Improved Collision detection and Response

Kasper Fauerby
kasper@peroxide.dk
http://www.peroxide.dk

25th July 2003

Contents

Chapter 1

Introduction

1.1 Previous work

This paper describes an updated version of the collision detection and response algorithm I first presented in [Fau00]. Some of the material has been copied directly from that article but the entire section on the collision detection step has been rewritten using a swept sphere technique as described in [Rou03]. This paper has been written in a way so it shouldn't be nessesary to read to any of the referenced papers but if something seems unclear, in particular regarding the swept sphere technique, then I encourage you to read Rouwés paper as it might go into more details on certain aspects of the algorithm. The response step is still based partly on the ideas first presented in [Net00] and partly from my own work.

This time a demo with full source will eventually be available at: http://www.dr-code.com. The demo will mostly be done by a friend of mine, Matt Ostlund also known by some as "Rhone Ranger", with probably just a little intervention by myself in the core collision code. So keep an eye on his site as well. A *huge* thanks to him for taking on this task!

1.2 The algorithm

The algorithm presented in this paper handles collision detection against arbitrary meshes stored as a so-called polygon soup. When a collision is detected the algorithm will slide the moving entity along the obstacles as it is typically seen in 3d computer games. The moving entity is approximated with an ellipsoid which gives a fairly tight fit for most humanoid or animal shapes. Ellipsoids have a very "friendly" shape that allows them to slide smoothly of the obstacles they collides against which is a good thing in a

game where the player does not want to get stuck against a hard egde in the middle of a fast-paced battle.

This updated version of the algorithm fixes in particular the problem where colliding against triangle edges could cause the original algorithm to fail resulting in the player being stuck or fall through the world geometry.

1.3 How to read this document

First of all let me say: this stuff is certainly not easy! It'll take a long time to understand correctly and probably several readings of this text. So give yourself time to fully understand what's written in a section before moving on to the next. To fully understand the theory presented you will need to have a fairly good understanding of linear algebra - i.e. vectors, matrices, vector spaces and the like. This paper is split up into two major parts - one on collision detection and one on collision response. Now what's the difference?

- Collision detection deals with finding out whether or not we'll bump into something when we move our ellipsoid though 3d space and if so, what do we hit and where?
- Collision response deals with what to do when a collision *do* occur. Do we stop? Do we continue movement in another direction ? What do we do? The collision response step is very important for the overall feel of your game.

I'll assume that the mesh being checked for collision is placed in the same space as the player - in world space. In other words I'll assume that each mesh has the identity matrix as its world matrix. If this is not the case then it's easy to apply the actual world matrix to the vertices before we check the triangles for collision.

Chapter 2

Vector spaces

This algorithm makes heavy use of non-standard vector spaces to simplify certain calculations. Those of you already familiar with the definition of vector spaces and most importantly with change-of-basis matrices to move between spaces can skip most of this chapter and only cast a glance on the section on the "ellipsoid space". The rest of you read on but be aware that this is only a very quick crash-course on the subject. A full explanation would include too much math and is beyond the scope of this paper anyway.

2.1 Definition of a vector space

Most of you will never have encountered any other vector spaces than the standard two and three dimensional spaces we usually think of from school math. These are called R_2 and R_3 and are those we use in a "standard" coordinate system (like the one D3D or openGL operates in).

As it turns out we can easily define vector spaces or 4, 5 or even infinitely many dimensions by listing a few rules (or axioms) which every vector space must follow. R_2 and R_3 follows these rules too! The rules are:

- It must be closed under addition and scalar multiplication. This means that given two vectors from the vector space, say X and Y, then $Z = X + Y$ must also be in the same space. For any real number r if X is in the space then so must rX. Informally said we cannot ever "leave" the vector space using these operations.
- It must contain a zero-vector V_0 so for all vectors X in the space $V_0 + X = X$. For all vectors X it must also be true that $0X = V_0$.
- It must follow standard mathematical rules such as the associative law, commutative law etc.

Now I'll introduce to you the concept of a linear combination of vectors from a vector space. Said in words it's what you get if you combine vectors from the space using addition and multiplication with scalars (real numbers). For example if you have vectors X, Y and Z you could combine them like this: $V = 2X + 4Y + (-3)Z$ and then V would be a linear combination of the vectors X, Y and Z.

OK, what can we say about V? Well, from rule number 1 above we can see that V must belong to the same vector space as X, Y and Z. Cool, that's what we need to define what a basis for a given vector space is.

A basis for a vector space is a collection of vectors, belonging to the space, for which the following holds:

- Every vector in the vector space is a linear combination of vectors from the basis.
- Each of the vectors in the basis can not be made from a linear combination of the other basis vectors.

The minimal number of vectors necessary to create a basis for a vector space is equal to the dimention of the space. So how many vectors are in the basis for R_3? Or R_2?

2.2 Case study: R_3

I think it's time to take a break from the abstract definitions above and look at a familiar example. After reading and understanding this example you should understand the stuff above well enough for this paper. Let us look at the standard 3D vector space we know as R_3. Is it a vector space? Well, we check the rules (very informally). Is it closed under addition? If we have two vectors from R_3, say $X = (2, 5, 4)$ and $Y = (6, 3, 8)$, is $Z = X + Y = (8, 8, 12)$ in R_3? Yes, we believe this is the case.

For any real number r, say $r = 2$, is $rX = (4, 10, 8)$ in R_3? Check! First rule holds.

Is there a zero vector then? Yes, we use $V_0 = (0, 0, 0)$ and the rule holds.

Checking the third rule involves a lot of tedious checks so I'll just mention the commutative law saying that $X + Y = Y + X$. Obviously that holds for vectors from R_3.

So R_3 is a vector space! What is its basis? The basis for R_3 is called the "standard basis" and contains 3 vectors (e_1, e_2, e_3). What are those vectors? They are: $e_1 = (1, 0, 0)$, $e_2 = (0, 1, 0)$ and $e_3 = (0, 0, 1)$.

Well, if that is indeed the basis for R_3 then we should be able to derive any vector from R_3 as a linear combination of those vectors right? Well, we'll have to convince ourselves about this so given any vector from R_3, say $V = (5, 2, 4)$, can we derive this from a linear combination of (e_1, e_2, e_3)? Yes, because in this case $5e_1+2e_2+4e_3 = 5(1, 0, 0)+2(0, 1, 0)+4(0, 0, 1) = (5, 2, 4)$. It's clear that e_1 cannot be made from e_2 and e_3 because no matter how you combine them the first vector component would still be zero right? Same can be said for the other two basis vectors.

2.3 Coordinates

Now for an important definition: the coordinate. The coordinate for a point in a vector space is build from the scalars that are multiplied to the basis vectors in the linear combination for the point. In the example above the point V has the coordinate $(5, 2, 4)$. This might seem obvious because we're used to working with R_3 but make sure you see *why* that is indeed its coordinate: $V = 5e_1 + 2e_2 + 4e_3$ so the scalars are $(5, 2, 4)$ - its coordinate.

2.4 Ellipsoid space

By now you should be convinced that R_3 is a vector space following the axioms for general vector spaces. It should also be clear that other vector spaces could exist - even other spaces of 3 dimensions. For example what about the vector space made from the basis $v_1 = (2, 0, 0)$, $v_2 = (0, 2, 0)$ and $v_3 = (0, 0, 2)$. Which vectors or points does this space contain? The same as R_3 of course but with different coordinates!

Given a point $(2, 2, 2)$ in R_3 - which coordinate would that same point have if we converted it into our new space? It would have the coordinate $(1, 1, 1)$ because: $1v_1 + 1v_2 + 1v_3 = (2, 2, 2)$.

This is *exactly* the trick we'll use in this paper to create a very special vector space which has the property that the ellipsoid we're using to approximate our player with becomes a unit-sphere in that space! A unit sphere is a sphere with radius = 1 but it's also an ellipsoid with radius *vector* = (1,1,1), right?

So how do we create such a space? Well, we'll have to come up with a basis for the space. If the ellipsoid is defined by a radius vector of (x, y, z) then we'll just chose the basis to be: $v_1 = (x, 0, 0)$, $v_2 = (0, y, 0)$ and $v_3 = (0, 0, z)$. Now in this space the ellipsoids radius vector will be $(1, 1, 1)$ - it'll be a unit sphere as we desired. You can check if that choice of basis really defines a

valid vector space if you want - or you can just take my word for it. We'll call our newly defined vector space for the "ellipsoid space".

The *last* thing we need to know is how to translate a point from R_3 into the ellipsoid space. We're in luck, because it turns out that because a linear combination is a linear function we can get away with just describing what to do with the 3 basis vectors (e_1, e_2, e_3) (remember these from above?).

We find that if we use the basis (v_1, v_2, v_3) from above, then:

$$e_1 = \frac{1}{x}v_1 + 0v_2 + 0v_3. \tag{2.1}$$

$$e_2 = 0v_1 + \frac{1}{y}v_2 + 0v_3. \tag{2.2}$$

$$e_3 = 0v_1 + 0v_2 + \frac{1}{z}v_3. \tag{2.3}$$

We list the coordinates for e_1,e_2 and e_3 as column vectors in a 3x3 matrix and get a "change of basis" matrix which has the effect of translating any vector from R_3 into the ellipsoid space.

