:

```
    ; EDX -> vertex 1
    ; ESI -> vertex 2
    ; EDI -> vertex 3
    mov     EAX, [EDX+vertex_y]
    cmp     EAX, [ESI+vertex_y]
    jle     short @@sorta
    xchg    EDX, ESI               ; swap v1 - v2
 @@sorta:
    mov     EAX, [EDX+vertex_y]
    cmp     EAX, [EDI+vertex_y]
    jle     short @@sortb
    xchg    EDX, EDI               ; swap v1 - v3
 @@sortb:
    mov     EAX, [ESI+vertex_y]
    cmp     EAX, [EDI+vertex_y]
    jle     short @@sortc
    xchg    ESI, EDI               ; swap v2 - v3
 @@sortc:
    ; EDX -> topmost vertex
    ; ESI -> middle vertex
    ; EDI -> bottom vertex
```

The following two equations needs only be calculated once for all the constant deltas in the triangle. Skip the triangle if y3 == y1, i.e. if the triangle has zero height. The width can be either positive or negative depending on which side the x2,y2 vertex is. This will be useful information when sorting out which is left and which is the right side of the triangle.

```
              (y2-y1) << 16
```

$$temp = \frac{}{y3-y1}$$

$$width = temp * (x3-x1) + ((x1-x2) << 16)$$

This will give you temp and width as 16:16 bit fixed point.

The equation below is used to calculate the delta for a variable that should be interpolated over the triangle, e.g. texture u. Beware of the denominator in this equation! Make sure it won't cause divide overflow in case the width is less than one pixel. (Remember that width is a 16:16 bit fixed point number.) Note that shift by 10 in the equation. This is because p1,p2,p3 has a 6 bit fractional part. The resulting delta p is a 16:16 bit number. Note that this divide should be done in asm where we can do 64/32 bit divides.

$$delta\ p = \frac{(temp * (p3-p1) + ((p1-p2) << 16)) << 10}{width}$$

So for a texture mapper where we have 2 variables (u,v) to interpolate over the triangle, we have a total of 3 divs and 3 muls to calculate dudx and dvdx.

Here is another equation that can be used to calculate the deltas with. It was posted to the newsgroup comp.graphic.algorithm by Mark Pursey.

> There is a cleaner way, which doesn't rely on finding the widest line:
> A-B-C: a triangle with screen x and y components, as well as t, a
> value which could represent lightning, texture coordinates etc.
> The following equation gives you the increment for t per horizontal pixel:

$$dt/dx = \frac{(At-Ct)*(By-Cy) - (Bt-Ct)*(Ay-Cy)}{(Ax-Cx)*(By-Cy) - (Bx-Cx)*(Ay-Cy)}$$

I've been told that this is the correct way to calculate the deltas (or constant texture gradients). This might very well be true but the other equations gives me good results and the length of the longest scanline for free. In this equation the denominator is reusable for both u and v. This makes a total of 6 muls and 2 divs. Remember to add the necessary shifts if you do this in fixed point.

9. Traditional inner loops

So assuming you have come so far that you have the triangle sorted, the constant deltas calculated, the u and v deltas on the left side calculated, deltas for triangle x calculated for both sides, and you are actually interpolating those values for each scanline, we come to the very core of the texture mapper, the inner loop. I'll first present a few traditional inner loops that interpolates u and v while plotting the scanline. These loops are simple, fast and works very well.

The loops assume the following:

```
 ebx     = ptr to bitmap aligned on 64k. (the low 16 bits zero)
 edi     = ptr to first destination pixel to plot in this scanline
 ebp     = width of scanline (loop counter)
 left_u  = current u on the left edge of the triangle (16:16 bit fixed point)
 left_v  = current v on the left edge of the triangle (16:16 bit fixed point)
 du      = our constant delta u (24:8 bit fixed point)
 dv      = our constant delta v (24:8 bit fixed point)
```

The first loop interpolates the u and v in two 32 bit registers (ecx, edx).

We are one register short here so we use the dudx variable directly in the inner loop. This loop should run at 6 ticks per pixel. eax is not used for anything else than holding the pixel so we could unroll this loop to plot a word or dword at a time.

```
   mov   ecx, [left_u]             ; current u
   mov   edx, [left_v]             ; current u
   shr   ecx, 8                    ; make them 28:8 bit fixed point
   shr   edx, 8
   mov   bl, ch                    ; make ebx point to the first textel
   mov   bh, dh
   mov   esi, [du]

 @@inner:
    add   edx, [dv]                ; update v
    add   ecx, esi                 ; update u
    mov   al, [ebx]                ; get pixel from aligned texture map
    mov   bl, ch
    mov   [edi], al                ; plot pixel
    mov   bh, dh
    inc   edi
    dec   ebp
    jnz   @@inner
```

Just to show that it is also possible to directly interpolate u and v in ebx I'll present this one that uses the carry flag to add the "overflow" from the fractional part to the whole part of u and v.

```
    mov   cl, byte ptr [left_u+1]  ; fractional part of current u
    mov   ch, byte ptr [left_v+1]  ; fractional part of current v
    mov   dl, byte ptr [du]        ; fractional part of delta u
    mov   dh, byte ptr [dv]        ; fractional part of delta v
    mov   bl, byte ptr [left_u+2]  ; whole part of current u
    mov   bh, byte ptr [left_v+2]  ; whole part of current v

 @@inner:
    mov   al, [ebx]                ; get pixel from aligned texture map
    add   cl, dl                   ; update fractional part of u
    adc   bl, byte ptr [du+1]      ; + whole part of dudx (+carry)
    add   ch, dh                   ; update fractional part of v
    adc   bh, byte ptr [dv+1]      ; + whole part of dvdx (+carry)
    mov   [edi], al                ; plot pixel
    inc   edi
    dec   ebp
    jnz   @@inner
```

