REALTIME 3-D TECHNIQUES

Bernhard Tschirren

CONTACT INFO

Author: Bernhard Tschirren
E-mail: bernie-t@geocities.com
WWW: http://www.geocities.com/SiliconValley/7259/
Address: 25 Regency Drive
Thornlie WA 6108
Australia

Revisions: 0.2 20/09/1998 Complete document renovation.
0.1 25/05/1998 Converted to Word97 and added diagrams.

TABLE OF CONTENTS

PRE-REQUISITES & AIM OF THIS TEX

This text is not a tutorial on vectors, matrices and 3-D systems. You should already be familiar with vectors, matrices, etc. Linear algebra is usually taught in the first year of university (in a CS course, not knitting with needles 101!). For example, I never explain what a normal is, or how to calculate the plane equation given three points. These are things that should be second nature to you. If you don't know what I'm talking about, look it up in a book. I'm also open to answer your questions via e-mail (find my address at the top of this document). But please don't be afraid of the maths involved!

In this tutorial, I won't go into the programming details. Nevertheless, it would be HUGELY beneficial to know at least one programming language. I will probably release a follow-up to this document that delves more into the programming aspects. With it, I will include my full 3-D engine source in C (or maybe Pascal). Anyway, that is still some time away...

This document merely is intended to TWIST your point of view to that required to build a realtime 3-engine. It should NOT tech new concepts, only point you in a new direction. I want you to think "Oh yeah, I never thought of it that way!" Then again, that is probably what teaching is all about...

OBJECTS

The most important first step when designing a 3-D engine is to look around you. The aim of a 3-D engine is to simulate the real world inside your computer. The best place to get hints on how to implement your 3-D engine is to look at the real world.

So, let's look around. On my left, I see a small table. On it are a couple of disk boxes, lots of paper, and one of those Microsoft licensing agreements. When I look out the window I can see a swimming pool with a few leaves floating in it. I can see trees, rocks, and a folding chair.

Hmm, lets think about all the things I just described. Wait a minute - Things! I mentioned lots of different objects, each one looking, sounding and feeling different. So here is our first clue: The 3-D engine must work with objects. We will make the Object the most important aspect of our engine, since everything can be classified as an object.

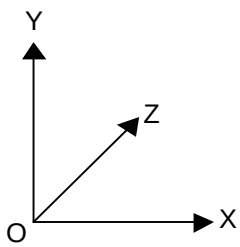
Every object has some things in common - they all have a shape, a texture, a position and direction in our world. The last two things might seem a little strange - a position and direction? Let's get back to the real world. Remember the disk boxes that are sitting on my table? Each one is positioned in a different place, and each one is facing a different direction.

Let's just focus on the position and direction properties of an object. They are special properties because they always change. The slight breeze is causing the leaves in my pool to slowly move and turn.

POSITION (AXES)

NO - I'm not talking about those sharp, heavy objects used to chop trees and frequently featured in horror movies. I'm talking about how we define 3-D space inside a computer. There are many different ways of doing this, and unfortunately there is no standard method - every textbook chooses a different implementation! Essentially, the only differences are the axis labels and directions. Nevertheless, PLEASE DO NOT MIX AND MATCH BETWEEN DIFFERENT AXIS SYSTEMS!

Throughout this text I will use my favourite system: The X-Axis points to the right, the Y-Axis points up and the Z-Axis points into the screen. A point is defined by (X,Y,Z), that is X units to the right, Y units up and Z units back. (0,0,0) is called the Origin (O) - this is where the three axes intersect.



DIRECTION (ROTATION)

Position is very easy to describe - its simply (,Y,Z). But how do we represent direction? One method is to describe it using a unit vector. Unfortunately, a unit vector does not make it very easy to change th direction of an object. A very simple and effective method is to use the XYZ angle system (Euler angles). This is a flawed system but it works very well in a first-person-perspective type 3-D engine. If the player is able to turn upside down it tends to reverse the controls! Also, it is possible for the rotations to cancel each other out (gimbal lock), but we shall ignore that for now…

I prefer to use more meaningful names than X, Y and Z, so I will call them Yaw, Pitch and Roll. Yaw basically describes the rotation about the Y-Axis, and it corresponds to looking left and right. Pitch describes looking up and down, and mathematically it is rotation about the X-Axis. Rotation about the Z-Axis is called Roll, and it simply corresponds to tilting our head sideways.

