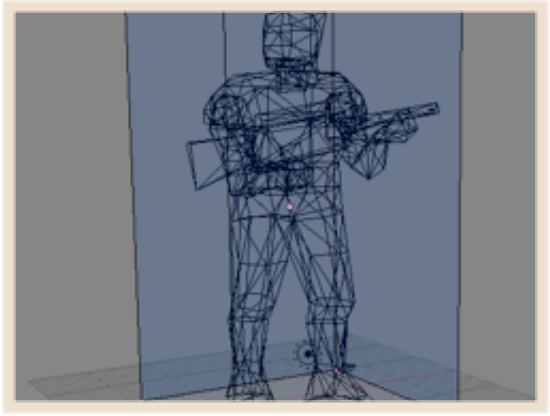
SIMON MAURICE - IPHONE - OPENGL ES



Monday, 27 July 2009

OpenGL ES 17.5 - Collision Detection 2: Moller & Trumbore's Fast Triangle Ray Intersect

So I've now had 24 hours to stew on the last tutorial and I've decided I probably shouldn't have done it that way. What seemed like a great idea to begin with, really was "over thought out". Probably better to say incorrectly thought out. Either way, I think I probably should have presented this algorithm instead.

So I'm going to do a quick post to realign collision detection in line with the rest of this series by providing a simpler perspective, and ignoring planes and internal detection algorithms; simply just present one of the most common algorithms out there: Möller & Trumbore's *Fast, Minimum Storage Ray/Triangle Intersection*.

I forget the exact circumstances in which I came across this algorithm. I'm pretty sure someone sent it to me after it was published. I've worked with it before but never really in it's "original" state. It's always been a bit customised in some way.

In learning collision detection, the published paper is probably not user friendly for those who are not used to reading all those funny maths symbols but, in this case, they kindly popped some C code at the end of their paper. This is unlike the book Moller co-authored, *Rendering in Real Time,* which is short on code, long on maths symbols.

I personally think that if you're going to spend the time writing anything targeted at programmers, you've got to include code. Pseudo code will do but there has to be the code otherwise you might as well be writing a maths book. That's probably why the *Graphics Gems* series was so good. The snippets of information were often complex, always advanced, and most went straight over my head the first (several) times I read them. However the code was included and could be treated like an API call until understood.

I'm damn certain that's why Foley & VanDams et al's *Principles in Computer Graphics* has also been so popular over the years. The first edition was used Pascal which, despite being looked down upon by "advanced" C programmers, it's like pseudo code like syntax is very readable. The second edition switched to C and was a major expansion which made that book the classic it is today.

Anyway, I digress. Back to the topic at hand.

So Moller and Trumbore (sorry Tomas, I can't be bothered inserting that special character ö each time I write your name as I don't know the keyboard shortcut :) wrote a paper in 1997 which is heavily referenced to this day. Just ask Google.

It's a Ray to Triangle intersection test which is well suited to collision detection just like what was presented in the last tutorial. A ray is cast from the current position to the destination position and, if an intersect occurs, then it is treated as a collision. Results, whilst quite similar, are not the same from this algorithm though.

Prerequisites to Determining Collision
Like the name says, minimum storage means that our data model is not required to hold much information in order to be able to execute the calculation. All we need to know is:

1. Our origin, or the "walk from" location is.
2. Direction of movement, defined as a ray.
3. The three vertices of the triangle for the test.

So really, this is about as simple as it can get as far as required information goes.

The triangle's face normal is not used, instead, the triangle's *determinant* is used. Whilst it's usually more common to use the surface normal to determine whether or not the triangle is facing away from us. Similarly, if the value of the triangle's determinant is negative, then the triangle is facing away from us.

I'm leaving 2 sided triangles alone for the moment. Whilst incredibly useful, I've got a gut feeling that it would double the quantity of emails I receive from distressed coders asking "what about the right hand rule?". :)

Implementing the Algorithm: Gathering the Ingredients
Firstly, let's look at the function prototype so we can see the parameters we're going to use:

```
int intersect_triangle(float orig[3], float dir[3],
```

```
                    float vert0[3], float vert1[3], float
vert2[3],
                    float *t, float *u, float *v);
```

Breaking that down:

- `orig` is our “walk from” location.
- `dir` is the ray direction or direction of movement.
- `vert0 ~ vert2` are the three triangle’s vertices.
- `t` represents the distance from the triangle (probably needs more discussion).
- `u & v` represent the co-ordinates of the intersect point.

We already know `orig`, and `vert0 ~ vert2`. `t, u & v` are return values so they don’t count, we just need somewhere to store them.