The following loop uses a combination of interpolation in one 32 bit register (ecx) and the carry overflow method. We have just enough registers in this loop that we don't need to use any memory variables. On the other hand this makes it impossible to unroll it and plot a word or dword at a time. Anyway, this version should run at 5 ticks per pixel.

```
    mov   ecx, [left_u]
    shr   ecx, 8                   ; make it 28:8 bit fixed point
    mov   esi, [du]
    mov   dl, byte ptr [dv]        ; fractional part of delta v
    mov   dh, byte ptr [left_v+1]  ; fractional part of current v
    mov   ah, byte ptr [dv+1]      ; whole part of delta v
    mov   bh, byte ptr [left_v+2]  ; whole part of current v
    mov   bl, ch

 @@inner:
    add   ecx, esi                 ; update u
    mov   al, [ebx]                ; get pixel from aligned texture map
```

```
    mov   bl, ch
    add   dh, dl                    ; update fractional part of v
    adc   bh, ah                    ; + whole part of of delta v (+carry)
    mov   [edi], al                 ; plot pixel
    inc   edi
    dec   ebp
    jnz   @@inner
```

The loop counter (ebp) in the above loop can be removed if we reorder the registers a bit and plot the scanline from right to left.

```
 @@inner:
    add   ecx, ebp
    mov   al, [ebx]
    mov   bl, ch
    add   dh, dl
    adc   dh, ah
    mov   [edi+esi], al
    dec   esi
    jnz   @@inner
```

The loop should now run at 4 clock ticks.

I'm sure there are other ways to make these kind of loops but this is what I could come up with.

After I wrote the above sentence, there was a post in the newsgroup comp.graphics.algorithms by Sean L. Palmer where he presented the following 4 tick loop:

```
    Texture must be aligned on a 64K boundary. Must be 256x256.
    Only 8 bits used for fractions, means shaky textures.
    Start at right end of scanline

    T=texture adr
    D=dest adr+count (start)
    E=dest adr (end)
    X=tex X int          (whole part of initial u)
    x=tex X frac         (fractional part of initial u)
    Y=tex Y int          (whole part of initial v)
    y=tex Y frac         (fractional part of initial v)
    H=tex X step int     (whole part of delta u)
    h=tex X step frac    (fractional part of delta u)
    V=tex Y step int     (whole part of delta v)
    v=tex Y step frac    (fractional part of delta v)
    m=account for borrow for negative Y step, either 0 or 0FFh
    p=texture pixel

    edi=DDDD
    esi=EEEE
    edx=TTYX
    eax=000p
    ebx=x0Yy
    ecx=hmVv
    ebp=000H
    esp=

      mov   dh,bh
    @@L:
      mov   al,[edx]
      add   ebx,ecx
      adc   edx,ebp
      dec   edi
      mov   dh,bh
      cmp   edi,esi
      mov   [edi],al
```

```
      jne   @@L
```

It's not necessary to simulate the loop counter this way. esi is not really used in the loop so we might as well use it as a loop counter and draw the scanline from left to right (the way I like to draw my scanlines). Like this:

```
  @@inner:
    mov   al, [edx]
    add   ebx, ecx
    adc   edx, ebp
    inc   edi
    mov   dh, bh
    dec   esi
    mov   [edi], al
    jnz   @@inner
```

Both of these loops uses eax only to hold the pixel so they can be unrolled to plot a word or dword at a time. In fact, by unrolling this loop to plot a dword per turn it might very well beat the table lookup inner loop presented below. By unrolling this loop we can remove 3 instructions, "inc edi", "dec esi" and "jnz @@inner". This will also mean that the loop will become too tight that will lead to AGI delays instead.

10. Memories from the past

I as many others, started coding asm in real mode and later on moved to protected mode and flat model. The thing I miss about real mode was the ability to have a pointer in the low 16 bit and a variable in the high 16 bit of a 32 bit register. In flat model we need all 32 bits for the pointer. Sure, one can setup a selector and address the data with only the low 16 bits but all prefix bytes can be seen as a 1 clock tick, nonpairable instruction on the Pentium. So addressing with only 16 bit and using a segment override will give 2 prefix bytes or 2 ticks delay.

The following loop in real mode was for a bitmap scaler I once used. We have 4 variables in only 2 registers (edi, ebx).

```
    ; ebx = neg(loop counter)   : source ptr
    ; edi = decision variable   : destination pointer
    ; ecx = frac. part of delta : 1
    ; edx = 1                   : whole part of delta
    ; the delta is 16:16 bit
 @@inner:
    mov   al, [bx]
    mov   es:[di], al
    add   edi, ecx      ; update fractional part : move dest. pointer
    adc   ebx, edx      ; update loop counter    : whole step in bmp (+carry)
    jnc   @@inner       ; jump if loop counter didn't overflow
```

OK, this loop is crap on a Pentium but ain't it pretty? Just two adds to move both pointers, update the decision variable and loop counter. If we only had 64 bit registers on the Pentium...

11. Selfmodifying code

One way to get rid of the memory variables in inner loops is to use selfmodifying code. When you have calculated a constant delta and are about to store it in a memory variable, why don't you store it right into a instruction as a constant in the inner loop? It's just as simple. Just remember to not use CS as segment override as we are in protected mode.

I must warn you about this way of coding, especially on the Pentium (read about the code cache at the end). It can actually make the loop slower even if you think you cut away a few ticks.

Doing more complex shadings like environment-bump or Phong-texture-bump, selfmodifying code might be the only way to get it to run at all. I.e. not having to write to any memory variables from the inner loop. If you are about to make your loop selfmodifying, compare it with your old loop by actually timing a typical scene. Then you'll know if you gained anything.

If your loop is faster with selfmodifying code and the environment your application is aimed for allows selfmodifying code, I'd definitely say go for it, use selfmodifying code.

12. Unrolled and selfmodifying inner loops

I don't really see these as an alternative to the traditional inner loops on the Pentium. I present them here just because they are interesting.

