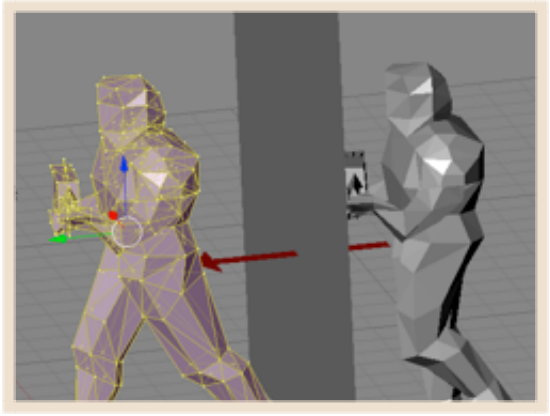
SIMON MAURICE - IPHONE - OPENGL ES



Saturday, 25 July 2009

OpenGL ES 17 - Collision Detection

It's been a bit too long since my last post. I've been involved in community theatre and just gone through the season meaning I've had very little or no time to do anything but work, theatre, eat and sleep. Right now, it's actually quite late at night (early morning actually, it's nearly 2AM) but I'm just going to try and get this out now. Actually, part of the delay since the season ended last weekend has been a few decisions on what to write.

I wrote everything which loaded a map or 3D world which, using the walking code in a previous tutorial, you could walk around in but then move through walls which is quite annoying. So I added collision detection to it and it resulted in, what would be, a huge tutorial! Especially the collision detection routines which are more maths than programming and I'm not really much of a maths fan.

Then.... I wrote 3 different collision detection routines trying to figure out which one to present. M&T, point-plane etc. *And even then....* I began looking at the Blender file thinking about adding multiple objects, ramps to walk up and just ground to a halt. Things were getting a bit out of hand so I've decided to do the following roadmap over the next couple of tutorials:

1. This one is collision detection. Damn basic and simple in execution.
2. Back into Blender maps next tutorial, this time with multiple objects per file so we can create a more interesting static world to work in. That should round up most of the Blender work.
3. Add in more collision detection to the new world. Not just walking into walls but walking up ramps, stairs, falling off ledges etc. Not just first person but third person

as well.
4. Use the environment to create OpenGL effects such as decals on walls, mirrors,water and fluids etc.

So that's where I'm headed. The first three are really already implemented, I just need to pull the separate details out of the mega-project which I had been adding things to. Number 4 will be probably a bit more random and feedback driven.

Despite leaving the last tutorial in a bit of limbo, I am going to move on because I don't think you need me to do a more complex example with the simple skinning or UV mapping. I do want to load a map file which we'll do in the next tutorial, or the one after. It just depends on whether or not this tutorial is one or two.

All I'm going to do is in this tutorial is to go back to working with a single triangle. Once you learn how to collide with a single triangle, you can test collision with a million. The principles are the same and the code can be the same, you perhaps just need to be aware of some different aspects of the type of collision you're dealing with. For example, if you are doing collision testing for walking on a landscape, you want to raise the height of the walking man as you go up a ramp but if you're collision testing for a rocket, you want it to blow that ramp up!!

I've chosen a way of collision detection here that is easier to break down into simple steps than perhaps an optimised algorithm. That's not to say it's not fast, but the fastest way isn't necessarily either the easiest to learn or suited to "most" applications. It is going to deal with triangles but can easily be expanded to deal with an object which is made up of more than one triangle (treated like a polygon if you like).

In this tutorial, we're going be using more maths than OpenGL. Actually, all maths because the OpenGL stuff you'll already know by now. There will be no fancy maths notations though, I tend not to use them as I'm a programmer, not a mathematician. My notation is C. I always prefer to look at things from a code perspective because I'd have a great deal of trouble trying to get the compiler to understand this:

C = A . B

However, the compiler easily understands this:

```
#define DOT(v1, v2) (v1[0]*v2[0] + v1[1]*v2[1] + v1[2]*v2[2])

float A = { 0, 1, 0 };
float B = { 1, 0, 1 };
float C = DOT(A, B);
```

You get my drift.

So, this is going to be about collision detection and I'm going to pretend it's a "wall" of some sort. We're going to move towards the wall and, instead of passing right though it, we're going to be forced to stop.