The matrix looks like this:

$$CBM = \begin{bmatrix} \frac{1}{X} & 0 & 0 \\ 0 & \frac{1}{Y} & 0 \\ 0 & 0 & \frac{1}{Z} \end{bmatrix} \tag{2.4}$$

As long as we stick to defining new spaces for which the change of basis matrix looks like above (only has entries on the diagonal) then we can simplify the multiplication of a vector to the matrix into just a component-wise multiplication.

For example we translate a vector V into our ellipsoid space like this (remember (x,y,z) is simply the radius vector for the ellipsoid defining the space):

$$V_e = CBM * V = (\frac{1}{x}, \frac{1}{y}, \frac{1}{z})V = \frac{V}{(x, y, z)} \tag{2.5}$$

At mentioned above this *only* holds for change of basis matrices which only has entries on the diagonal and those spaces just happens to include those you get from R_3 when you multiply each basis vector (e_1, e_2, e_3) with a constant. If you check back, that's exactly what we do to achieve the 'scaling trick' above.

Chapter 3

Collision detection

3.1 Overview

OK, enough talk - lets get started! So, we have our guy we wish to move around the world. He's represented by an ellipsoid with a center which we will consider his position and a radius vector defining the ellipsoids size along the three axis. See Figure 3.1.

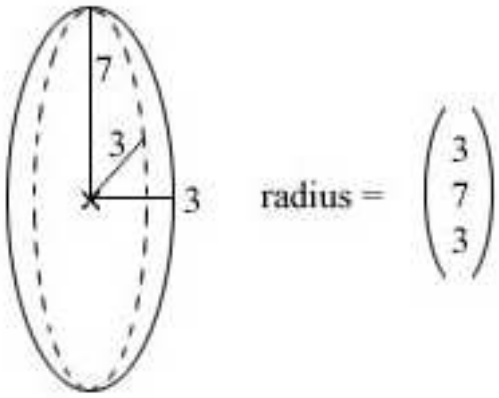

Figure 3.1: Ellipsoid radius vector

We move our guy though the world by applying a force to him in some direction. This is represented by a *velocity vector*. We wish to move the ellipsoid through the world so its new position equals its current position plus the velocity vector. See Figure 3.2.

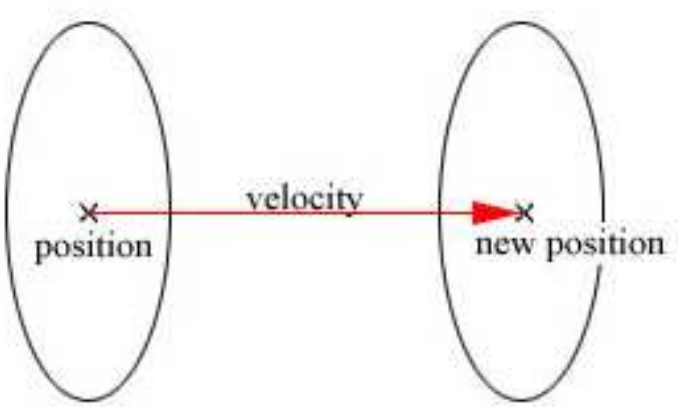

Figure 3.2: Moving our guy by a velocity

But we don't know if we can do that though as there might be something in his way - one or more of the triangles making up our world might be obstructing the path. We have no chance of knowing exactly which triangle he'll hit, so to some degree we'll have to check them all (it is possible, and recommended, to split up large meshes into an octree which can then be queried to check only those triangles close to the player).

Also we cannot stop once we find *one* triangle he'll hit - we'll have to check *all* potential colliders to find the closest collision. Figure 3.3 shows what happens if we detect a collision with triangle A and then stops without checking the rest, including for example triangle B.

First of all we'll reduce the complexity of the problem greatly by working in the ellipsoid space that I've discussed above in section 2.4 - from now on referred to as *eSpace*. As you'll recall this is a vector space in which the ellipsoid is really a unit sphere so from now on we'll only consern ourselves with detecting collision between a triangle and a unit sphere. To work in eSpace we have to translate the triangles, the velocity vector and the player position from R_3 into the space by applying the CBM matrix listed above. Once our collision detection routine has finished the result will also be in eSpace but we can translate the result back into R_3 by applying the inverse of the CBM matrix to the result[1].

As it turns out we need two things for the collision response step in the

[1]Where the transition from R_3 to eSpace was done by a division of the point with the ellipsoid radius vector it turns out that the transition back from eSpace to R_3 can be reduced to a *multiplication* of the point with the ellipsoid radius vector.

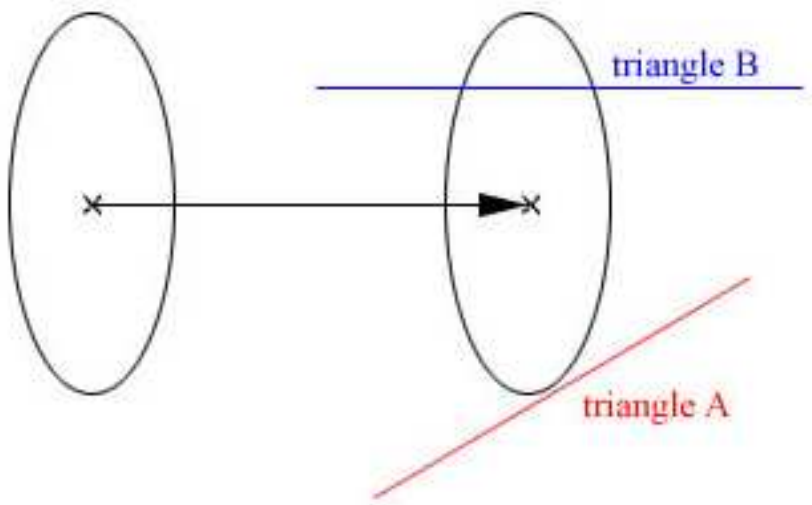

Figure 3.3: We must check all triangles

case of a collision:

- The position where the sphere hits the geometry.
- The distance along the velocity vector that the sphere must travel before the collision occurs.

So to check a single triangle for collision we'll first have to figure out *if* a collision occurs and then if that is the case the algorithm must be able to calculate the two things above.

3.2 Checking a single triangle

So given a triangle defined by points p_1, p_2 and p_3 in eSpace and in clockwise order we want to check for collision against a sphere positioned at $basePoint$ and moving along a vector $velocity$.

If we parameterize the movement of the sphere along the velocity with t then we'll get a formula for the position of the sphere that looks like this:

$$C(t) = basePoint + t * velocity, t \in [0, 1] \tag{3.1}$$

A sphere in 3d can be defined as a position and a radius (which in our case will be 1 because we're working in eSpace). For a moving sphere the

radius will remain constant but the position will of course change. Thus the formula above gives us the orientation of the collision sphere at any time t moving along the velocity vector. This is what is meant by a *swept sphere*.

The first task is to check if the swept sphere will intersect with the triangle *plane*. If this is not the case then it certainly can't collide with the triangle, can it? So we construct the plane *trianglePlane* from the points of the triangle (see Appendix B for help with this). If we have the normalized plane normal N and the plane constant C_p then we can calculate the *signed* distance from a point p to the plane as:

$$SignedDistance(p) = N \cdot p + C_p \tag{3.2}$$

If the swept sphere intersects with the triangle plane then at some time t_0 it'll rest on the *front side* of the plane and at some other time t_1 it'll rest on the *back side*. At all time values $t \in [t_0, t_1]$ the swept sphere will intersect with the plane. The time t_0 is exactly when the signed distance from the swept sphere to the triangle plane is 1 so we can calculate it like this:

$$\begin{aligned} SignedDistance(C(t_0)) = 1 \Rightarrow \\ N \cdot (basePoint + t_0 * velocity) + C_p = 1 \Rightarrow \\ N \cdot basePoint + t_0 * (N \cdot velocity) + C_p = 1 \Rightarrow \\ t_0 * (N \cdot velocity) + SignedDistance(basePoint) = 1 \Rightarrow \\ t_0 = \frac{1 - SignedDistance(basePoint)}{N \cdot velocity} \end{aligned}$$

The time t_1 is when the signed distance from the swept sphere position to the triangle plane is -1 so similar to above it can be calculated as:

$$t_1 = \frac{-1 - SignedDistance(basePoint)}{N \cdot velocity}$$

Now, if both t_0 and t_1 is outside the range $[0, 1]$ then we know that the swept sphere will *not* at any point along the velocity intersect with the plane and thus it cannot collide with the triangle. Otherwise we know that a collision with the triangle will occur at a time $t_{collision} \in [t_0, t_1]$.