Because we are in 3 dimensional space, we require a 3x3 matrix to represent any rotation about the origin (0,0,0). Looking at any textbook on computer graphics (and a bit of common sense) we can se that rotations can be achieved by the following three matrices. We can also combine these three matrices into one rotation matrix, but we will leave that until later.

Rotation about X-Axis
$$= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(pitch) & -\sin(pitch) \\ 0 & \sin(pitch) & \cos(pitch) \end{bmatrix}$$

Rotation about Y-Axis
$$= \begin{bmatrix} \cos(yaw) & 0 & -\sin(yaw) \\ 0 & 1 & 0 \\ \sin(yaw) & 0 & \cos(yaw) \end{bmatrix}$$

Rotation about Z-Axis
$$= \begin{bmatrix} \cos(roll) & -\sin(roll) & 0 \\ \sin(roll) & \cos(roll) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

NOTE: These matrices are dependent on the coordinate system that you use. Every textbook and all code seem to use a different axis system - there is no STANDARD! Please do not mix and match between different axis systems because you will only get a major headache!

STORING OBJECTS

How do we store object data in our 3-D engine? How do we represent the shape of the leaf floating in my pool? Don't forget the leaf must be able to move and rotate freely in our simulated world.

Lets first address the problem of storing the shape of an object. A simple and effective approach is to describe the shape using a wire-frame mesh. It is called a wire-frame mesh because the object looks as if it was modelled with wire. This mesh is stretched and curved around so that it resembles our leaf.

The mesh is made up of lots of flat surfaces called "faces". Each face shares its vertices with the adjoining faces. So, when storing an object, we must maintain two lists. One is a list of vertices and the other is a list of faces. Each face also has a list indexes (pointers) to its vertices. In my engine, I prefer to call the faces polygons, because a face must have at least three sides (triangle), must be convex, and must be planar (flat).

To allow our mesh to rotate freely, we must consider where we will store its vertices. Our rotation matrices rotate any vertex around the origin, so our object will always rotate around (0,0,0). Usually, the object will be stored so that it's centred on (0,0,0), but there are cases where it may be desirable to have the object rotate about some other arbitrary point - its pivot point.

Each object carries around a position vector and directional information (Yaw, Pitch and Roll). This information will be used to rotate the object and position it at its proper location.

SPACES

Now, here is where it becomes a little more complicated! Remember the leaf floating on my pool? If it has a position of (20,20,20), where exactly is that? You can't answer that question without some sort of reference point. One reference point would be ME. I'm standing at (10,20,30), so the object is (10,0 ,-10) units away from me. But as soon as I move the definition changes. We need a static reference point, one that never moves. We call this reference point the World, and for simplicity we say that the world is centred at (0,0,0). So, the leaf is positioned (20,20,20) units away from the centre of the world. Another way to say this is the leaf is located at (20,20,20) in World-Space. The world itself is located at (0,0,0) in World-Space.

Hmm, lets go one step further and say that the world is located (-20 ,-20,-20) in Leaf-Space, and the leaf itself is located at (0,0,0) in Leaf-Space! This may sound stupid, but it's very important in a 3-D engine.

That takes care of position, but what do we do with direction… exactly the same thing! The leaf has a yaw of 20 degrees, and zero pitch and roll. I shall express this as <20,0,0> in World-Space. In Leaf-Space, the world has a direction of <-20,0,0> and the leaf itself has a direction of <0,0,0>. Pretty simple, eh?

In Leaf-Space, the leaf itself NEVER moves - it is always stored at (0,0,0)! This is important, because it describes the space where our object mesh is stored. So, now all we need is a method of converting th leaf vertices from Leaf-Space into World-Space...

TRANSFORMATIONS

To move something from one space into another is called an Affine Transformation. A transformation is composed of two operations - Rotation and Translation. First we point the object in the right direction, then we move it into place.

There are two ways I write a transformation matrix-vector pair. One is **(R, t)** where "**R**" is the rotation matrix, and "**t**" is the translation vector. I always write vectors in lower case letters, and matrices in capitals. The other method shows the source and target spaces, eg: Object→World.