To get you started, here's the base project for this tutorial if you want to follow along. I'll try not to leave any thing out like I think I've done in a few recent tutorials.... ;-)

If you load this code up, hit "Build & Go", you'll see a red triangle. Click in the top

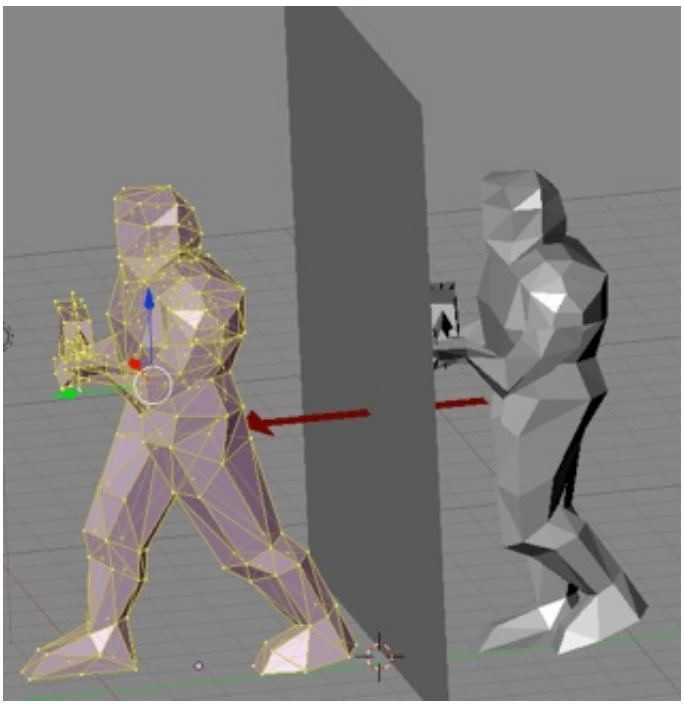
part of the screen and you'll start moving toward it and, eventually, pass right through it. So we're going to detect this collision and bring ourselves to a halt. So it's the walking man walking into a wall scenario.

The First Concepts of Detection
Just before I start, I do want to apologise in advance for any stoopid mistakes I make. Like I said at the outset, it's late here.

There are a whole plethora of ways of testing collision detection and if you've done any reading on the subject, you're bound to have found things like bounding boxes and other concepts. For the purposes of our simple testing, we're just going to do testing on our eye location. In terms of the code, that's the "look from" position in what we pass to `gluLookAt()`.

The movement code is identical to that which I wrote in that "introduction to moving in 3D" tutorial. So, in other words, each time we execute a "walk forward" movement, we are advanced a certain number of units determined by our `WALK_SPEED`.

To illustrate what happens in relation to collision detection, have a look at the following picture:



This is exaggerated of course, but if you can see, the guy on the right hand side is the position of the man ***before*** we execute the walk and he is short of the wall. ***After*** we execute the walk, he is on the other side (dude on the left). The length of the red arrow represents the displacement of the moment.

To stop the machine gun dude from walking through a wall, I'm going to break this down into a three step process. Whilst overkill for a single triangle, it can be scaled up to a larger triangle set and be used for other types of collision tests.

Our test method will perform the following three steps:

First, test to see if we cross the plane on which the triangle lies. If not, then there is no chance of collision.

Next, since we know we are going to cross the plane, get the intersect point with the plane.

Finally, determine if that intersect point is in the triangle we are currently testing.

The first test is just a quick and dirty way of deciding whether or not we need to examine this triangle in detail. If there's no need, then we don't need to waste CPU time testing it with some more expensive function calls. The second test tells us the exact point of intersection which, whilst not 100% necessary for a walk test, but it does allow for the use of this same method with things other than a "go/no go" walk test. Once we know where we intersect with the triangle's plane, we then just need to test to see if that point is inside the triangle.