Notice that there is a special case if $N \cdot velocity = 0$ - then we obviously can't use the formulas above. But when does this happen? That happens when the velocity vector is perpendicular to the plane normal - in other words when the sphere is travelling in parallel to the triangle plane. In this case there are two possibilities, either the absolute distance from $basePoint$ to the

triangle plane is smaller than 1 and the sphere is embedded in the triangle plane. If this is the case then we set $t_0 = 0$ and $t_1 = 1$ as the swept sphere intersects the plane at all times. If the distance is greater than 1 then we know that a collision cannot ever happen and we can return early from the function.

Now that we have found the two time values t_0 and t_1 there are three cases for a potential collision:

- The sphere can collide with the *inside* of the triangle.
- The sphere can collide against one of the three *vertices* of the triangle.
- The sphere can collide against one of the three *edges* of the triangle.

3.3 Colliding with the inside of a triangle

The first case is probably the most common one (depending on triangle size) and if the sphere *does* indeed collide with the inside of the triangle then a collision against a vertex or edge must happen "further down the velocity" so to speak. So if we can quickly detect collision against the inside then we can skip the more expensive "sweep test" against the vertices and edges. So lets take a look at this case first. Note that this case can only happen when the sphere is *not* embedded in the plane (as discussed above). An embedded sphere can *only* collide against a vertex or an edge of the triangle.

The idea is to calculate the point on the plane where the sphere will first make contact when moving along the velocity vector. Let call that the $planeIntersectionPoint$. We already know that the contact will happen at time t_0 because that's exactly the time when the sphere rests on the front side of the plane, but where exactly is the point? The plane intersection point is calculated like this:

$$planeIntersectionPoint = basePoint - planeNormal + t_0 * velocity \quad (3.3)$$

To see why this must be the case take a look at Figure 3.4 and Figure 3.5. The point on the *sphere* that'll first hit the plane is given by $basePoint - planeNormal$ and the intersection happens at time t_0.

Now all we need to do is check if the $planeIntersectionPoint$ is *inside* the triangle. A function that does this check is listed in Appendix C but you can use whatever function you like. If the point is inside the triangle then we have the result for the collision test:

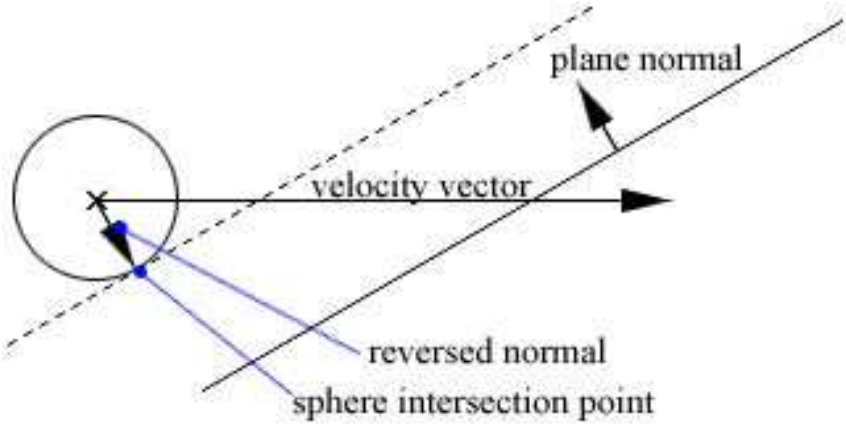

Figure 3.4: Sphere intersection point

$$\begin{aligned} intersectionPoint &= planeIntersectionPoint \\ intersectionDistance &= t_0\|velocity\| \end{aligned}$$

3.4 The sweep test

If the sphere does not collide with the inside of the triangle then we'll have to do the "sweep test" against the vertices and edges of the triangle. The idea in both cases is to check if there is a $t \in [t_0, t_1]$ where the swept sphere collides against either a vertex or edge.

Lets look at the easiest of the two cases first - the sweep against a vertex p. When does a collision between a vertex and the swept sphere take place? Well, if at any time the *distance* between the swept sphere center and the vertex is 1 then the two will collide. Or rather, to make the calculations a bit easier, when the *squared* distance between the swept center and the vertex is 1^2 - that's the same thing right? If we use that for any vector V, $V \cdot V = \|V\|^2$ then we can set up the following equation:

$$(C(t) - p) \cdot (C(t) - p) = 1^2 \tag{3.4}$$

The above reduces to a quadratic equation of the form:

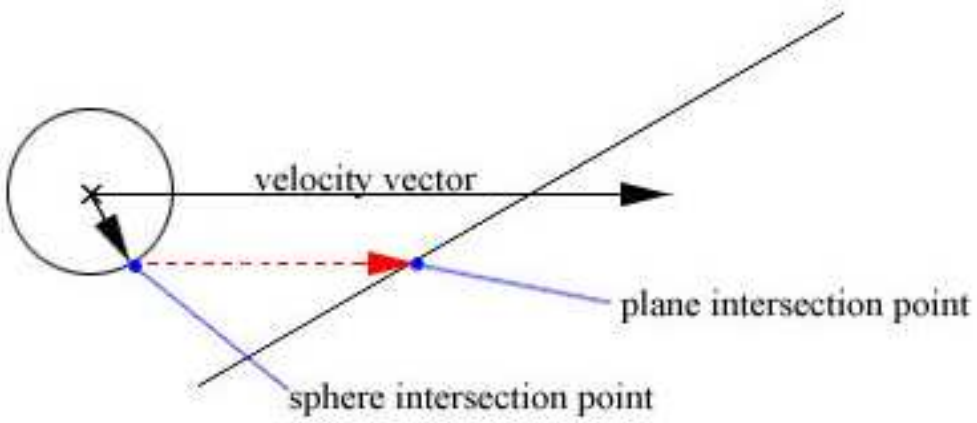

Figure 3.5: Plane intersection point

$$At^2 + Bt + C = 0 \tag{3.5}$$

where:

$$\begin{aligned} A &= velocity \cdot velocity \\ B &= 2(velocity \cdot (basePoint - p)) \\ C &= \|p - basePoint\|^2 - 1 \end{aligned}$$

A quadratic equation generally have two solution and this also makes sense in this case. There will probably be *two* times when the distance between the swept sphere and the vertex is 1 - one for each side of the plane. We're interested in the *first* collision against the vertex so we use the smallest solution (because the solutions are exactly the time values for the collisions). For help solving a quadratic equation take a look at Appendix D. If there is a smallest solution x_1 then the intersection data that we need is given as:

$$\begin{aligned} intersectionPoint &= p \\ intersectionDistance &= x_1\|velocity\| \end{aligned}$$

Even if a collision occurs with one of the vertices we'll still have to sweep against the edges of the triangle too to see if the sphere will collide against an edge at smaller time value. The sweep against an edge is a bit more complicated than the sweep against the vertex but it boils down to the same thing basically. Consider the edge going from p_1 to p_2 then let

$$\begin{aligned} edge &= p_2 - p_1 \\ baseToVertex &= p1 - basePoint \end{aligned}$$

First we check if there is any time where the swept sphere collides against the *infinite* line going through the edge. This happens when the distance between the swept sphere center and the *line* is 1. Again this reduces to a quadratic equation with:

$$\begin{aligned} A &= \|edge\|^2 * -\|velocity\|^2 + (edge \cdot velocity)^2 \\ B &= \|edge\|^2 * 2(velocity \cdot baseToVertex) - \\ &\quad 2((edge \cdot velocity)(edge \cdot baseToVertex)) \\ C &= \|edge\|^2 * (1 - \|baseToVertex\|^2) + (edge \cdot baseToVertex)^2 \end{aligned}$$

If this quadratic equation has a smallest solution x_1 then we know that the swept sphere will intersect with the infinite line at some point but we don't know if that point is within the line *segment* that makes out the triangle edge. So if we parameterize the line L by f such that $L(0) = p_1$ and $L(1) = p_2$ then we can calculate f_0 on the line where the intersection takes place as:

$$f_0 = \frac{(edge \cdot velocity)x_1 - (edge \cdot baseToVertex)}{\|edge\|^2} \tag{3.6}$$

If $f_0 \in [0, 1]$ then the intersection takes place within the line segment and the sphere collides against the triangle edge at time x_1. The intersection data that we need is given as:

$$\begin{aligned} intersectionPoint &= p_1 + f_0 edge \\ intersectionDistance &= x_1 \|velocity\| \end{aligned}$$

3.5 In summary

And that's all there is to it! To recap the whole procedure:

- We first calculated the triangle plane
- Then we found time values t_0 and t_1 where the swept sphere intersected the plane.
- First we checked if we had a collision inside the triangle. If this was the case we could skip the sweep because it must take place *before* any vertex or edge collisions.
- Finally we swept the sphere against the vertices and edges of the triangle.
- After all the above we'll know if a collision takes place against the triangle and at what point and distance. We can now check if the collision is the closest one yet in which case we store it.
- Continue to the next triangle until all has been checked.

To further help you understand how to implement this step I've included a source listing of the whole procedure of checking a triangle against a moving sphere. You can find the code listing in Appending E.

Chapter 4

Collision response

So now you know how to check if your sphere collides with any of the triangles in your world given a certain velocity vector. But what do we do when a collision actually occurs? The easiest way to handle a collision is to simply stop and not allow the move when a collision is detected. On the other hand - that is not how the professionals do is it? No, we want fancy collision response like sliding along the walls, automatically climbing stairs and automatically bumping over smaller obstacles on the ground.