The deltas are constant so the offsets for each pixel in each scanline into the bitmap will also be constant. I.e. we can precalculate a whole run and use that in the inner loop. The inner loops for these type of texture mappers can look very different. The most radical must be to unroll it all the way and to plug in the offsets right into the mov instructions, i.e. selfmodifying code. These completely unrolled loops will be pretty big also. The loop below is 14 byte per pixel which means over 4k code for a whole 320 pixel scanline. The loop will take up half of the code cache. Ouch! (read about the code cache at the end). Here is some code that shows the principle of this type of "inner loop":

```
    jmp   ecx                   ; Jump to the right place in the "loop"
    mov   al, [esi+12345]
    mov   [edi+319], al
    mov   al, [esi+12345]       ; Get pixel
    mov   [edi+318], al         ; Plot pixel
    ......
    mov   al, [esi+12345]       ; '12345' is the selfmodifying part
    mov   [edi+2], al           ; that will be modified once per triangle
    mov   al, [esi+12345]
    mov   [edi+1], al
    mov   al, [esi+12345]
    mov   [edi+0], al
```

Note that we are doing it backwards, from right to left. This makes it easier to setup esi and edi. As the code for each pixel in this loop is 14 byte you will be doing a X*14 when calculating the jump offset. X*14 is (X<<4)-X-X. You should of coarse not plug in the offsets for the whole loop if you only have a small triangle. The length of the longest scanline is a byproduct from the constant delta calculations.

So what about the 1.5 tick per pixel loop?
Well the following peace of code is usually what people think of. I'm not really sure that this is actually 1.5 tick per pixel as the 'mov [edi+?],ax' has a operand size prefix byte. This code will need some work to make the instructions pair on the Pentium. Of coarse this loop also suffers from the same problems as the previous selfmodifying, unrolled loop.

```
    jmp   ecx
    ......
    mov   al, [esi+12345]
    mov   ah, [esi+12345]
    mov   [edi+4], ax
    mov   al, [esi+12345]
```

```
    mov   ah, [esi+12345]
    mov   [edi+2], ax
    mov   al, [esi+12345]
    mov   ah, [esi+12345]
    mov   [edi], ax
```

13. Table lookup inner loops

Now to a cooler method that is not selfmodifying and don't need to be unrolled all the way. The idea is very similar to the unrolled loops above but in this loop we have the offsets stored in a lookup table instead. For each pixel we get the address of the next pixel from the lookup table. This method should be much more Pentium friendly. Also this inner loop don't need to have the bitmap aligned on 64k as the traditional inner loops.

The loop assume the following:

```
 esi     = ptr to bitmap (no alignment needed)
 edi     = ptr to first destination pixel to plot in this scanline
 ebp     = width of scanline (loop counter)
 left_u  = current u on the left edge of the triangle (16:16 bit fixed point)
 left_v  = current v on the left edge of the triangle (16:16 bit fixed point)
 lookup  = ptr to the precalculated lookup table. The lookup table is an
           array of dwords.
```

```
    mov   edx, [lookup]
    xor   eax, eax
    mov   al, byte ptr [left_u+2]
    mov   ah, byte ptr [left_v+2]
    add   esi, eax

  @@inner:
    mov   al, [esi+ebx]         ; Get pixel
    mov   ebx, [edx]            ; Get offset for next pixel
    mov   [edi], al             ; Plot pixel
    add   edx, 4
    inc   edi
    dec   ebp
    jnz   @@inner
```

The same loop could look like this in C:

```
    // destptr = ptr to screen + y*320
    // bitmap  = ptr to bitmap
    // lookup  = ptr to lookup table
    // x1      = start screen x coordinate of scanline
    // width   = width of scanline

    char * dest = destptr + x1;
    char * src  = bitmap + (left_u>>16) + (left_v>>16)*256;

    for(; width--; )
    {
        *(dest++) = src[ *(lookup++) ];
    }
```

The above loop in asm should be 4 clock ticks per pixel on a Pentium. This loop can be changed to plot 4 pixels at a time:

```
  @@inner:
    mov   al, [esi+ebx]         ; Get pixel #1
    mov   ebx, [edx]
```

```
    mov   ah, [esi+ecx]         ; Get pixel #2
    mov   ecx, [edx+4]
    shl   eax, 16               ; Move pixels 1 and 2 to the high word
    add   edi, 4
    mov   al, [esi+ebx]         ; Get pixel #3
    mov   ebx, [edx+8]
    mov   ah, [esi+ecx]         ; Get pixel #4
    mov   ecx, [edx+12]
    rol   eax, 16               ; Swap the high and low words
    add   edx, 16
    mov   [edi], eax            ; Plot all 4 pixels
    dec   ebp
    jnz   @@inner
```

Now this loop is 9 (8 if we assume that shl and rol are pairable in the U pipeline) ticks per 4 pixel with the pixels written as a dword. Very good if we align the write on dword. Use the other loop for very short lines or to get this one aligned on dword and use this for the rest of the scanline.

Calculate the lookup table with the following loop (this loop can also be used to calculate the offsets in the selfmodifying example):
(dudx and dvdx are 16:16 bit fixed point. lookup is an array of dwords)

```
   int du = dudx >> 8;
   int dv = dvdx >> 8;
   int u = 0;
   int v = 0;
   for( width of longest scanline )
   {
      *lookup++ = (u>>8) + (v & 0xffffff00);
      u += du;
      v += dv;
   }

   ; ebx = ecx = 0
   ; esi = delta u  (26:8 bit fixed point)
   ; edi = delta v  (26:8 bit fixed point)
   ; edx = ptr to lookup table
   ; ebp = length of table (the width of the longest scanline)

 @@mklookup:
    mov   eax, ecx
    add   ecx, edi          ; update v
    mov   al, bh
    add   ebx, esi          ; update u
    mov   [edx], eax        ; lookup[edx] = u+256*v
    add   edx, 4
    dec   ebp
    jnz   @@mklookup
```

14. Problems with precalculated runs

The more I play around with inner loops that uses the same precalculated run for each scanline, the more skeptic I get. This is because they all suffers from the same problem, no matter if we use a lookup table or if we have a unrolled selfmodified loop.