Let's get mathematical:

$$\begin{aligned} \mathbf{p'} &= \mathbf{Rp} + \mathbf{t} \\ &= \mathbf{(R,\ t)} \end{aligned}$$

where
p = The original point in Object-Space
p' = The transformed point in World-Space
R = The rotation matrix
t = The translation vector

INVERSE TRANSFORMATIONS

OK, now that we know how to transform a point from Object-Space into World-Space, it might be handy to transform a point from World-Space back into Object-Space (World →Object). This is accomplished by using the inverse transformation. Now, mathematically:

$$\begin{aligned} \mathbf{Rp} + \mathbf{t} &= \mathbf{p'} \\ \mathbf{Rp} &= \mathbf{p'} - \mathbf{t} \\ \mathbf{p} &= \mathbf{R'(p'} - \mathbf{t)} \\ &= \mathbf{R'p} - \mathbf{R't} \\ &= \mathbf{(R',} - \mathbf{R't)} \end{aligned}$$

where
p' = Original point in World-Space
p = Inversely transformed point back in Object-Space
R = Rotation matrix (Object→World)
t = Translation vector (Object→World)
R' = Rotation matrix (World→Object)

So, the inverse of **(R, t)** is **(R', -R't)**. The only problem comes with finding **R'**, which is the inverse of **R**. Finding the inverse of a matrix is usually NOT a very trivial operation, but because our matrices are ortho-normal, the inverse is simply the transpose! An ortho-normal matrix is any matrix where the column vectors are orthogonal and normalised.

COMBINING TRANSFORMATIONS

Ok, we have the leaf mesh stored in Leaf-Space, and we want to see the leaf from the point of view of a camera (the viewer's position and viewing direction). The camera is just like any other object, and we can express its transformation as Camera→World. If we calculate the inverse of this we get World→Camera. So, all we need to do is:

Leaf→Camera = Leaf→World + World→Camera
= Leaf→World + Inverse(Camera→World)

Now, if there were only a way to combine these two transformations into one...

p' = **Ap** + **s**
p'' = **Bp'** + **t**
= **B(Ap** + **s)** + **t**
= **BAp** + **Bs** + **t**
= **(BA, Bs** + **t)**

where **p** = Point to transform
p' = Point after first transformation
p'' = Point after second transformation
(A, s) = First transformation (Leaf→World)
(B, t) = Second transformation (Leaf→Camera)

Very cool! We can now store all our objects in Object-Space, and transform them into Camera-Space with only one transformation. This allows arbitrary object AND arbitrary camera locations!

HIERARCHICAL TRANSFORMATIONS

Let us get back to those disk boxes on my desk. If I lift my desk up into the air, everything on top of the desk also moves up, including those disk boxes. How do we EFFICIENTLY implement this?

Simple, we just combine the disk boxes transformation matrix-vector with that of the table. Watch this:

Disks→Camera = Disks→Table + Table→Camera
= Disks→Table + Table→World + World→Camera

As you can see in the above equation, we need Disks→Table. This just means that we refer to the disks position and direction RELATIVE to that of the table. The disks are "child objects" of the table.

Let's get EVEN more fancy! Remember, our 3x3 rotation matrix always rotates around (0,0,0). If we store the object mesh slightly above (0,0,0) in Object-Space, it will still rotate around (0,0,0). What are the implications of this?

Consider a model of a human body. The torso is stored so that its pivot point (0,0,0) is roughly in the centre of the model. The upper arm is stored so that its pivot point is at the end that connects to the torso.

Position the arm so that it connects to the torso at the shoulders. Similarly, store the forearm so that it pivots around the end, and position it so that it connects to the upper arm. Hopefully you get the idea... After a bit of work you will have a working human body, with parts that move realistically. If you rotate the upper arm, the forearm and hand and all fingers rotate as well. If the torso moves up (jumps), all the connected body parts jump up as well!

INVERSE KINEMATICS

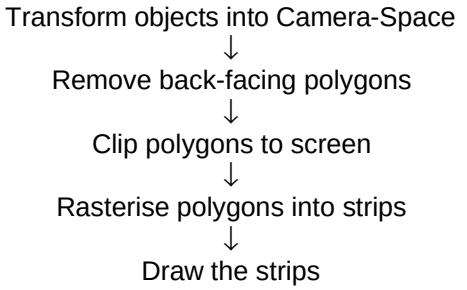
This is the final step required to provide realistic body movements. Inverse Kinematics represents the latest technology available - it is still being researched. As of yet, there is no realtime engine supporting inverse kinematics, but several are under construction, including my own!