Luckily all these things come automatically from the implementation of sliding that I'm going to show you. So what exactly do we mean by sliding? Well, what we want to do if we collide with a triangle in our world is to move close to it, change the direction of our velocity vector and then continue movement in a new direction. See Figure 4.1.

4.1 The sliding plane

The sliding plane is, as the name implies, the plane along which we will continue our movement. From Figure 4.1 you might get the idea that the sliding plane is identical with the plane on which the colliding triangle lies. In some cases this is also correct but as Figure 4.2 demonstrates it is not always the case:

Lets generalize the idea of the sliding plane. What does the two sliding planes in Figure 4.2 have in common? They are both the *tangent plane* to the sphere in the point where it'll collide with the triangle. So we have to find a way to calculate the tangent plane for the sphere, given a point on its surface.

Remember how we represent planes? We define them by a point on the plane and a normal to the plane, right? We already have the point - namely

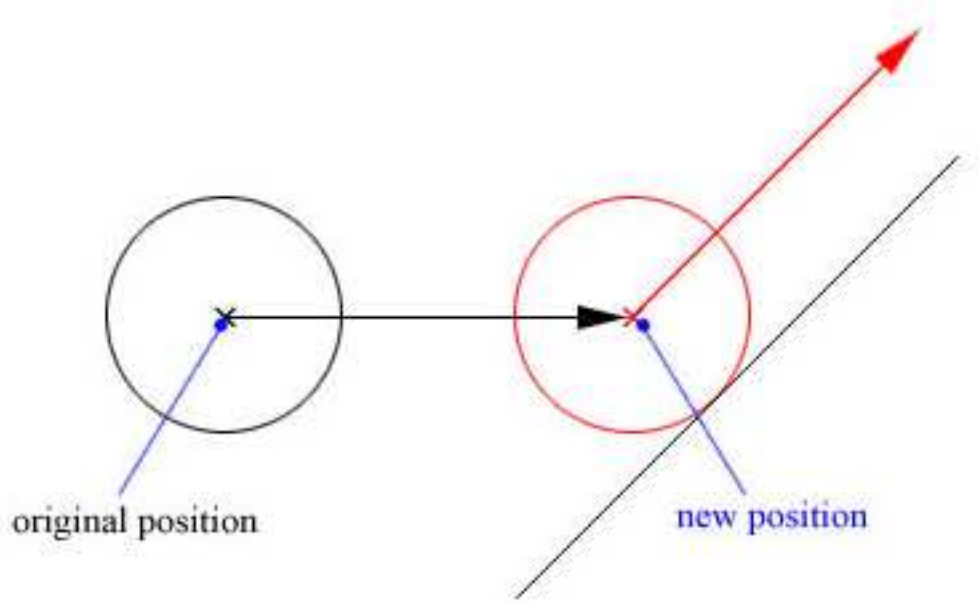

Figure 4.1: Sliding when we hit a triangle

the intersection point from the collision step - so our task at hand is really to calculate the *normal* to the tangent plane of the sphere in the intersection point.

It turns out that once again our choice of working in the ellipsoid space saves us from a lot of computations because the normal to the tangent plane at any point on a unit sphere is simply the vector going from the point on the surface to the center of the sphere! This is not the case with the more general ellipsoid shape! See Figure 4.3.

So calculating the sliding plane is easy in ellipsoid space and boils down a few lines of code:

```
VECTOR planeOrigin = intersectionPoint;
VECTOR planeNormal = newPosition - intersectionPoint;
planeNormal.normalize();
PLANE slidingPlane(planeOrigin, planeNormal);
```

Well, this isn't really a satisfying explanation is it? What if we hadn't worked in this fancy ellipsoid space? Couldn't we have computed this sliding plane anyway? Of course we could and for those interested I've included an

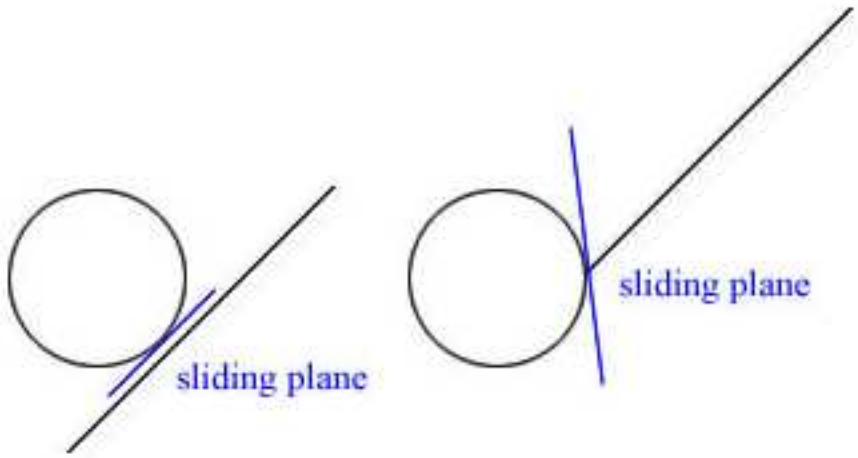

Figure 4.2: The concept of the "sliding plane"

Appendix which shows a general method for calculating the normal to *any* regular surface. In that Appendix it'll also become clear why the calculations are so simple in the case of a unit sphere. See Appendix A for more details.

4.2 Sliding the sphere

Now we know how to calculate the sliding plane but what do we do with it?. Here is the idea:

- Move our sphere as close as possible to the triangle we're colliding with. Lets call this position for "newPosition".
- Calculate the sliding plane based on this new position.
- Project the original velocity vector to the sliding plane to get a new destination.
- Make a new velocity vector by subtracting the polygon intersection point from the new destination point.

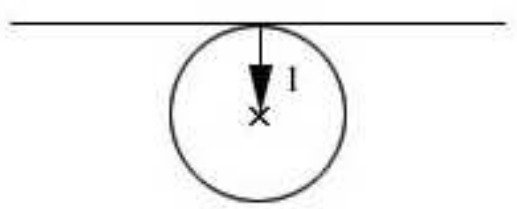

Figure 4.3: The tangent plane of a unit sphere

- Recursively call the entire collision detection routine with the new position and the new velocity vector.

The last step might come as a surprise but yes, collision detection is a recursive function and for each recursion we check all triangles all over again! We continue to recurse until either:

- We do not hit anything, so we just update the position.
- The velocity vector gets very small.

What about this projecting thing? Well, it just so happens that it does exactly what we want. It generates a vector on the sliding plane we can move along (a new velocity vector) and the more directly we hit the triangle the less do we slide. See Figure 4.4.

The only thing that requires a more in-depth explanation is perhaps how we'll project the velocity vector to the sliding plane. We have the sliding plane calculated as a point on the plane (the polygon intersection point) and a plane normal. We want to project the original destination point along the direction of the sliding plane normal onto the sliding plane. To do that we need the signed distance from the original destination point to the sliding plane. The projected point is then given as:

```
float distance = slidingPlane.distanceTo(destination);
VECTOR newDestinationPoint = destination-distance*planeNormal;
```

As outlined above it is now just a matter of calculating a new velocity vector and recursing with the newPosition and newVelocity on the collision detection and response algorithm.