That only leaves dir, or the the direction of movement.

In order for this algorithm to work, we need to take our movement direction and convert it into a ray. For the slightly simplified implementation here, the ray is defined as:

```
Ray = Origin + Normalised Direction
```

First of all, in the `handleTouches` method, we calculated a vector variable which is the direction of movement (I don’t think I used this last time).

```
// This vector gives us our direction. We use it to determine
collision
    //  with walls.
    GLfloat vector[3];
    vector[0] = facing[0] - position[0];
    vector[1] = facing[1] - position[1];
    vector[2] = facing[2] - position[2];
```

Note that this is just a vector subtraction and, with the macros I used last time, could be achieved using the `VECT_SUB` macro.

This is our direction. It does need to be normalised (ie made into a unit vector, or a vector of unit length). Since I already had the SGI function `normalize()` there for the `gluLookAt()` function, I just called that to normalise the vector.

Once normalised, we can add it to the origin to complete the ray as follows:

```
    float ray[3];
    normalize(vector);
    VECT_ADD(ray, vector, position);
```

That gives us the `dir` variable for the function prototype (I’ve just called it “`ray`”).

Since this function returns an `int` to be treated like a boolean, we can then just go ahead and wrap it in an if statement like so:

```
float t, u, v;
```

```
if (intersect_triangle(position, ray, (float *)&triangleVerts[0],
                                      (float *)&triangleVerts[3],
                                      (float *)&triangleVerts[6],
                                                        &t, &u,
&v)) {
    NSLog(@"Collision: %f %f %f", t, u, v);
    return;
}
```

Implementing the Algorithm

When it comes to implementing the algorithm, I just used copy and paste... :)

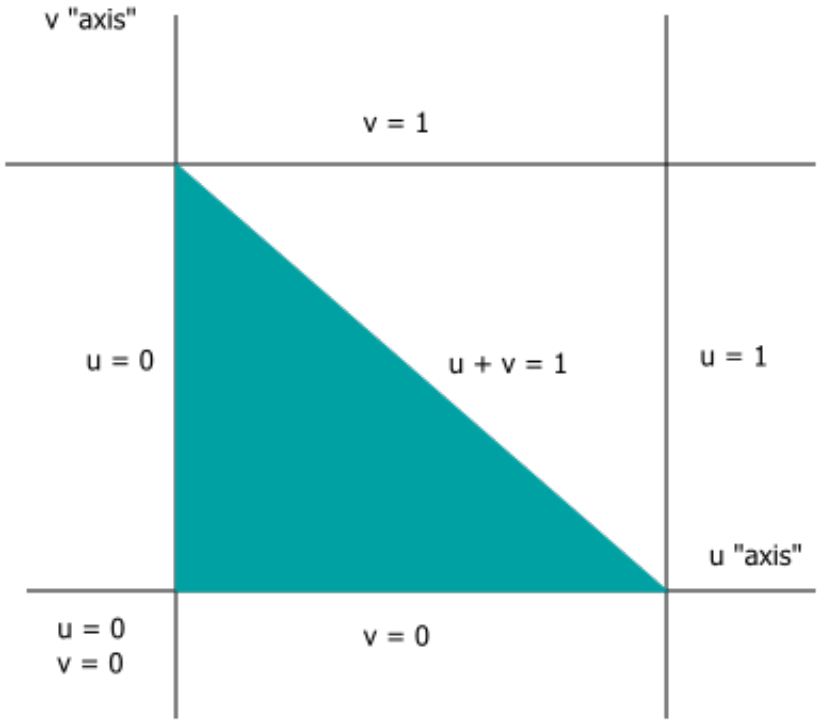
Whilst the above is true, let's start to break down what they do:

1. Calculate the two edges (as a vector) which are joined by `vert0`
2. Use these two edges to calculate the determinant of the triangle.
3. If the determinant is negative (allowing for some rounding errors defined by the `EPSILON` setting), then we are looking at the rear of a triangle which should not happen (unless you're using IDNOCLIP).
4. If all ok, then calculation for the two co-ordinates u & v commence.
5. First calculate `u`. Check to see if it's in the allowable range (see below).
6. If ok, calculate `v` and perform the same check.
7. If all still ok, complete the calculation for t and return the co-ordinates.

So basically, it's just working out firstly that the triangle is facing us, then checking *one* of the two intersect co-ordinates must be valid before calculating the other (and checking that it too is valid).

Now, the co-ordinates returned are *barycentric* co-ordinates. Yeap, just another annoying term to remember. Basically, they are like texture map co-ordinates *s & t* in as far as they start from 0 but in the case of these co-ordinates, the *sum* of *u & v* is less than 1 unit.