In the case of the lookup table inner loop we always start at the beginning of the table when drawing a scanline. This is wrong and will give very bad distortion especially when the triangle is zoomed in close. Always starting at the beginning of the table is the same as ignoring the fractional parts of

the initial u and v of the scanline. So to fix this we should start somewhere into the table depending on the initial fractional parts of u and v. But this is impossible because u and v are interpolated separately on the triangle edge but are fixed to each other in the lookup table. Wilco Dijkstra posted the following solution in comp.graphics.algorithms:

> The basic idea is correct. What you mean is using subpixel positioning with one or two bits precision. For example, for 2 bits subpixel positioning you have to create 4 * 4 tables of the longest scanline. The first table starts at u = v = 0, second u = 0, v = 0.25, third u 0, v = 0.50 fourth u = 0, v = 0.75, fifth u = 0.25, v = 0, etc.
> When stepping down the scanlines, select the table giving the 2 most significant fractional bits of u and v. The maximum error you get is 1/8 in each direction (when proper rounding is used!). Thus this is 64 times more precise than using no subpixel positioning.
>
> The problem is that it's only faster for very large triangles (eg. more than 32 scanlines deep), so it may be faster (and more accurate) to draw the texture in the standard way, without a table.

This method will reduce the distortion. On the other hand the lookup tables will require much more memory that in turn will push out other cached data, not to mention the additional time it takes to setup the tables.

15. Pre-stepping texture coordinates

When we interpolate u, v and triangle x along the left edge of the triangle we always truncates triangle x when drawing a scanline. This is natural because we can only draw whole pixels. When we truncates x we must also adjust the initial u and v of the scanline. Adjusting u and v will give much cleaner and stable textures. Note that this only applies if you use a traditional inner loop. Don't bother doing this if you are using a table lookup inner loop. Kevin Baca sent me the following explanation:

> No matter how you compute screen pixels, you need to "pre-step" your texture coordinates by the difference between actual screen coordinates and screen pixels. It looks like this:
>
> ```
> // sp = screen pixel, sc = screen coordinate.
> float sc, diff, u, v, dudx, dvdx;
> int sp;
>
> sp = (int) sc;
> diff = sc - (float) sp;
> u -= dudx * diff;
> v -= dvdx * diff;
> ```
>
> You can actually do this without multiplies (by calculating a dda for each edge that determines when to add an extra 1 to the texel coordinates).

16. Special case code

It often pays off to make special case code that takes care of the edge delta calculations when a triangle section is 1, 2 or 4 pixels high. Then you can skip the divs and use shifts instead.

I once made a histogram of the length of each scanline in the very popular chrmface.3ds object. This object has about 850 triangles and was scaled up

so it just touched the top and the bottom of a 320x200 pixel screen. The histogram showed that most scanlines was only 1 or 2 pixels wide. This proves that the outer loop is just as important as the inner loop and also that it might be a good idea to have special case code for those 1 or 2 pixel lines.

```
width    number of scanlines
  1     ********************
  2     ******************
  3     **********
  4     ******
  5     ***
  6     **
  7     **
```

17. Clipping

Clipping is most of the time a real pain in the ass implementing. It will always mess up a nice looking routine with extra junk. One possibility is to have two separate functions, one with clipping and one with no clipping. Then test the triangle if it needs clipping before calling any of the functions.

The actual clipping code is not that difficult to implement really. Say if you need to clip a texture mapped scanline, you first have to get the number of pixels you need to skip at the end of the scanline and the number of pixels in the beginning of the scanline. Then subtract the number of pixels skipped from the original scanline width. If you skipped some pixels at the start of the scanline, the new starting u and v must be calculated. This is done by multiplying the pixels skipped by delta u and delta v respectively. And adding the original starting u and v of coarse.

The following code is what I'm using to sort out the stuff:

```
    movsx   EBP, word ptr [left_x+2]    ; Get the integer part from the
    movsx   ECX, word ptr [right_x+2]   ; 16:16 bit numbers.
    mov     EDX, EBP
    sub     EDX, ECX
    ; EDX = width of scanline
    ; ECX = x1
    ; EBP = x2
    mov     EBX, EDX
    sub     EBP, [_RightClip]
    jle     short @@rightok
    sub     EDX, EBP                    ; skip pixels at end
 @@rightok:
    xor     EAX, EAX
    cmp     ECX, [_LeftClip]
    jge     short @@leftok
    mov     EAX, [_LeftClip]
    sub     EAX, ECX
    mov     ECX, [_LeftClip]
 @@leftok:
    sub     EDX, EAX                    ; skip pixels at start
    jle     @@notvisible
    ; EAX = pixels skipped at start
    ; ECX = clipped x1
    ; EDX = clipped width of scanline
```

So now you just have to multiply EAX by delta u and add the original u to get the clipped u. The same apply for v.

18. Clipping using matrices

I've been told that clipping should not be done scanline by scanline in the texture mapping function. But I have yet to find a simple alternative solution to this. Don't confuse the clipping I'm referring to with removal of nonvisible polygons. When we arrive at the texture mapping function we should already have removed those triangles that are backface or outside the viewcone.

Kevin Baca sent me the following explanation on how to decide if vertices should be clipped or not.

> If you use homogeneous matrices to do your transformations it's actually very simple to clip before you do the perspective divide to get screen coordinates.
>
> Using homogeneous coordinates, you get vertices of the form [X Y Z W] after doing the perspective projection. To get actual screen coordinates, you divide X and Y by W. If you are going to "Normalized Device Coordinates" the results of these divisions will be -1 < X' < 1 and -1 < Y' < 1. Therefore, to do clipping you need to perform the following comparison before the perspective divide:
>
> -W < X < W, -W < Y < W.
>
> To clip along the Z axis, you can do the same thing, but I usually use the following comparison instead:
>
> 0 < Z < W.
>
> To do a perspective projection, multiply the projection matrix, P, by the view matrix, V: M = P * V.
>
> The view matrix is the result of all your transformations (translations, rotations, scalings, etc.) of both the model and the camera. For the projection matrix, I use the following:
>
> 1 0 0 0
> 0 a 0 0
> 0 0 b c
> 0 0 f 0
>
> where:
> a = the aspect ratio (width / height of screen)
> b = f * (yon / (yon - hither))
> c = -f * (yon * hither / (yon - hither))
> f = sin(aov / 2) / cos(aov / 2)
> aov = angle of view
> yon = distance to far clipping plane
> hither = distance to near clipping plane
>
> These values allow me to clip using:
> -W < X < W
> -W < Y < W
> 0 < Z < W
>
> After clipping, divide X and Y by W and multiply by the width and height of your screen to get final screen coordinates.