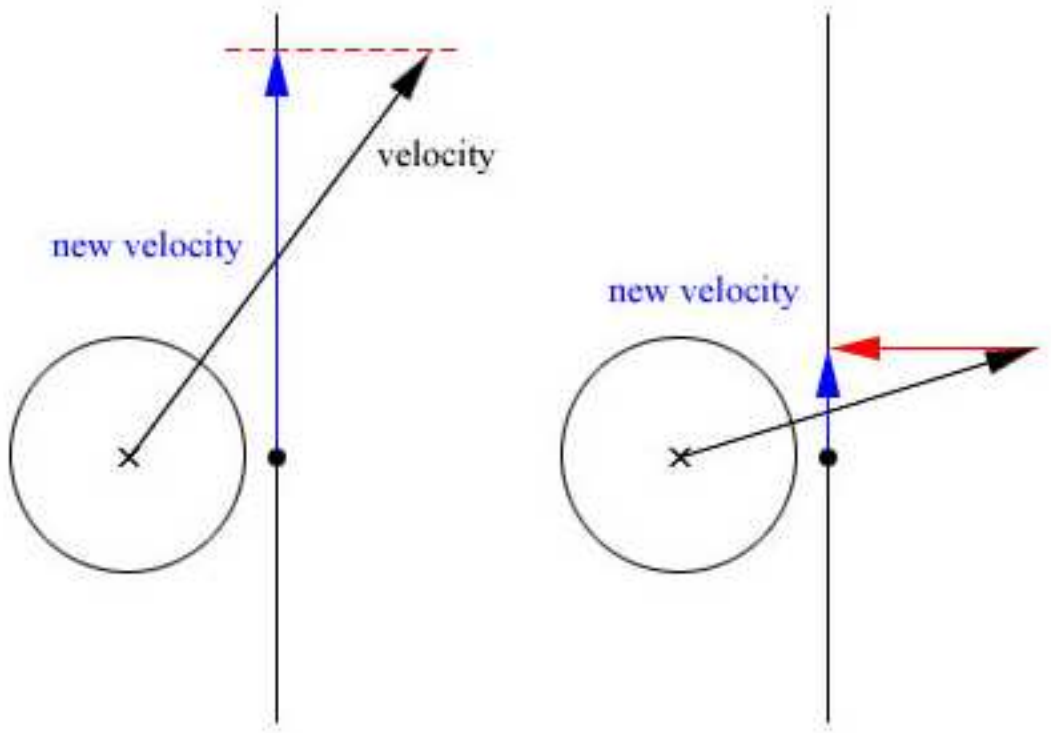

Figure 4.4: Angle of intersection affects the amount of sliding

4.3 Gravity

What if we want gravity in our world? Will that be hard? No not at all, each frame you just make *two* calls to the collision detection routines - one time with the player velocity and another time with a gravity vector. You can combine the two vectors and then just call the routines once with a single velocity vector v = velocity + gravity *but* that will make climbing stairs become more difficult as the velocity vector must be much longer for the projected vector to slide upwards enough when you hit the edge of a stair (think about it). In most cases making two separate calls wont even affect the total numbers of calls to the functions because when moving along flat ground if you combine velocity and gravity you will have two recursions as shown in Figure 4.5. If you first move along the ground with the velocity you will not hit anything, and the second call with the gravity vector will collide and not recurse because the collision is perpendicular to the surface.

4.4 In summary

So to recap the response step and the general collision detection and response algorithm:

- First we convert the input velocity and position from R_3 to eSpace.
- Then we call the collision detection routines to find the closests intersection point and the distance along the velocity we must move to hit it.
- We move the sphere very close to the intersection to get a new position.
- We calculate a sliding plane from the returned intersection data.
- We project the velocity vector onto the sliding plane to get a new destination point.
- We calculate a new velocity vector and calls the algorithm recursively.
- Finally we convert the result back to R_3 and updates the character position.
- Optionally we do the above again with a gravity vector[1].

To further help you understand this step I've included a source listing in Appendix F for the response step. Please note that this is included for inspiration only and you'll have to tweak it to match your exact needs.

It should also be noted that the code included might slide a bit too much - even the smallest slope and the gravity vector might cause the player to slide. You should implement code to detect when the player is on an *almost* flat surface and only let gravity slide when the slope is really steep. How exactly you want this to work is up to you.

[1] Gravity can be combined with the movement velocity but with a penalty when colliding against stairs.

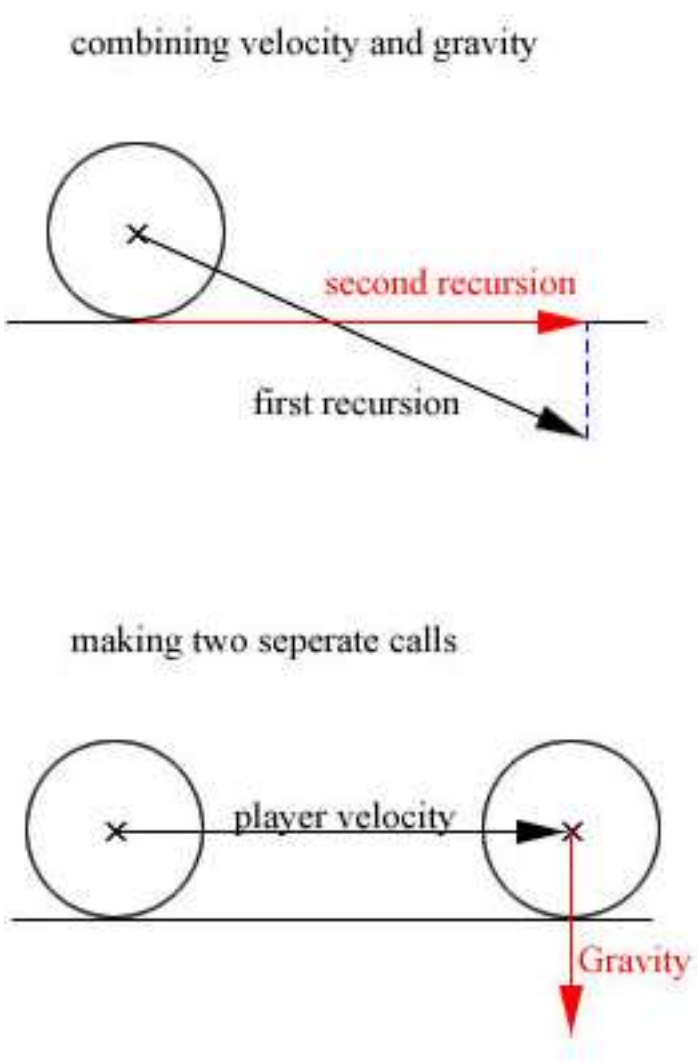

Figure 4.5: Two calls has same number of iterations as one in this case

Chapter 5

Conclusions

Well, this concludes my updated version of the collision detection and response algorithm first presented in [Fau00]. In practice it has proven to be very stable when implemented correctly. I've not had a single case of getting stuck or falling through the geometry in my own implementation in the ERA game engine.

As some of you might have noticed the function for checking a single triangle for collision is a bit long this time so you might be wondering how fast the algorithm is. Personally I haven't had any performance problems with it and one also has to remember that there are two "early out" points in the function which actually kicks in quite often. Here are some statistics: I walked around a few minutes in an ERA test map which consisted of both a terrain (build from a heightmap) and polygon meshes such as houses, stairs etc. In total 1.1 million triangles were sent to the function over the whole test period and of those approximately 40% were detected to be backfacing and thus skipped[1]. Of the remaining 700.000 triangles 65% were able to exit early from the function after just a few cheap tests to calculate the time values t_0 and t_1. Of those triangles that didn't exit early most had to perform the sweep test but a few could skip the sweep because a collision against the inside of the triangle were detected. All in all only about 20% of the triangles being sent to the function had to actually be checked for collision against the inside or edge of the triangle and thus payed the full price for the rather long function.

One should note of course that these statistics depends a lot on the dataset and on how aggressively the triangles has been culled before being sent to the function but I think the results above represents a pretty average case

[1] One would suspect 50% to be backfacing but in my case the statistics were biased by the fact that I had a terrain mesh where most triangles is frontfacing to a velocity along it.

where decent culling has been applied through the use of an octree to only send those triangles near the movement to the function. I can only encourage you to do your own tests and see how the algorithm behaves on your data.

I am aware that this paper presents incomplete code in the sense that it cannot be just cut out from this text and plugged into your own engine directly! However, I believe I've presented the material well enough for you to use what is given here *as reference* and implement your own version of the algorithm. The reason why I stress this fact here is that I've recieved a lot of questions about my previous version of the document where people complained that they couldn't find the VECTOR class or that they didn't knew how to apply these techniques on a very specific type of mesh like the X-files from Microsoft or .md3 characters from quake or whatever.

It is assumed that the reader is able to figure these things out for himself - otherwise reading this document is a bit premature and I'm afraid I cannot be of much help with questions of that type (read: I probably won't answer your mails if you ask). Of course, if you *really* think something in the presented code looks wrong or that it's impossible to understand then I won't bite if you send me an email anyway ;)

On the other hand, if you have questions or comments on the *algorithm* then I would very much like to hear them. Especially if you've spotted an error in the algorithm or in my presentation of it. Such questions or comments can be sent to the email address that I've listed on the front page of this paper.

Appendix A

Calculating the normal to a regular surface

Again this will have to be a little quick and I cannot cover every detail because then this would turn into a math tutorial. What I try to do is introduce exactly enough concepts for you to understand the important part of this section - the calculation of the tangent plane for a general regular surface.

The answer lies in the theory of differential geometry of regular surfaces. I will not go into the definition of regular surfaces but think of them as certain "smooth" surfaces in 3d - like the surface of an ellipsoid or sphere.

Ok, first lets think about how a tangent plane is defined. Most of you will have an intuitive understanding of what it is but not all have been introduced to a precise definition before. Lets go one dimension down and look at what we know about tangents in 2d. From school we know that the tangent to a curve looks like in Figure A.1 and that it is calculated by differencing its function.

In Figure A.1 the curve is in 2d but we could actually consider it as lying on a plane in 3d right? Namely the xy plane in R_3. What if I told you that we could have the curve lying on a regular surface like a sphere or ellipsoid! It would run over the surface and be parameterized by one single variable t. This is just like in 2d where the curves we know are parameterized by a single variable x (we often consider a curve in 2d as a function $f(x) = y$, right?). To do that we will have to change the definition of the function for the curve but it would still be a function of *one* variable and we would know how to differentiate it.