Kinematics is the process of finding out position of objects given velocity, acceleration, etc. Inverse Kinematics does the opposite. It calculates the forces required to move an object into the desired position. So, all you need to do is move the foot up and the knee and hip joints respond by bending realistically.

Unfortunately, I don't have enough accurate information available to delve into this subject. But please, stick around...

RENDERING PIPELINE

The rendering pipeline basically describes all the steps to display the final rendered scene. The rendering pipeline is composed of many stages. Each stage receives its input from the previous stage, processes it, and then feeds the output to the next stage. Below is a diagram of a traditional rendering pipeline.



This is an example of an extremely inefficient pipeline. In the subsequent sections of this text I shall explain why, and hopefully present a better pipeline.

BSP TREES

Probably the most important part of a 3-D engine is to efficiently remove the polygons that are NOT visible. Usually, only a small portion of the entire scene is visible at one time. To gain as much speed as possible, it is imperative that these culling operations are performed as early as possible in our rendering pipeline. We do not want to clip, transform or project anything that is not visible.

An efficient method of removing hidden faces is to use a Binary Space Partition (BSP) tree. BSP trees provide a method of quickly determining which faces are visible without any runtime processing. The entire scene is pre-processed and stored in a big hierarchy of polygons. The renderer then simply steps through the tree to draw only the visible polygons in a back-to-front order. BSP trees sound almost to perfect to be true! Unfortunately, they have two major drawbacks.

Creating an efficient BSP tree is a difficult and slow process. It is practically impossible to generate an optimal tree for a large scene with many polygons. Scene creation is therefore hampered because a new tree has to be calculated before you can check out your changes!

The second major drawback of BSP trees is that the scene cannot be changed once the tree is created. Walls cannot move around and entire rooms cannot remodel themselves right in front of your eyes. This significantly reduces the realism of the engine because the environment cannot react to you.

PORTALS

Portal technology is very new, and its promising to solve all the problems associated with BSP trees. A portal engine requires NO pre-processing and the environment can be completely dynamic!

The idea of portals is very simple - divide the scene into cells (rooms) and connect them via portals (doors and windows). You start off by drawing the cell that you are standing in, then when you hit a portal polygon you draw the cell on the other side. When you finished drawing the other cell, you revert back to the current cell and continue drawing. This is just a simple recursive procedure. Obviously, this method only works with indoor scenes, whereas BSP trees can easily accommodate outdoor environments.

With just a tiny bit more work, you can even have tinted-glass portals and perfect mirror-portals. I personally like the idea of realtime mirrors :) But, I wont go into that yet...

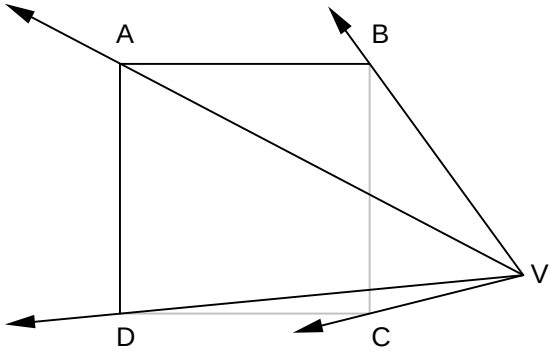
A problem with portal polygons is that the cell on the other side must be clipped to the portal polygon outline. Everything on the other side of the window can only be seen THROUGH the window. Luckily, this problem is easily solved using frustum clipping.