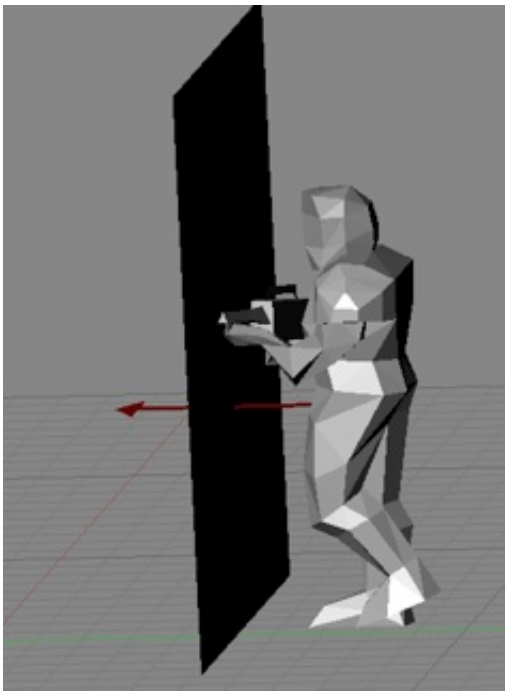
The reason for the last test is just that a plane is a mathematical concept. It's a flat surface which extends out in all directions to infinity. But the triangle only occupies a section of the plane.

First Test: Do We Cross the Plane
Testing crossing of the plane initially requires us to know two values: our position before the move, and our position after the move. We already know our position before the move, that's in the variable position. So to calculate the destination, we just use the same maths as we did before to execute a move like this:

```
destLocation[0] = position[0] + vector[0] * WALK_SPEED;
destLocation[1] = position[1] + vector[1] * WALK_SPEED;
destLocation[2] = position[2] + vector[2] * WALK_SPEED;
```

Remember that position is just our original location (before we move), vector is just our movement direction which we already worked out, and `WALK_SPEED` is how fast we walk. That's all back in tutorial 13.

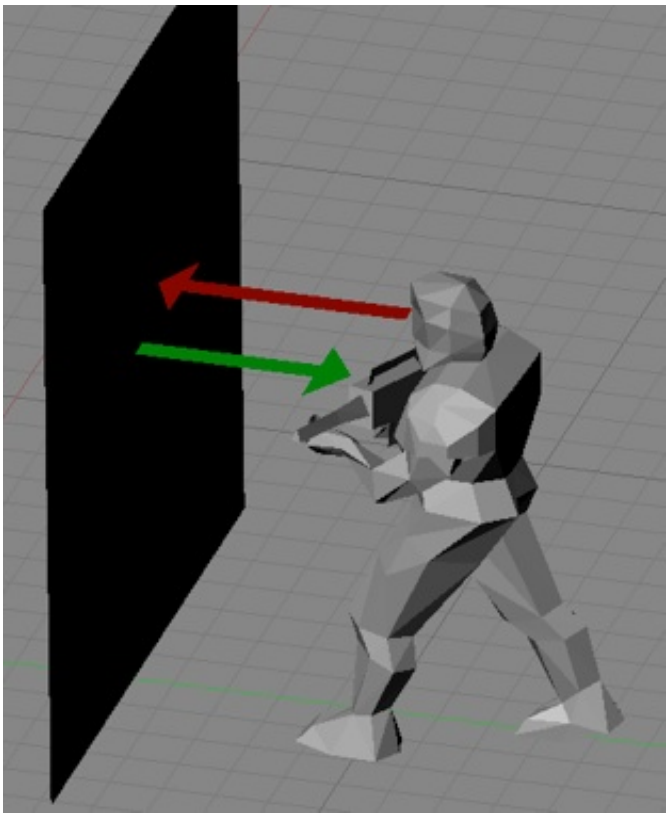
So, graphically we now have this information:



The location pointed to by the red arrow is our `destLocation`.

Ha-ha! I've just realised where I've put the arrow in relation to the rest of his anatomy!! :D Actually, that arrow should be at the eye level which is where we are testing.

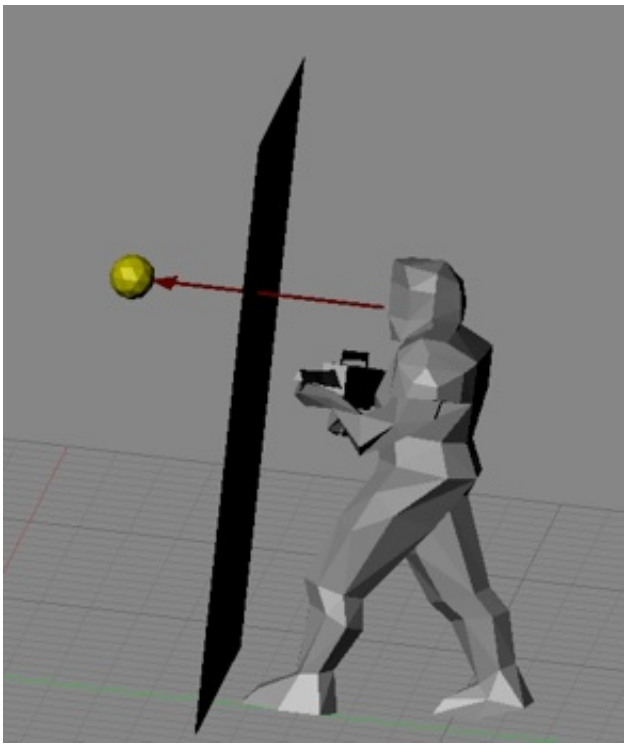
Anyway, knowing those two pieces of information, we can test if we cross the plane. To do that, we have a couple of pieces of information already at our disposal. Have a look at this picture:



(Apart from moving the red arrow upwards... :), there is a green arrow coming out of the wall. That is the wall's normal. The normal of the triangle which sits on the plane, will have the same normal as the plane. In our export from Blender, we'll grab the normal so we won't need to calculate this in a more "real" example, but in this tutorial, I've calculated the triangle's normal in a C function included in the project. That's just in case you guys are going to do something like changing the angle of the triangle which would change the normal.

The red arrow is stored in the vector variable. It's like our machine gun dude's normal for his current movement.

So if we move our machine gun dude closer to the wall, and then project out our movement so we have the movement before and movement after locations. For this example, we are going to cause the collision. So it's going to look like the picture below where the starting location is the eye location and the destination location is the yellow sphere:



Location Plane Scalar
Now we can do a calculation to determine if we cross the plane. Basically what we do is:

1. For the start location, calculate the dot product between our location and the plane's normal, then add the plane shift constant.
2. Repeat step 1 for the destination location.
3. If the sign of the two values are different, we have crossed the plane (ie Result#1 * Result#2 yields a negative value).

Taking step one, we can do it like this:

```
#define DOT(v1, v2) (v1[0]*v2[0] + v1[1]*v2[1] + v1[2]*v2[2])

float locationToPlane(float triangle[3], float againstLocation[3]) {

    // Calculate the normal of the triangle. Usually, you'd store this in
the data array.
    float normal[3];
    triangleNormal(triangle, normal);

    // Since we are treating the triangle as a plane, we need the plane's
special "D".
    float A = normal[0] * triangle[0];
    float B = normal[1] * triangle[1];
    float C = normal[2] * triangle[2];
    float D = A + B + C;                // This is the equation for a plane.

    return DOT(normal, againstLocation) + D;
}
```

The first two parts of the function could well be pre-calculated and stored in a data file if the mesh is static (eg a world map). Most walls and floors don't move so it would make sense to pre-calculate what you can.

The triangle's normal is calculated, then we need to work out the plane's plane shift constant, represented by the variable D. What I've done there is use the formula for a plane as calculated with a point and a normal, to solve for the plane shift constant.

The formula for a plane with a point on the plane and it's normal vector is:

```
float planeNormal[3] = { A, B, C };
float pointOnPlane[3] = {x, y, z};

D = A * x + B * y + C * z;
```

`PlaneNormal[0] = X` value of the plane's normal. `planeNormal[1] = Y` value of plane's normal etc.

Does something look familiar back up in the function above? Dot product perhaps?

Then, it's just a matter of the dot product with our location and the plane's normal, and add in the plane shift constant (a reference point of the plane which gives it's distance from the origin) to get a reference distance to the plane.

That needs to be repeated for the destination location.

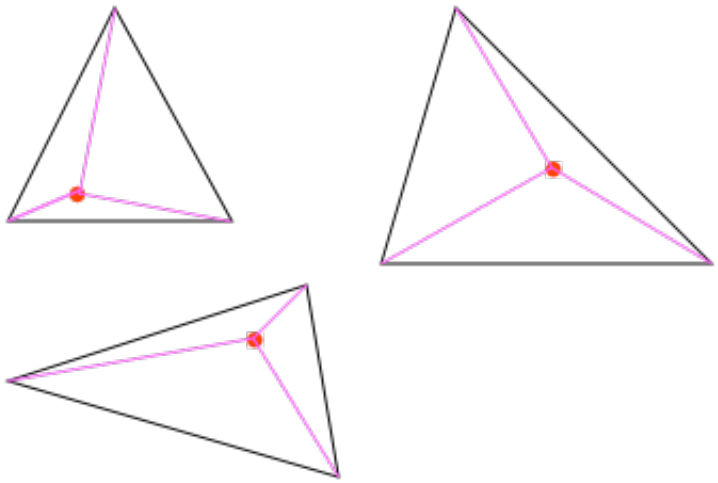
Due to the use of the plane's normal and the plane shift constant, if we have passed through the triangle, we will get a different sign on each of the two results.