In 3d a curve is defined by 3 functions, each one describing the value of one of the 3 coordinates given a parameter t. So if we call the curve A then its function becomes:

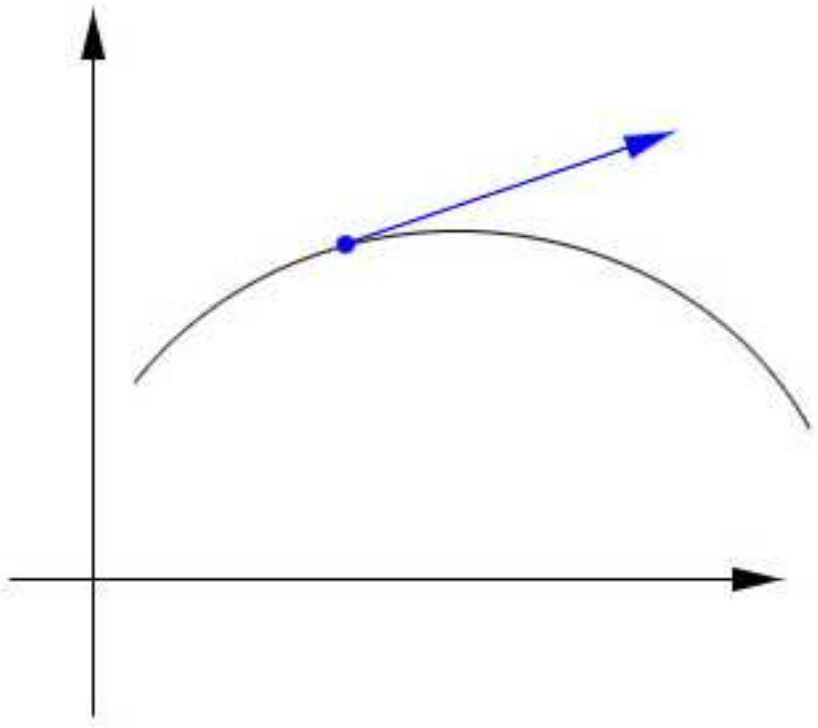

Figure A.1: The tangent to a curve in 2D

$$A(t) = (x(t), y(t), z(t)) \tag{A.1}$$

and to differentiate it we differentiate its 3 coordinate functions, so:

$$A\prime(t) = (x\prime(t), y\prime(t), z\prime(t)) \tag{A.2}$$

OK, back to the task at hand - to determine the tangent plane of the regular surface. Like described earlier to define a plane we'll need two vectors lying on the plane as well as a point on it. We already have the point because that's the point on the surface we're trying to get the tangent plane for, so what we need are the two vectors. Here is the trick: Say the point is called p. We make 2 distinct curves A and B running on the surface and both passing through p. Now if we differentiate both A and B we'll get two vectors $A\prime(p)$ and $B\prime(p)$.

Those two vectors are both tangents to the surface and they span a plane - exactly the tangent plane! See Figure A.2

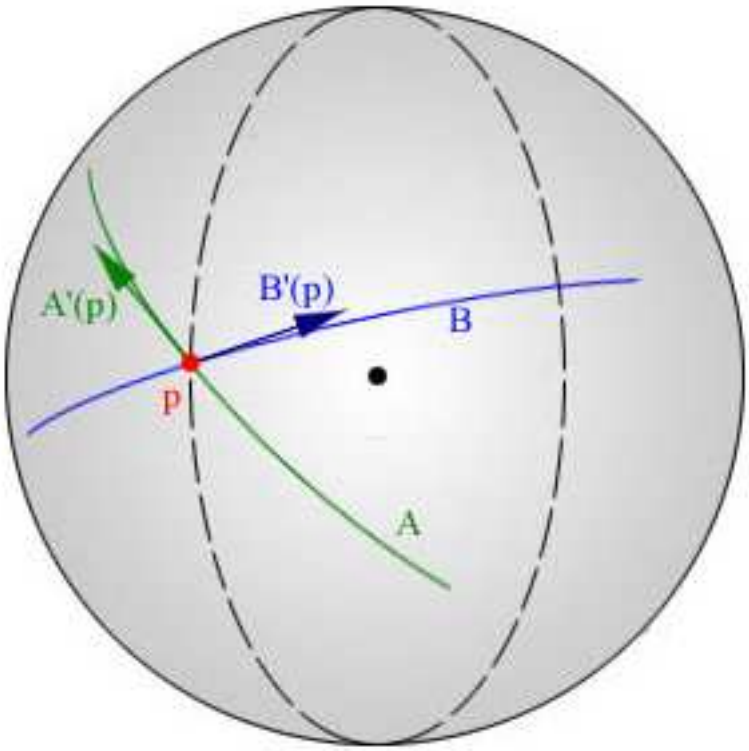

Figure A.2: Two curves on the surface defines the tangent plane

Now the only problem is that we don't *have* those two curves A and B. And there really is no easy way of finding two curves running though any point on a sphere or ellipsoid. It might seem we have a problem but first lets take a look at how the equations for a sphere and an ellipsoid looks like:

- A sphere of radius r is given by the points (x, y, z), where $x^2+y^2+z^2 = r^2$
- An ellipsoid with radius vector (a, b, c) is given by the points (x, y, z), where $\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1$.

That makes sense because if you insert a vector (r, r, r) in the equation for the ellipsoid you get exactly the equation for the sphere with radius r.

Now we are saved by a theorem which says something about a totally different thing than what we are looking for. Here is what it says:

Theorem 1 *If a regular surface is given by S = {(x,y,z) | f(x,y,z) = a}, where f is a differentiable function from R_3 into R and "a" is a constant, regular value of f - then S is orientable.*

The formula we need is in the proof of this theorem but first we have to make sure our ellipsoid fits the theorems requirements. OK, we know that an ellipsoid is a regular surface. What do we use as the function f? We let $f(x, y, z) = \frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2}$ and as described above our ellipsoid is exactly the set: $A = \{(x, y, z) | f(x, y, z) = 1\}$.

From school we know that a function like f is differentiable and 1 sure is a constant. You'll have to trust me when I say that 1 is a regular value of f. So, our ellipsoid fits the requirements of the theorem, so lets look at the proof:

- Given *any* point $p = (x_0, y_0, z_0)$ on the surface we consider *any* parameterized curve $A(t) = (x(t), y(t), z(t))$ passing through p for some value of t, say for $t = t_0$. So $A(t_0) = p$, right?
- Because the curve lies on the surface we know that $f(x(t), y(t), z(t)) = a$, for all t. (Because the theorem talks about a surface where for all points on it we have $f(p) = a$)
- If we differentiate the expression $f(x(t), y(t), z(t)) = a$ on both sides with respect to t, then for $t = t_0$ we get:

  $f\prime x(p)(\frac{dx}{dt}) + f\prime y(p)(\frac{dy}{dt}) + f\prime z(p)(\frac{dz}{dt}) = 0$

  , where dx,dy and dz all are evaluated at t_0.
- But this is the same as: $(f\prime x(p), f\prime y(p), f\prime z(p)) \cdot (\frac{dx}{dt}, \frac{dy}{dt}, \frac{dz}{dt}) = 0$
- What do we know about the dot product? We know that if the dot product is 0 then the two vectors are perpendicular to each other. So we conclude that $(f\prime x(p), f\prime y(p), f\prime z(p))$ is perpendicular to $(\frac{dx}{dt}, \frac{dy}{dt}, \frac{dz}{dt})$ in particular.
- But what is $(\frac{dx}{dt}, \frac{dy}{dt}, \frac{dz}{dt})$? That is exactly the tangent vector to the curve A, and as it were evaluated at $t = t_0$ it is exactly the tangent vector to the curve at the point p! Thus it must lie on the tangent plane for the ellipsoid at the point p.
- As the above holds for *any* curve on the surface we conclude that $(f\prime x(p), f\prime y(p), f\prime z(p))$ must be perpendicular to *any* vector in the tangent plane, and thus it must be the normal to the tangent plane at p.

Cool, now we have exactly what we need! For *any* point p on the ellipsoid we now know that the normal to the tangent plane is $(f\prime x(p), f\prime y(p), f\prime z(p))$.

So in the case of an ellipsoid we just have to differentiate $f(x,y,z) = \frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2}$ with respect to x,y and z.

$$f\prime x = \frac{2x}{a^2} + 0 + 0 = \frac{2x}{a^2} \tag{A.3}$$

$$f\prime y = 0 + \frac{2y}{b^2} + 0 = \frac{2y}{b^2} \tag{A.4}$$

$$f\prime z = 0 + 0 + \frac{2z}{c^2} = \frac{2z}{c^2} \tag{A.5}$$

Voila, what you got is a general normal-function which looks like this:

$$N(x,y,z) = (\frac{2x}{a^2}, \frac{2y}{b^2}, \frac{2z}{c^2}) \tag{A.6}$$

where (x,y,z) is a point on the sphere/ellipsoid and (a,b,c) is the radius vector. You can then normalize the result if you want.