VIEW FRUSTUM

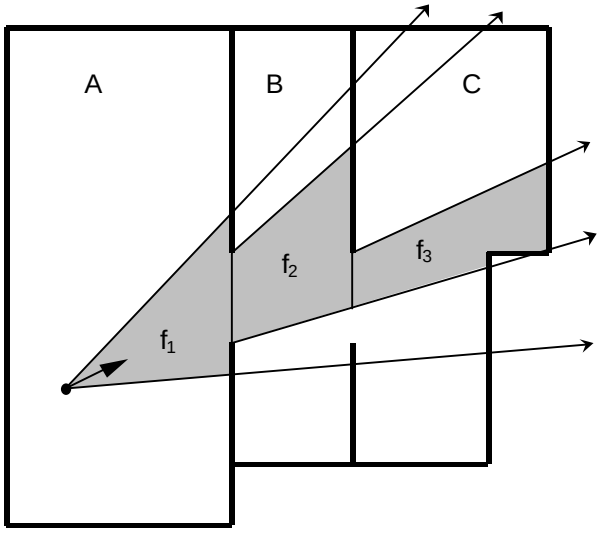
If you take a square based pyramid and cut its tip off, you are left with a frustum. This frustum describes our view volume – everything inside it is visible, everything outside is not. Positioned at the apex of the pyramid is the camera and it is looking down into the frustum. The flat top of the frustum is our portal polygon.

A view frustum is defined by a list of planes. Each plane describes one side of the frustum. We need another plane to cap the top, and if we want, we can also add another plane to cap the base of the pyramid. This limits the viewing distance of the camera.

To define our view frustum, we simply "cast" it from our camera position through the portal polygon. Let me look out my window again - that leaf has now reached the end the pool! Let's say that the window is the portal polygon and I am the camera looking out the window. Remember , to calculate the plane equation we need at least 3 points on the plane. The top-left corner of the portal (A), the top-right corner of the portal (B), and the position of myself (V) define the top plane. The bottom-left corner of the portal (D), the top-left corner of the portal (A) and myself (V) define the left plane... and so on until we have th four planes defining the shape of the frustum. The plane that caps the top of the frustum is simply the window itself. Sometimes, this window is open and objects can move freely through it. In this case, we don't add the window itself to the view frustum - we have an open frustum.



Viewer = V
Portal = (A,B,C,D)
Top Plane = (A,B,V)
Left Plane = (D,A,V)
Bottom Plane = (C,D,V)
Right Plane = (B,C,V)



Here is the top view of a typical scene divided into three cells (A, B and C). The doorways joining these cells are the portals. The shaded region denotes the final visible area inside the view frustums.

There are three view frustums (f_1, f_2 and f_3) extending out to infinity. The first frustum (f_1) starts at the camera position and maps out the field of vision. The second frustum (f_2) starts at the portal polygon joining cells A and B. The last frustum (f_3) has been cast through the portal polygon joining cells B and C. You may notice that this portal polygon was clipped to f_2 when cell B was being drawn.

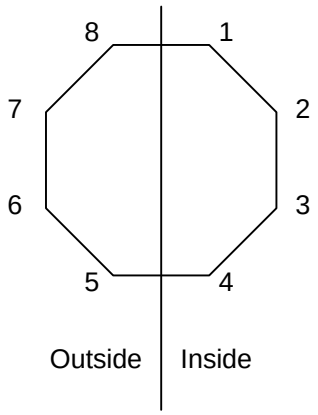
3-D CLIPPING

Clipping is a step that traditionally used to be done by the polygon drawer (rasteriser). However, clipping is the type of operation that removes a lot of polygons so it would be very beneficial to do it before transforming the object into Camera-Space. But how can we clip something that isn't in Camera-Space - there is no way of knowing whether it will end up on the screen or not!

Actually there is a way, and it's very simple! Instead of transforming the object into Camera-Space, we transform the frustum into Object-Space! It is much faster transforming a view frustum than transforming an entire object. Once we have the frustum in Object-Space, we simply clip all polygons against each frustum plane in turn.

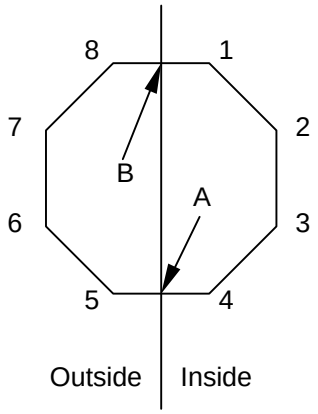
Clipping a polygon against a plane is relatively simple - but quite expensive in terms of CPU time. To clip, select the polygon's two edges where one vertex is inside the plane, and the other is outside the plane, find the point of intersection with the plane, and chop! See diagrams below:

(1)