19. Writing a byte, word, dword and other weird things

Now to a weird thing on the Pentium. The Pentium has a so called Write-Back cache. Well, the fact that the Pentium has a Write-Back cache is not weird at all. It's how the Write-Back cache works in practice that is weird if you are used to a Write-Trough cache that is used on the 486.

Write-Trough:
    When we write a byte to memory the byte is always written to RAM. If that
    same byte is also present in the cache, the byte in the cache is also
    updated.

Write-Back:
    When we write a byte to memory the byte is only written to RAM if the
    same byte is not present in the cache. If the byte is present in the
    cache, only the cache will be updated. It is first when a cacheline is
    pushed out from the cache that the whole cacheline will be written to
    RAM.

I have done tests on my system (Pentium 120, L1:8+8k, L2:256k) using the
time stamp counter to see how it actually behaves. These are the results:

Writing to a byte (or aligned word or dword) that is not present in the L1
cache takes 8 clock ticks (no matter if the byte is present in the L2 cache).
If the byte is present in the L1 cache, the same "mov" instruction takes the
theoretical 0.5 clock tick.

This is very interesting and potentially useful. If we e.g. manage to keep
the cacheline where we have our memory variables in the L1 cache, we can
write to them at the same speed as writing to a register. This could be very
useful in the case of a Phong-texture or Phong-texture-bump inner loop where
we need to interpolate many variables and only have 7 registers.

The problem is that our cacheline will be pushed out from the cache as soon
as we start getting cache misses when reading the texture data. Then we are
back at 8 clock tick per write. To fix this we must read a byte from our
cacheline so that it won't be marked as old and thrown out. But this is
usually what we do anyway. We read a variable, interpolates it, uses it and
writes it back.

Juan Carlos Arevalo Baeza presented in an article to comp.graphics.algorithms
another way to make use of the Write-Back cache in a texture mapping inner
loop. The idea is to ensure that the destination pixel written is always
present in the cache. This is done by reading a byte from the destination
cacheline first:

```
     ; edi = ptr to first destination pixel (+1) to plot
     ; esi = ptr to last destination pixel to plot
     ; The scanline is plotted from right to left

      push esi
      mov  al,[edi-1]    ; read the first byte into the cache.

    @@L1:
      lea  esi,[edi-32]
      cmp  esi,[esp]
      jae  @@C
      mov  esi,[esp]
    @@C:
      mov  al,[esi]      ; read the last byte of the 32-byte chunk.
    @@L:
      mov  al,[edx]
      add  ebx,ecx
      adc  edx,ebp
      dec  edi
      mov  dh,bh
      cmp  edi,esi
      mov  [edi],al
      jne  @@L

      cmp  edi,[esp]
      jne  @@L1
```

```
      pop  esi
```

This ensures that whenever you write a pixel, that address is already in the cache, and that's a lot faster. A LOT. My P90 takes 20-40 cycles to read a cache line, so that's around 1 more cycle per pixel. Problems: when painting polys, rows of very few pixels (let's say 1-8 pixels) are the most common, and those don't feel so good about this loop. You can always have two loops for the different lengths.

Another way to speed up writes (that also works on 486) is to collect 4 pixels in a 32 bit register and write all 4 pixels at a time as a aligned dword. This will split the 8 clock tick delay on all 4 pixels making the delay only 2 clock ticks per pixel. This method will almost always gain speed especially if the scanlines are long.

20. The data cache

Although it is fun optimizing inner loops there are other important factors that one should look at. With the Pentium processor the cache aspects are very important. Maybe more important than the speed of the inner loop. Don't know how long this is true though as newer processors seems to get bigger and bigger caches that probably will become smarter also.

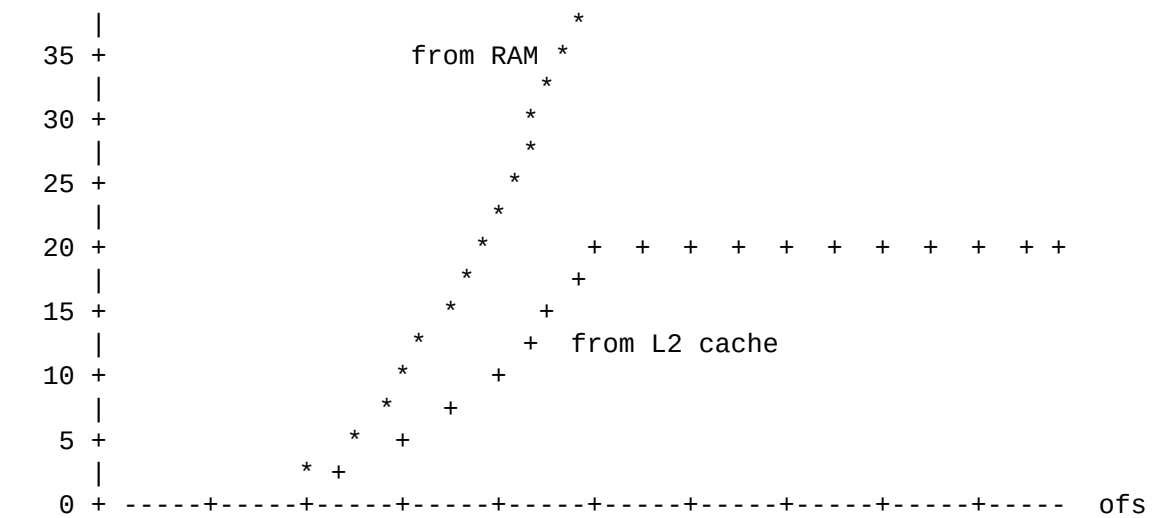
The general idea of the cache is:
When the CPU has decoded an instruction that needs to get a variable from memory, the CPU first checks the cache to see if the variable is already in the cache. If it is there the CPU reads the variable from the cache. This is called a cache hit. If the variable is not in the cache the CPU first has to wait for the data to be read from RAM (or the secondary cache, L2) into the cache and first after that get the variable from the cache. The cache always loads a full cacheline at a time so this will take a few clock ticks. A cacheline is 16 byte on a 486 and 32 byte on Pentium. The advantage of this is when reading byte after byte from the memory, the data will most of the time already be loaded into the cache because we have accessed the same cacheline just before. Also a cacheline is always aligned on 16 byte on the 486 and on 32 byte on the Pentium.