As a last interesting point we can now see why the calculations can be simplified in the case of a unit sphere. A unit sphere is really an ellipsoid with a radius vector (1,1,1). So the general formula can be simplified so given any point on a unit sphere the normal to the tangent plane becomes: $N(x,y,z) = (2x, 2y, 2z)$. But that is really just $2(x,y,z)$. What is the length of that? Well, as (x,y,z) is on the unit sphere we know that $\|(x,y,z)\| = 1$ so the length of $2(x,y,z)$ must be 2. We want to normalize the normal, so we divide each vector component by the length and get that the normalized normal is $(\frac{2x}{2}, \frac{2y}{2}, \frac{2z}{2}) = (x,y,z)$.

So the normal to the tangent plane for a unit sphere at the point p is the point itself, so if we reverse it (which will only affect which side of the plane is front) then we have that *all* normals to the sphere go from the point on the sphere to the center of the sphere. That is exactly what we used in section 4.1 to find the sliding plane.

Appendix B

The plane class

Below is the PLANE class that I use my collision detection code:

```
class PLANE {
 public:
  float equation[4];
  VECTOR origin;
  VECTOR normal;

  PLANE(const VECTOR& origin, const VECTOR& normal);
  PLANE(const VECTOR& p1, const VECTOR& p2, const VECTOR& p3);

  bool isFrontFacingTo(const VECTOR& direction) const;
  double signedDistanceTo(const VECTOR& point) const;
};

PLANE::PLANE(const VECTOR& origin, const VECTOR& normal) {
  this->normal = normal;
  this->origin = origin;
  equation[0] = normal.x;
  equation[1] = normal.y;
  equation[2] = normal.z;
  equation[3] = -(normal.x*origin.x+normal.y*origin.y
                  +normal.z*origin.z);
}

// Construct from triangle:
PLANE::PLANE(const VECTOR& p1,const VECTOR& p2,
```

```
            const VECTOR& p3)
{
  normal = (p2-p1).cross(p3-p1);
  normal.normalize();
  origin = p1;

  equation[0] = normal.x;
  equation[1] = normal.y;
  equation[2] = normal.z;
  equation[3] = -(normal.x*origin.x+normal.y*origin.y
                  +normal.z*origin.z);
}


bool PLANE::isFrontFacingTo(const VECTOR& direction) const {
  double dot = normal.dot(direction);
  return (dot <= 0);
}

double PLANE::signedDistanceTo(const VECTOR& point) const {
  return (point.dot(normal)) + equation[3];
}
```

Appendix C

Utility functions

Below is a function that determines if a point is inside a triangle or not. This is used in the collision detection step. Thanks to Keidy from Mr-Gamemaker who posted this particular version of the function in a little competition we held about who could post the fastest one.

```
typedef unsigned int uint32;
#define in(a) ((uint32&) a)
bool checkPointInTriangle(const VECTOR& point,
     const VECTOR& pa,const VECTOR& pb, const VECTOR& pc)
{

  VECTOR e10=pb-pa;
  VECTOR e20=pc-pa;

  float a = e10.dot(e10);
  float b = e10.dot(e20);
  float c = e20.dot(e20);
  float ac_bb=(a*c)-(b*b);
  VECTOR vp(point.x-pa.x, point.y-pa.y, point.z-pa.z);

  float d = vp.dot(e10);
  float e = vp.dot(e20);
  float x = (d*c)-(e*b);
  float y = (e*a)-(d*b);
  float z = x+y-ac_bb;

  return (( in(z)& ~(in(x)|in(y)) ) & 0x80000000);
}
```

Appendix D

Solving quadratic equations

Below is a snippet of code that solves a quadratic equation and returns the lowest root, below a certain treshold (the maxR parameter):

```
bool getLowestRoot(float a, float b, float c, float maxR,
                   float* root) {
// Check if a solution exists
float determinant = b*b - 4.0f*a*c;

  // If determinant is negative it means no solutions.
if (determinant < 0.0f) return false;

  // calculate the two roots: (if determinant == 0 then
  // x1==x2 but let's disregard that slight optimization)
  float sqrtD = sqrt(determinant);
  float r1 = (-b - sqrtD) / (2*a);
  float r2 = (-b + sqrtD) / (2*a);

  // Sort so x1 <= x2
  if (r1 > r2) {
    float temp = r2;
    r2 = r1;
    r1 = temp;
  }

  // Get lowest root:
  if (r1 > 0 && r1 < maxR) {
    *root = r1;
    return true;
```

```
  }

  // It is possible that we want x2 - this can happen
  // if x1 < 0
  if (r2 > 0 && r2 < maxR) {
    *root = r2;
    return true;
  }

  // No (valid) solutions
  return false;
}
```

Appendix E

Code for collision step

Below is a source listing of the whole "triangle vs. moving sphere" procedure I discussed in Chapter 3. Data about the move is comming in through the "CollisionPacket" and this is also where we put the result of the check against the triangle. I appologize for the formatting but I really had to compress things to make them fit into a rather narrow LaTex page.

First the structure I use to pass in information about the move and in which I store the result of the collision tests:

```
class CollisionPacket {
 public:
  VECTOR eRadius;  // ellipsoid radius

  // Information about the move being requested: (in R3)
  VECTOR R3Velocity;
  VECTOR R3Position;

  // Information about the move being requested: (in eSpace)
  VECTOR velocity;
  VECTOR normalizedVelocity;
  VECTOR basePoint;

  // Hit information
  bool foundCollision;
  double nearestDistance;
  VECTOR intersectionPoint;
};
```