Intersection Point with the Plane
Now, remember that the plane is larger than the triangle: it goes off to infinity in all directions. So just because we have intersected the plane, it doesn't mean that we've intersected the triangle. Therefore, we need to grab the intersection point and then (later) test it to discover if the intersection point is within the triangle.

```
void intersectWithPlane(float fromPos[], float toPos[], float
triangle[], float intPoint[]) {
    float normal[3];
    triangleNormal(triangle, normal);
    // Now we want the vector for the line.
    float lineVector[3];
    lineVector[0] = toPos[0] - fromPos[0];
    lineVector[1] = toPos[1] - fromPos[1];
    lineVector[2] = toPos[2] - fromPos[2];

    // Now we can start processing all this information. We have
the
    //  normal of the triangle and a vector
    //  indicating our direction. Now we can begin to detect if our
    //  movement will pass through the triangle.
    //  Start by calculating the dot product between the plane's
normal
    //  and the movement vector
    float a = DOT(lineVector, normal);

    // This should not occur but is here to avoid any possible
divide by
    //  zero errors. Actually, the only
    //   way this can occur is if we are travelling parallel to the
```

```
plane
    //   so it means that this function
    //   has been called without passing the plane intersect test.
    if (a == 0) {
        intPoint[0] = fromPos[0];
        intPoint[1] = fromPos[1];
        intPoint[2] = fromPos[2];
        return;
    }

    float b[3];

    float distance = locationToPlane(triangle, fromPos);
    b[0] = lineVector[0] * (distance / a);
    b[1] = lineVector[1] * (distance / a);
    b[2] = lineVector[2] * (distance / a);
    VECT_SUB(intPoint, fromPos, b)
}
```

Testing if the Intersection Point with the Plane is within the Triangle
Finally, we discover if we've collided with the actual triangle. Have a look at the picture below:



These are all triangles with different points on it which we could call intersection points which we calculated in the last step. Note that no matter where the intersection point is, we can divide the triangle up into three separate segments formed from the intersection point and the three vertices of the triangle?

So, there will be three segments, regardless of whether or not we're in the triangle or not. The first thing we do is to calculate the vectors for each of these three segments (ie work out the edges as a vector). Then we work out the length of the segments (needed especially for the expensive calls at the end) and do a quick elimination test if the we are on the boundary of the triangle; the boundary is included for collision detection.

The final steps are to determine the cosine of the angle between the two segments, then add the angle to an angle sum. If the angles sum up to 360º (actually 2 * PI because we pass radians as parameters), we have a collision. Note that a little bit of tolerance is allowed for due to rounding with floating point numbers.

If the point is outside the triangle, you will still get three angles but they will not sum up to 360º.

```
int pointInTriangle(float triangle[], float intPoint[]) {
    float angleSum = 0.0;
    float segment1[4], segment2[4], segment3[4];
    // There will be three segments, one for each point on the
triangle
    // There are four elements to each array. First three are the
vertex,
    //   fourth one is the length
    float *triVert;
    triVert = &triangle[3];
    VECT_SUB(segment1, triangle, intPoint)
    VECT_SUB(segment2, triVert, intPoint)
    triVert = &triangle[6];
    VECT_SUB(segment3, triVert, intPoint)

    // Calculate the segment length. Stored in the 4th element
    segment1[3] = sqrt(segment1[0]*segment1[0] +
segment1[1]*segment1[1] +
                                        segment1[2]*segment1[2]);

    segment2[3] = sqrt(segment2[0]*segment2[0] +
segment2[1]*segment2[1] +
                                        segment2[2]*segment2[2]);

    segment3[3] = sqrt(segment3[0]*segment3[0] +
segment3[1]*segment3[1] +
                                        segment3[2]*segment3[2]);

    if ((segment1[3] * segment2[3]) <= 0) {
        // Point is on boundary so include
        return 1;
    }
    if ((segment2[3] * segment3[3]) <= 0) {
        // Point is on boundary
        return 1;
    }
    if ((segment3[3] * segment1[3]) <= 0) {
        return 1;
    }

    // Now all the cheap calls are out of the way, time for the
heavy
    //   lifting
    float cosAngle = (DOT(segment1, segment2)) / (segment1[3] *

segment2[3]);
    angleSum = acos(cosAngle);

    cosAngle = (DOT(segment2, segment3)) / (segment2[3] *
segment3[3]);
    angleSum += acos(cosAngle);

    cosAngle = (DOT(segment3, segment1)) / (segment3[3] *
```

```
segment1[3]);
    angleSum += acos(cosAngle);

    // Now we look for 360° (in radians). Need to allow a little
tolerance
    if ((angleSum >= (2*M_PI - 0.0001)) && (angleSum <=
(2*M_PI+0.0001))){
        return 1;
    }
    return 0;
}
```