I did a few tests on my system (Pentium 120 MHz, L1 cache 8+8k, L2 cache 256k) using the time stamp counter to check the actual time for loading a cacheline. In the first test I flushed the L2 cache so that each cacheline must be read all the way from RAM. This was done by allocating a 256k memory chunk and read each byte of that first. This would cause the memory I did the test on to be pushed out of the L2 cache. The testloop looked like this:

```
    mov  ecx, 1000
  next:
    mov  al, [esi]
    add  esi, ofs
    dec  ecx
    jnz  next
```

The overhead of the loop was first timed by replacing the "mov al, [esi]" by "mov al, cl". The loop ran at exactly 2 clock tick per turn. The "ofs" value was replaced for each run with 1, 2, 4, 8, 16, 32, 64, ... In the second test I first forced the L2 cache to load the memory by reading each byte of a 128k memory chunk and then run the testloop on the same memory. Here are the results of both tests:

```
 clock ticks
                                                          * *
    |                                          *  *  *  *  *
 40 +                           *  *  *  *
```



So this tells me that it takes 40-45 clock ticks minimum to load a cacheline all the way from RAM and exactly 18 clock ticks from the L2 cache. When "ofs" was 1 the "mov al, [esi]" ran at 2.0 ticks when loading from RAM and 1.1 ticks from the L2 cache. 0.5+40/32=1.75 and 0.5+18/32=1.06 so this makes sense.

This is pretty scary! 18 clock ticks to load a cacheline from the L2 cache. 18 clock ticks minimum for the inner loops if we assume that a cacheline must be filled for each byte read. Ouch!

So in the case of a texture mapper where we might be reading texels in a vertical line in the bitmap, the inner loop will be accessing pixels that are >256 bytes apart. The CPU will then be busy filling cachelines for each texel. A 64k bitmap won't fit inside a 8k cache, you know. So what can we do? Well, we can wait on Intel to invent bigger caches or we might consider storing our bitmaps some other, more cache friendly way.

I got an interesting tip from Otto Chrons on channel #coders the other day about this. He said that one should store the bitmap as a set of tiles, say 8 x 8 pixels instead of the usual 256 x 256 pixel. This makes perfect sense. It would mean that a small part of the bitmap (8 x 4 pixel) would fit in the same 32 byte cacheline. This way, new cachelines don't need to be loaded that often when reading pixels in a vertical line in the bitmap.

The following was suggested in a mail to me by Dare Manojlovic:

> If you are saving bitmap as a set of tiles (8*4) the inner loop wouldn't have to be more complicated (this is my opinion - not yet tested).
>
> For example, let's say that we have u&v texture coordinates, we only have to reorder bits to get the correct address (before the inner loop):
>
> Normally for a bitmap of 256*256 the texel address would look like:
>
> ```
> EAX                          AH           AL
>  oooo oooo    oooo oooo    oooo oooo     oooo oooo
>                           v coordinate   u coordinate
> ```
>
> And now:
>
> ```
> EAX                          AH           AL
>  oooo oooo    oooo oooo    oooo oooo    oooo oooo
>   v(other 6 bits)  u(other 5 bits) v(lower 2 bits) u(lower 3 bits)
> ```
>
> Adding a constant value,that is also converted, in the loop shouldn't be a problem.
>
> Now, as I understand cache loading procedure,it always loads 32 bytes of data (Pentium), so the whole bitmap tile of (8*4 pixels) will be in cache. Of course bitmap tile must be 32 bytes aligned.

This would also work faster on 486 where cache is loaded with 16 bytes.

There is a small problem to the above method. We can't just add a constant value to a number in this format (even if they both are converted). This is because there is a gap between the bits. We must make the bits jump over the gap to make the add correct. There is a simple solution to this problem though. Just fill the gap with 1:s before adding the constant value. This will cause the bit to jump over the gap. Filling the gap is done with a bitwise OR instruction.

Converting u and v (16:16 bit) to this format can be done with the following code:

```
int uc = (u & 0x0007ffff) | ((u<<2) & 0xffe00000);
int vc = (v & 0x0003ffff) | ((v<<5) & 0xff800000);


; eax = u  --------wwwwwwwwffffffffffffffff  (w=whole, f=fractional)
; ebx = v  --------wwwwwwwwffffffffffffffff
; ecx = scratch register
mov   ecx, eax
shl   eax, 2
and   ecx, 00000000000001111111111111111111b
and   eax, 11111111110000000000000000000000b
or    eax, ecx
mov   ecx, ebx
shl   ebx, 5
and   ecx, 00000000000001111111111111111111b
and   ebx, 11111111100000000000000000000000b
or    ebx, ecx
; eax = u  ------wwwww--wwwffffffffffffffff
; ebx = v  ---wwwwww-----wwffffffffffffffff
```

Adding dudx and dvdx to u and v in this format can be done with the following code (all variables are in the converterd format):

```
uc = (uc | 0x00180000) + dudx;
vc = (vc | 0x007c0000) + dvdx;


; eax = u  ------wwwww--wwwffffffffffffffff
; ebx = v  ---wwwwww-----wwffffffffffffffff
; dudx, dvdx = 16:16 bit converted to this format
or    eax, 00000000001100000000000000000000b    ; fill the bit-gap in u
or    ebx, 00000000011111000000000000000000b    ; fill the bit-gap in v
add   eax, [dudx]
add   ebx, [dvdx]
```