And below a function that'll check a single triangle for collision:

```
// Assumes: p1,p2 and p3 are given in ellisoid space:
void checkTriangle(CollisionPacket* colPackage,
     const VECTOR& p1,const VECTOR& p2,const VECTOR& p3)
{

  // Make the plane containing this triangle.
  PLANE trianglePlane(p1,p2,p3);

  // Is triangle front-facing to the velocity vector?
  // We only check front-facing triangles
  // (your choice of course)
  if (trianglePlane.isFrontFacingTo(
                   colPackage->normalizedVelocity)) {

    // Get interval of plane intersection:
    double t0, t1;
    bool embeddedInPlane = false;

    // Calculate the signed distance from sphere
    // position to triangle plane
    double signedDistToTrianglePlane =
    trianglePlane.signedDistanceTo(colPackage->basePoint);

    // cache this as we're going to use it a few times below:
    float normalDotVelocity =
          trianglePlane.normal.dot(colPackage->velocity);

    // if sphere is travelling parrallel to the plane:
    if (normalDotVelocity == 0.0f) {
      if (fabs(signedDistToTrianglePlane) >= 1.0f) {
        // Sphere is not embedded in plane.
        // No collision possible:
        return;
      }
      else {
        // sphere is embedded in plane.
        // It intersects in the whole range [0..1]
        embeddedInPlane = true;
        t0 = 0.0;
        t1 = 1.0;
      }
```

```
}
else {
  // N dot D is not 0. Calculate intersection interval:
  t0=(-1.0-signedDistToTrianglePlane)/normalDotVelocity;
  t1=( 1.0-signedDistToTrianglePlane)/normalDotVelocity;

  // Swap so t0 < t1
  if (t0 > t1) {
    double temp = t1;
    t1 = t0;
    t0 = temp;
  }

  // Check that at least one result is within range:
  if (t0 > 1.0f || t1 < 0.0f) {
    // Both t values are outside values [0,1]
    // No collision possible:
    return;
  }

  // Clamp to [0,1]
  if (t0 < 0.0) t0 = 0.0;
  if (t1 < 0.0) t1 = 0.0;
  if (t0 > 1.0) t0 = 1.0;
  if (t1 > 1.0) t1 = 1.0;
}


// OK, at this point we have two time values t0 and t1
// between which the swept sphere intersects with the
// triangle plane. If any collision is to occur it must
// happen within this interval.
VECTOR collisionPoint;
bool foundCollison = false;
float t = 1.0;

// First we check for the easy case - collision inside
// the triangle. If this happens it must be at time t0
// as this is when the sphere rests on the front side
// of the triangle plane. Note, this can only happen if
// the sphere is not embedded in the triangle plane.
```

```
if (!embeddedInPlane) {
  VECTOR planeIntersectionPoint =
        (colPackage->basePoint-trianglePlane.normal)
         + t0*colPackage->velocity;

  if (checkPointInTriangle(planeIntersectionPoint,
                           p1,p2,p3))
  {
    foundCollison = true;
    t = t0;
    collisionPoint = planeIntersectionPoint;
  }
}

// if we haven't found a collision already we'll have to
// sweep sphere against points and edges  of the triangle.
// Note: A collision inside the triangle (the check above)
// will always happen before a vertex or edge collision!
// This is why we can skip the swept test if the above
// gives a collision!
if (foundCollison == false) {
  // some commonly used terms:
  VECTOR velocity = colPackage->velocity;
  VECTOR base = colPackage->basePoint;
  float velocitySquaredLength = velocity.squaredLength();
  float a,b,c; // Params for equation
  float newT;

  // For each vertex or edge a quadratic equation have to
  // be solved. We parameterize this equation as
  // a*t^2 + b*t + c = 0 and below we calculate the
  // parameters a,b and c for each test.

  // Check against points:
  a = velocitySquaredLength;

  // P1
  b = 2.0*(velocity.dot(base-p1));
  c = (p1-base).squaredLength() - 1.0;
  if (getLowestRoot(a,b,c, t, &newT)) {
    t = newT;
```

```
  foundCollison = true;
  collisionPoint = p1;
}

// P2
b = 2.0*(velocity.dot(base-p2));
c = (p2-base).squaredLength() - 1.0;
if (getLowestRoot(a,b,c, t, &newT)) {
  t = newT;
  foundCollison = true;
  collisionPoint = p2;
}

// P3
b = 2.0*(velocity.dot(base-p3));
c = (p3-base).squaredLength() - 1.0;
if (getLowestRoot(a,b,c, t, &newT)) {
  t = newT;
  foundCollison = true;
  collisionPoint = p3;
}


// Check agains edges:

// p1 -> p2:
VECTOR edge = p2-p1;
VECTOR baseToVertex = p1 - base;
float edgeSquaredLength = edge.squaredLength();
float edgeDotVelocity = edge.dot(velocity);
float edgeDotBaseToVertex = edge.dot(baseToVertex);

// Calculate parameters for equation
a = edgeSquaredLength*-velocitySquaredLength +
    edgeDotVelocity*edgeDotVelocity;
b = edgeSquaredLength*(2*velocity.dot(baseToVertex))-
    2.0*edgeDotVelocity*edgeDotBaseToVertex;
c = edgeSquaredLength*(1-baseToVertex.squaredLength())+
    edgeDotBaseToVertex*edgeDotBaseToVertex;

// Does the swept sphere collide against infinite edge?
```

```
if (getLowestRoot(a,b,c, t, &newT)) {
  // Check if intersection is within line segment:
  float f=(edgeDotVelocity*newT-edgeDotBaseToVertex)/
           edgeSquaredLength;
  if (f >= 0.0 && f <= 1.0) {
    // intersection took place within segment.
    t = newT;
    foundCollison = true;
    collisionPoint = p1 + f*edge;
  }
}

// p2 -> p3:
edge = p3-p2;
baseToVertex = p2 - base;
edgeSquaredLength = edge.squaredLength();
edgeDotVelocity = edge.dot(velocity);
edgeDotBaseToVertex = edge.dot(baseToVertex);

a = edgeSquaredLength*-velocitySquaredLength +
    edgeDotVelocity*edgeDotVelocity;
b = edgeSquaredLength*(2*velocity.dot(baseToVertex))-
    2.0*edgeDotVelocity*edgeDotBaseToVertex;
c = edgeSquaredLength*(1-baseToVertex.squaredLength())+
    edgeDotBaseToVertex*edgeDotBaseToVertex;

if (getLowestRoot(a,b,c, t, &newT)) {
  float f=(edgeDotVelocity*newT-edgeDotBaseToVertex)/
           edgeSquaredLength;
  if (f >= 0.0 && f <= 1.0) {
    t = newT;
    foundCollison = true;
    collisionPoint = p2 + f*edge;
  }
}

// p3 -> p1:
edge = p1-p3;
baseToVertex = p3 - base;
edgeSquaredLength = edge.squaredLength();
edgeDotVelocity = edge.dot(velocity);
```

```
      edgeDotBaseToVertex = edge.dot(baseToVertex);

      a = edgeSquaredLength*-velocitySquaredLength +
          edgeDotVelocity*edgeDotVelocity;
      b = edgeSquaredLength*(2*velocity.dot(baseToVertex))-
          2.0*edgeDotVelocity*edgeDotBaseToVertex;
      c = edgeSquaredLength*(1-baseToVertex.squaredLength())+
          edgeDotBaseToVertex*edgeDotBaseToVertex;

      if (getLowestRoot(a,b,c, t, &newT)) {
        float f=(edgeDotVelocity*newT-edgeDotBaseToVertex)/
                edgeSquaredLength;
        if (f >= 0.0 && f <= 1.0) {
          t = newT;
          foundCollison = true;
          collisionPoint = p3 + f*edge;
        }
      }
    }


    // Set result:
    if (foundCollison == true) {
      // distance to collision: 't' is time of collision
      float distToCollision = t*colPackage->velocity.length();

      // Does this triangle qualify for the closest hit?
      // it does if it's the first hit or the closest
      if (colPackage->foundCollision == false ||
          distToCollision < colPackage->nearestDistance)  {
        // Collision information nessesary for sliding
        colPackage->nearestDistance = distToCollision;
        colPackage->intersectionPoint=collisionPoint;
        colPackage->foundCollision = true;
      }
    }
  } // if not backface
}
```

Appendix F

Code for response step

Below is a source listing for the response step. You will have to tweak this to match your own needs - this lising is just for inspiration. The application calls the "collideAndSlide" with velocity and gravity input in R3. The function converts this input to eSpace and calls the actual recursive collision and response function "collideWithWorld". Finally it converts the result back to R3 and updates the character position.

```
void CharacterEntity::collideAndSlide(const VECTOR& vel,
                                      const VECTOR& gravity)
{
  // Do collision detection:
  collisionPackage->R3Position = position;
  collisionPackage->R3Velocity = vel;

  // calculate position and velocity in eSpace
  VECTOR eSpacePosition = collisionPackage->R3Position/
                          collisionPackage->eRadius;
  VECTOR eSpaceVelocity = collisionPackage->R3Velocity/
                          collisionPackage->eRadius;

  // Iterate until we have our final position.
  collisionRecursionDepth = 0;

  VECTOR finalPosition = collideWithWorld(eSpacePosition,
                         eSpaceVelocity);


  // Add gravity pull:
```

```
  // To remove gravity uncomment from here .....

  // Set the new R3 position (convert back from eSpace to R3
  collisionPackage->R3Position =
                  finalPosition*collisionPackage->eRadius;
  collisionPackage->R3Velocity = gravity;

  eSpaceVelocity = gravity/collisionPackage->eRadius;
  collisionRecursionDepth = 0;

  finalPosition = collideWithWorld(finalPosition,
                 eSpaceVelocity);
  // ... to here


  // Convert final result back to R3:
  finalPosition = finalPosition*collisionPackage->eRadius;

  // Move the entity (application specific function)
  MoveTo(finalPosition);
}


// Set this to match application scale..
const float unitsPerMeter = 100.0f;

VECTOR CharacterEntity::collideWithWorld(const VECTOR& pos,
                        const VECTOR& vel)
{
  // All hard-coded distances in this function is
  // scaled to fit the setting above..
  float unitScale = unitsPerMeter / 100.0f;
  float veryCloseDistance = 0.005f * unitScale;

  // do we need to worry?
  if (collisionRecursionDepth>5)
    return pos;

  // Ok, we need to worry:
  collisionPackage->velocity = vel;
```

```
collisionPackage->normalizedVelocity = vel;
collisionPackage->normalizedVelocity.normalize();
collisionPackage->basePoint = pos;
collisionPackage->foundCollision = false;

// Check for collision (calls the collision routines)
// Application specific!!
world->checkCollision(collisionPackage);

// If no collision we just move along the velocity
if (collisionPackage->foundCollision == false) {
  return pos + vel;
}


// *** Collision occured ***

// The original destination point
VECTOR destinationPoint = pos + vel;
VECTOR newBasePoint = pos;

// only update if we are not already very close
// and if so we only move very close to intersection..not
// to the exact spot.
if (collisionPackage->nearestDistance>=veryCloseDistance)
{
  VECTOR V = vel;
  V.SetLength(collisionPackage->nearestDistance-
              veryCloseDistance);
  newBasePoint = collisionPackage->basePoint + V;

  // Adjust polygon intersection point (so sliding
  // plane will be unaffected by the fact that we
  // move slightly less than collision tells us)
  V.normalize();
  collisionPackage->intersectionPoint -=
                    veryCloseDistance * V;
}

// Determine the sliding plane
VECTOR slidePlaneOrigin =
```

```
        collisionPackage->intersectionPoint;
  VECTOR slidePlaneNormal =
        newBasePoint-collisionPackage->intersectionPoint;
  slidePlaneNormal.normalize();
  PLANE slidingPlane(slidePlaneOrigin,slidePlaneNormal);

  // Again, sorry about formatting.. but look carefully ;)
  VECTOR newDestinationPoint = destinationPoint -
  slidingPlane.signedDistanceTo(destinationPoint)*
  slidePlaneNormal;


  // Generate the slide vector, which will become our new
  // velocity vector for the next iteration
  VECTOR newVelocityVector = newDestinationPoint -
                     collisionPackage->intersectionPoint;


  // Recurse:

  // dont recurse if the new velocity is very small
  if (newVelocityVector.length() < veryCloseDistance) {
    return newBasePoint;
  }

  collisionRecursionDepth++;
  return collideWithWorld(newBasePoint,newVelocityVector);
}
```

Bibliography

[Fau00] Kasper Fauerby. Pxdtut10: Collision detection and response. 2000.

[Net00] Paul Nettle. Generic collision detection for games using ellipsoids. 2000.

[Rou03] Jorrit Rouwé. Collision detection with swept spheres and ellipsoids. 2003.