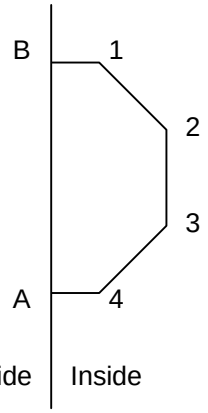
1 = Visible
2 = Visible
3 = Visible
4 = Visible
5 = Invisible
6 = Invisible
7 = Invisible
8 = Invisible

(2)



A = Intersection
B = Intersection

(3)



New Polygon = (1,2,3,4, ,B)

As you can see, we need a method of determining which side of the plane a vertex resides on. Luckily, it's as easy as feeding the vertex coordinate into the plane equation. When the following equation is TRUE, the vertex is in front of the plane (the side which the normal points out of, which is also the inside of the frustum), otherwise the vertex is behind the plane (outside the frustum).

$\mathbf{v} \bullet \mathbf{n} \geq d$ where $\mathbf{v}$ = Vertex
$\mathbf{n}$ = Plane normal
d = Plane constant

So, now we know how to accomplish step (1) of polygon clipping (according to the diagrams). The next step is to find the point of intersection of the polygon edge with the plane. An edge is simply a line from one vertex ($\mathbf{v}_1$) to another ($\mathbf{v}_2$). Somewhere along this line (at time t) we have the point of intersection (**i**).

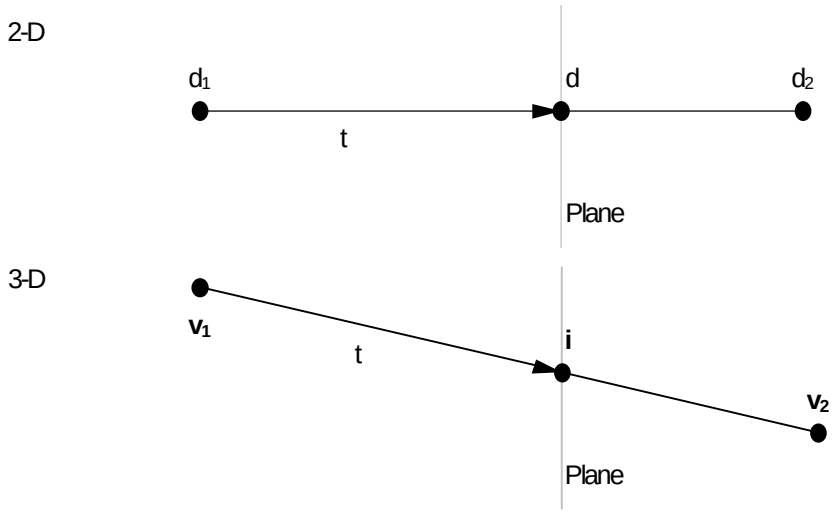
$$\mathbf{i} \bullet \mathbf{n} = d$$
$$\mathbf{v}_1 \bullet \mathbf{n} = d_1$$
$$\mathbf{v}_2 \bullet \mathbf{n} = d_2$$

where $\mathbf{i}$ = Point on plane (aka: the point of intersection)
$\mathbf{n}$ = Plane normal
d = Plane constant
d_1 = Plane constant for $\mathbf{v}_1$
d_2 = Plane constant for $\mathbf{v}_2$
$\mathbf{v}_1$ = Vertex 1 (front of plane)
$\mathbf{v}_2$ = Vertex 2 (back of plane)
t = Distance to d along line ($d_2 - d_1$)

$$d_1 + t(d_2 - d_1) = d \quad (1)$$
$$t(d_2 - d_1) = d - d_1$$
$$t = (d - d_1) \div (d_2 - d_1)$$

$$\mathbf{i} = \mathbf{v}_1 + t(\mathbf{v}_2 - \mathbf{v}_1) \quad (2)$$

Step (1) may be a bit hard to understand. We formed the equation of a 2-D line using d_1 and d_2 as the end points. We know that d, the point of intersection, lies somewhere between d_1 and d_2. Therefore, point d lies some t units away from point $_1$. Once we know t, we can substitute it into equation (2), which will give us the intersection point in 3-D. Here is a diagram that may explain things better:



BACKFACE CULLING

Backface culling removes all the polygons that are facing away from the viewer. With typical solid objects, this method can remove up to 50% of the polygons in a scene!