In a mail sent to me, Russel Simmons preresented the following method to reorder the bits to acheive a simpler inner loop by eliminating a bit-gap:

> In one post, someone suggested a bit structure to find the corect
> position in your tiled texture given u and v. He suggested something
> like:
>
> high bits of v | high bits of u | low bits of v | low bits of u
>
> This way the high bits of u and v determine which tile our texel is in,
> and the low bits of u and v determine where in our tile the texel is.
> If we store our tiles in a different manner, we can simplify this to:
>
> high bits of u | high bits of v | low bits of v | low bits of u
>
> which is in other words:

high bits of u | all bits of v | low bits of u

In order to facilitate this, instead of storing our tiles in this order:

```
-------------
| 0| 1| 2| 3| ...  (here i am showing the upper 4x4 tiles of a 256x256
-------------       texture store in 8x8 tiles)
|32|33|34|35| ...
-------------       Original Method
|64|65|66|67| ...
-------------
|96|97|98|99| ...
-------------
|  |  |  |  |
```

store them in this order:

```
-------------
| 0|32|64|96| ...  (here i am showing the upper 4x4 tiles of a 256x256
-------------       texture store in 8x8 tiles)
| 1|33|65|97| ...
-------------       New Method, in order to acheive a simpler inner loop
| 2|34|66|98| ...
-------------
| 3|35|67|99| ...
-------------
|  |  |  |  |
```

Also, if we are storing our bitmap in a tiled fashion, then it would greatly improve our cache performance if we can back and forth across scan lines.. in other words alternate the direction we scan across lines. Say we have just scanned forward across one scan line. If we start backwards across the next scan line, we are likely to be pulling texels from the same tiles as we were at the end of the previous scan line.

The last part about alternating the drawing direction is definitely something to try out!

I was hoping I would be able to present some code here that uses all these techniques and 16:16 bit interpolation in a slick inner loop but due to lack of time and the fact that I'm fed up with this document, I leave this to you.

21. The code cache

The cool thing about Pentiums is that it can execute two instructions in parallel. This is called instruction pairing. But there is a lot of rules that must be fulfilled for the pairing to take place. One rule is that both instructions must already be in the code cache. This means that the first time trough a inner loop, no instructions will pair. There is one exception to this rule. If the first instruction is a 1 byte instruction, e.g. inc eax, and the other is a simple instruction, then they will pair the first time.

If by chance our inner loop happens to be in the code cache, by modifying an instruction in the inner loop (selfmodifying code) the cacheline where we did the modification will be marked as not up to date. So that cacheline must be loaded into the cache again before we can execute the inner loop again. Loading of code cachelines seems to be exceptionally slow also. In other words, we have found yet another source of delay.

So to have a completely unrolled loop that almost fills up the whole code cache and also is selfmodifying is a pretty bad idea on the Pentium. On the other hand, we are not modifying the loop for each scanline so chances are

that parts of it will be in the code cache from drawing the previous
scanline.

22. Some pairing rules

As mentioned above, the Pentium can execute two instructions in parallel.
This is possible because the CPU has dual integer pipelines, they are
called the U and V pipelines. The Pentium has a so called superscalar
architecture. The U pipeline is fully equipped and can execute all integer
instructions. The V pipeline on the other hand is a bit crippled and can only
execute simple, RISC type instructions.

Simple instructions are:

```
 mov, inc, dec, add, adc, sub, sbb,
 and, or, xor, cmp, test, push, pop,
 lea, jmp, call, jcc, nop, sar, sal,
 shl, shr, rol, ror, (rcl), (rcr)
```

(What I've heard there are different opinions on if the shift/rotate
instructions are pairable or not. The book I have here states that these
instructions are pairable but can only execute in the U pipeline)

The first pairing rule is that both instructions must be simple instructions.
Also, no segment registers can be involved in the instructions.

Another rule is that the two instructions must be completely independent of
each other. Also they must not write to the same destination register/memory.
They can read from the same register though. Here are some examples:

```
    add   ecx, eax      ; store result in ecx
    add   edx, ecx      ; get result from ecx. No pairing!

    mov   ecx, eax
    mov   edx, ecx      ; No pairing!

    mov   al, bh        ; al and ah is in the same register
    mov   ah, ch        ; No pairing!


    mov   ecx, eax      ; read from the same register
    mov   edx, eax      ; Pairs ok.

    mov   ecx, eax      ; note eax in this example
    add   eax, edx      ; Pairs ok.
```

There are two exception to this rule. Namely the flag register and the stack
pointer. Intel has been kind enough to optimize these.

```
    dec   ecx           ; modifies the flag register
    jnz   @@inner       ; Pairs ok.

    push  eax           ; both instructions are accessing esp
    push  ebx           ; Pairs ok.
```

So for example the loop we used to calculate the lookup table with, all
instructions are simple and not dependent on the previous one. The 8
instructions should execute in 4 clock ticks.

```
 @@mklookup:
    mov   eax, ecx
    add   ecx, edi      ; Pairs ok.
    mov   al, bh
```

```
    add   ebx, esi      ; Pairs ok.
    mov   [edx], eax
    add   edx, 4        ; Pairs ok.
    dec   ebp
    jnz   @@mklookup    ; Pairs ok.
```

23. Pipeline delays

There are a whole bunch of these that will delay the pipelines:

- data cache memory bank conflict
- address generation interlock, AGI
- prefix byte delay
- sequencing delay

I personally think that the AGI is most important to consider in the case of tight inner loops. Because that is what's happening in a inner loop, where we are calculating an address and need it right away to access some data. There will be a AGI delay if a register used in a effective address calculation is modified in the previous clock cycle. So if we have our instructions nicely pairing we might have to put 3 instructions in between to avoid the AGI delay.

```
    add   esi, ebx      ; Move the array pointer.
    mov   eax, [esi+8]  ; AGI delay. You just modified esi.


    add   esi, ebx      ; Move the array pointer.
    add   ebx, ecx      ; Do something useful here
    inc   edi           ;  "
    add   ebp, 4        ;  "
    mov   eax, [esi+8]  ; Now it's OK to access the data. No AGI delay.
```

If you don't have any useful instructions to fill out the gap with you could try to swap the two instructions so that you access the data first and then modify the index register.

```
    mov   eax, [esi+8]
    add   esi, ebx      ; Pairs ok. No AGI delay.
```

There are a lot more rules one must follow so I suggest you buy a good book on the subject. I don't know of any free info about this on the net as of this writing. Maybe you'll find something at Intel's www-site (http://www.intel.com). Anyway, a book that got me started was: "Pentium Processor Optimization Tools" by Michael L. Schmit ISBN 0-12-627230-1 This book has a few minor errors and some of the explanations are a bit cryptic but it is a good starting point. The way to really learn is to get the basics from e.g. a book and then time actual code to see what is faster and what's not.