That's it in a nutshell. Well, a rather short nutshell but never the less, works.

One final thing, when I tested the code, I found that I would collide with the triangle but the triangle would no longer appear. A quick check revealed the problem. When I've been setting up the near clipping plane, I had been using a value of 0.1 as a distance. If we are stopped with the triangle in-between the near clipping plane and the origin, it of course isn't going to get rendered, is it? So I just changed the near clipping plane to 0.01 to fix it rather than go back and re-work some other code. Just remember your world co-ordinates are in units that you decide. 1.0 can mean 1cm, 1inch, 1 metre or 1 yard. So probably the movement code is a little fine for real world use at the moment. We can look at that later once we've rendered a complete map and see just how we want things to be.

Bringing Everything Together

The project download below contains all the code. All I've done is put the collision test in the `handleTouches` method and I only handled it for walking in the forward direction. The code looks like this:

```
    switch (currentMovement) {
        case MTWalkForward:

            // First get our destination location as if we have
just
            //  completed the move.
            destLocation[0] = position[0] + vector[0] * WALK_SPEED;
            destLocation[1] = position[1] + vector[1] * WALK_SPEED;
            destLocation[2] = position[2] + vector[2] * WALK_SPEED;

            // Now check to see if we cross the triangle's plane
            float distanceFrom = locationToPlane((float
*)triangleVerts,

position);
            float distanceAfter = locationToPlane((float
*)triangleVerts,

destLocation);
            if ((distanceFrom * distanceAfter) < 0) {
                // They differ in sign. Therefore, we cross the
plane
                float intPoint[3];
                intersectWithPlane(position, destLocation,
                                   (float *) triangleVerts,
intPoint);
```

```
                if (pointInTriangle((float *)triangleVerts,
intPoint)) {
                    NSLog(@"Collision: %f  %f  %f", position[0],
                                                    position[1],
position[2]);
                    return; // Don't move.
                }
            }

            position[0] += vector[0] * WALK_SPEED;
            position[2] += vector[2] * WALK_SPEED;
            facing[0] += vector[0] * WALK_SPEED;
            facing[2] += vector[2] * WALK_SPEED;
            break;
```

The procedure is simple. First calculate our destination location as though we completed the move. Then test to see if that crosses the plane of the triangle. If not, then we can just go ahead and move. If yes, then we need to decide if the “ray” passes through the triangle by first getting the intersection point with the plane and then test to see if that point is within the triangle.

If it’s in the triangle, we just return out. Otherwise we just perform the walk as per normal.

Conclusion
All of the above seems like a lot of work, especially compared with say, Moller & Trumbore’s Fast Ray/Triangle Intersection. Actually it is and I probably should have done Moller and Trumbore!! :) However, with multiple objects in a 3D world, plane intersection tests become quite a handy thing to know how to do. I have a hunch we’ll be using this method into the future but if it becomes a case that we don’t need to, we’ll switch algorithms.

If that seems like overload, then wait until we put it into the context of a 3D environment where you can see it properly in action.

The algorithm will change as there is so much that we can pre-calculate and speed up. Things like the triangle’s normal vector would come with the map file, then there’s distance testing using an object tree structure but that’s all for later.

Here is the project for this tutorial: Tutorial17.zip

I know it's a bit rushed n just to sit on it I always say.

Until next time, take ca
Hooroo!
Simon Maurice

previous next