Traditionally, this step was performed near the end of the rendering pipeline because the calculations were very simple. You just test whether the polygon normal (in Camera-Space) is facing towards you (z-component is negative) or facing away from you (z-component is positive).

Culling can remove up to 50% of a scene, so we must move this step as far to the beginning of the pipeline as possible. It must be done before clipping - because clipping is slow. But, clipping is performed before transformation so we need to cull in Object-Space. The calculations become a bit more difficult, but in the end it is much more efficient.

Mathematically, the polygon is visible if the following equation is TRUE:

$$\mathbf{c} \bullet \mathbf{n} \geq d$$

where $\mathbf{c}$ = Camera location
$\mathbf{n}$ = Polygon normal (plane equation)
d = Plane distance (plane constant)

You may notice that this is exactly the same formula that we used to test which side of a plane a point resides (for clipping). In essence, that is exactly what we are doing. We're checking which side of the polygon the camera is located. The polygon is only visible if the camera is in front of it. If the polygon is facing away from us, the normal is facing away from us; therefore we are behind the polygon. Simple isn't it?

BOUNDING SPHERES

Armed with inverse transformations, 3-D clipping and backface culling, we can now create a more efficient rendering pipeline. All we have to do is re-order the steps so operations that remove polygons are placed at the beginning of the pipe. Here's one I prepared earlier:

Remove back facing polygons
↓
Create frustum through portal
↓
Clip objects to frustum
↓
Transform objects into Camera-Space
↓
Rasterise polygons into strips
↓
Draw the strips

Clipping the objects to the view frustum is a slow operation, so we must find a way to speed up this process. We want to quickly pick out any objects that don't require clipping (ie: objects that are completely inside or outside our frustum). One way to achieve this is to use bounding spheres.

A bounding sphere maps out the maximum extent of the object at any rotation. Think of the bounding sphere creating a bubble around the object. No matter what direction the object is facing, it will always be constrained inside this bubble.

Before we clip the object against the view frustum, we check whether the bubble intersects with the frustum planes. If the sphere is completely inside the frustum, the object must be completely inside the frustum. If the sphere is completely outside the frustum, we can just remove the object without testing its individual vertices. If the sphere intersects with the frustum the object MIGHT intersect with the frustum planes. In this case, we remember which planes intersect with our sphere and only clip the object against those planes!

Here is the formula for determining the bounding sphere's relationship with a plane (Yep, it that darned plane equation again).

if $\mathbf{s} \bullet \mathbf{n} - d > r$	inside plane	where	$\mathbf{n}$ = Plane normal (plane equation)
if $\mathbf{s} \bullet \mathbf{n} - d < -r$	outside plane		d = Plane distance (plane constant)
else	intersects with plane		$\mathbf{s}$ = Centre of bounding sphere
			r = Radius of bounding sphere

REFERENCES

I formulated most of the things written in this document. That means, I only present ONE method and not necessarily the best. This information was acquired over a period of about 4 months, most of the tim being spent scavenging the Internet for useful information.

[1] ABRASH, Michael. " Ramblings in Realtime", *Dr. Dobb's Sourcebook* (March/April 1995)
File: ddjclip.zip (also available on the net somewhere)

[2] HAMMERSLEY, Tom. Various 3D tutorials
Internet: http://www.users.globalnet.co.uk/~tomh/

[3] LOISEL, Sebastien. "Zed3D - A compact reference for 3d computer graphics programming"
Internet: http://www.cs.mcgill.ca/~zed/

[4] NETTLE, Paul. "Project Spandex v1.0"
Internet: http://www.grafix3d.dyn.ml.org/

[5] PIPENBRINCK, Nils. "Backface Culling in Object Space"
Internet: http://www.cubic.org/

ACKNOWLEDGEMENTS

The following people have helped me improve this document by giving me very helpful feedback. If YOU want to be on this list, just give me some feedback. (Constructive criticism is highly appreciated).

Scott Jenkin	jenkinsd@cc.curtin.edu.au	Helped me out with some the maths (especially quaternions).
Lionel Bonnetier	leo@easynet.fr	Corrected my 3-D clipping algorithm.
Jerry A Green	graphyx@sprynet.com	General grammar and maths queries.