Have a look at this picture for a better explanation:



I didn't really want to say "u axis" or "v axis" because that's not really true. I've only labelled them as such to perhaps try and make it a bit easier to understand. However I think you can see that there are two directions, u & v just like you'd expect.

Excuse me while I yawn for a moment. It's just gone past midnight.

So let's have a look at the function now:

```
int
intersect_triangle(float orig[3], float dir[3],
                   float vert0[3], float vert1[3], float
vert2[3],
                   float *t, float *u, float *v)
{
float edge1[3], edge2[3], tvec[3], pvec[3], qvec[3];
float det,inv_det;

/* find vectors for two edges sharing vert0 */
VECT_SUB(edge1, vert1, vert0);
VECT_SUB(edge2, vert2, vert0);
```

Basic starting point. Just the usual variables declared plus the calculation of the two vectors which represent `edge1` & `edge2` (the two edges joined by `vert0` (think the top point of the triangle).

```
   /* begin calculating determinant - also used to calculate U
parameter */
    CROSS(pvec, dir, edge2);

    /* if determinant is near zero, ray lies in plane of triangle
*/
    det = DOT(edge1, pvec);

    if (det < EPSILON)
```

```
    return 0;
```

Here the determinant is calculated and then tested for a negative (or near enough) value. If we've got a negative value, then we cannot collide with this triangle.

```
/* calculate distance from vert0 to ray origin */
VECT_SUB(tvec, orig, vert0);
```

This is just a reference distance between where we are and the triangle's `vert0`. This is used later on for the calculations for u, v & t.

```
/* calculate U parameter and test bounds */
*u = DOT(tvec, pvec);
if (*u < 0.0 || *u > det)
    return 0;
```

u is the first of the two co-ordinates to get calculated. Once known, it can be checked to ensure it's within the allowable range to be considered inside the triangle (or on the edge). If not, make a quick escape!

```
/* prepare to test V parameter */
CROSS(qvec, tvec, edge1);

/* calculate V parameter and test bounds */
*v = DOT(dir, qvec);
if (*v < 0.0 || *u + *v > det)
    return 0;
```

Now everything is repeated for the *v* co-ordinate. Note the difference in the validity checking? Now that we know both u & v, we can check to ensure their sum is in bounds without need to check v individually.

Then, if we got this far:


```
/* calculate t, scale parameters, ray interse
*t = DOT(edge2, qvec);
inv_det = 1.0 / det;
*t *= inv_det;
*u *= inv_det;
*v *= inv_det;

return 1;
```

Final calculations and return a positive result.

Done.

Conclusion
I'm actually not going to discuss this any longer really. The main reason is that it's late and I want to go to bed. Other than that though, it will need to eventually be woven into a more practical context at some point in the future. Whilst just using eye position as initial co-ordinates is fine for getting the basic gist of the topic, more complexity needs to be integrated in order to do things like collision response and testing with more complex models.

For those of you who are interested, here's a link to the original paper. I hadn't read that in a few years and I was quite surprised at it's brevity.

There are lots of variations on this algorithm and it's well referenced across the internet.

Again, if you're still not following, then I think all will become clear when we hit the very practical examples.

Also, if you are mathematically minded, the book the Tomas Moller co-wrote *Rendering in Real Time* is quite a nice work. Don't expect to find code in the pages, it's all theory but if you've got a maths degree, or at least taken time to understand algebraic notation, it might be worth a read. Note that it is a more advanced work and, if you're a beginner, don't expect to understand much of it (hint: read the appendix on the maths first).

So now you've seen two completely different collision detection algorithms. Both work. The first one, whilst more complex, once completed you have everything you need really to do collision response and other details. This one, much simpler in code although uses some "esoteric" techniques for the beginner, fast, efficient but needs after-test work to be done to determine information for physics code or decal rendering.

Both have some shortcomings in other areas which I haven't gone into. Their easy to live with though.

Either works. One or the other will be used in the future. Like I said in the last tutorial, I probably should have done this one as it's got raw speed but maybe not in this exact form.

Here's the project for this tutorial. Note that I've not really made any efforts for a nice tight integration with the movement code yet. I'll do that later if needed.

Tutorial17.5.zip

You know, I think I've never given a proper explanation of what a triangle ray intersection is over the past tutorials. Hmmmm.... May need to fix that. Let me know if you don't understand and I will post something else.

Anyway, will have the next tutorial up soon. Back in business now after the sort of long delay. Looks like my social life is over... :(

Anyway, 'til next time, take care.

previous next

Made on a Mac

Email Me