24. The time stamp counter

The Pentium has a built in 64 bit counter called the Time Stamp Counter that is incremented by 1 for each clock tick. To read the counter you use the semi-undocumented instruction RDTSC (db 0fh,31h). This will load the low 32 bit of the counter into EAX and the high 32 bit into EDX. Perfect for timing code!

```
    ; First time the overhead of doing the RDTSC instruction
```

```
    db      0fh,31h         ; hex opcode for RDTSC
    mov     ebx, eax        ; save low 32 bit in ebx
    db      0fh,31h
    sub     eax, ebx        ; overhead = end - start
    mov     [oh_low], eax

    ; Now do the actual timing
    db      0fh,31h
    mov     [co_low], eax
    mov     [co_hi], edx

    ; Run some inner loop here of whatever you want to time

    db      0fh,31h
    sub     eax, [co_low]   ; ticks = end - start
    sbb     edx, [co_hi]
    sub     eax, [oh_low]   ; subtract overhead
    sbb     edx, 0
    ; Number of clock ticks is now in  edx:eax
```

You'll notice that I first time the overhead of doing the RDTSC instruction. This might be a bit overkill but it's no harm in doing it. Note also that I ignore the high 32 bit. The overhead should not be more than 2^32 clock ticks anyway. The RDTSC can be a privileged instruction under some extenders (?) but still be available (under the control of the extender) so there might actually be a overhead to time.

You can usually ignore the high 32 bit. Using only the low 32 bit will allow a maximum of 2^32 clock ticks which is 35 seconds on a Pentium 120 MHz.

When you are timing your code e.g. when you have done some optimizations on your texture mapper, don't time just one triangle over and over. Time how long it takes to draw a complete object with hundreds (thousands) of triangles. Then you'll know if that optimization made any difference.

25. Branch prediction

The Pentium has some sort of lookup table called the Branch Target Buffer (BTB) in which it stores the last 256 branches. With this it tries to determine the destination for each jump or call. This is done by keeping a history of whether a jump was taken or not the last time it was executed. If the prediction is correct then a conditional jump takes only 1 clock tick to execute.

Because the history mechanism only remembers the last time the jump was executed, the prediction will always fail if we jump different each time. There is a 4-5 clock tick delay if the prediction fails.

The branch prediction takes place in the second stage of the instruction pipeline and predicts if whether a branch will be taken or not and its destination. Then it starts filling the other instruction prefetch queue with instructions from the branch destination. If the prediction was wrong, then both prefetch queue must be flushed and prefetching restarted.

So to avoid this delay you should strive to use simple loops that always takes the jump or always not takes the jump. Not like the following that jumps different depending on the carry flag.

```
    jmp   @@inner
  @@extra:
    ....                ; Do something extra when we get carry overflow
    dec   ebp
```

```
    jz    @@done
  @@inner:
    ....                ; Do something useful here
    add   eax, ebx
    jc    @@extra       ; Jump on carry overflow
    dec   ebp
    jnz   @@inner
  @@done:
```

In this loop it's the 'jc @@extra' instruction that will mess up the branch prediction. Sometimes the jump will be taken and sometimes not. The typical way of doing masking with compares and jumps has this problem also.

26. Reference

Most of the Pentium specific information on optimization was found in the book: "Pentium Processor Optimization Tools" by Michael L. Schmit
ISBN 0-12-627230-1

27. Where to go from here

When you have implemented your texture mapper you automatically also have Phong shading and environment mapping. It's only a matter of making a suitable bitmap and to use the normal vectors at each triangle vertex to get the u and v values.

From there the step is not far from combining Phong shading and texture mapping. And then adding bumps to all this. The only difficult part is that you need to interpolate 4 variables in the inner loop when you do Phong-texture, environment-bump or Phong-texture-bump and still have registers left for pointers and loop counter. These shadings can't really be called "fast" as the inner loops will become pretty ugly. They can definitely be called real time though.

28. Credits and greetings

Juan Carlos Arevalo Baeza (JCAB/Iguana-VangeliSTeam) <jarevalo@daimi.aau.dk>
Wilco Dijkstra <wdijkstr@hzsbg01.att.com>
Kevin Baca <kbaca@skygames.com>
Sean L. Palmer <sean@delta.com>
Tiziano Sardone <tiz@mail.skylink.it>
Mark Pursey <nerner@world.net>
Dare Manojlovic <tangor@celje.eunet.si>
Russel Simmons (Armitage/Beyond) <resimmon@uiuc.edu>
Aatu Koskensilta (Zaphod.B) <zaphod@sci.fi>
Otto Chrons (OCoke) (a legend)
Nix/Logic Design (a cool coder)
Phil Carmody (FatPhil) (The optimizing guru, why all this silence?)
Jmagic/Complex (another legend)
MacFeenix (you are young)
BX (keep on coding)
thefear (a cool swede)
John Gronvall (MIPS R8000 rules!)
LoZEr (when will PacMan for Linux be out?)
Addict, Damac, Dice, Doom, Swallow, Wode / Doomsday (a bunch of finns ;)

When I started out writing this document I didn't know half of what I now

know about texture mapping. I've learned a lot and this is much because of those 12 first persons in the credits and greetings list. Thanks a lot for the help. I hope that the readers of this document also will learn something.

If you truly find this document useful, you could consider giving me a small greeting in your production. That would be cool.

<EOF>