Quake's Player Movement

A Breakdown Of Its Code & Ramblings

MIA IVY

Version 3, January 2023

Contents

1 Beginning

In games, we want our movement to be responsive, accurate, and feel nice all around. You absolutely do not want to be fighting against a game's movement mechanics, as they are one of the most central pieces to a game. Quake handles this quite exceptionally, and it is one of many reasons why so many current game engines have their roots in the Quake engine. So what are we looking for in our movement? There are a few things that are essential: collision detection, acceleration, speed limits, and so forth.

We will first begin with a look at how Quake determines our velocity based on user input and view angle. Next, I will give a brief ramble about collisions, and from there, we can conclude with how Quake actually commits to translating the player in a 3D environment with the given velocity. Additionally, I have included a list of all the math functions that Quake uses throughout its code, and a brief explanation of what they do. I include the entire source code for the function being discussed, and then break it into smaller pieces with a thorough explanation of what each section does.

Part I
Velocity & Acceleration

2 PM_AirMove

`PM_AirMove`, despite its name, mainly handles finding the direction that the player should be moved in. Our player presses some input keys, and is facing a certain direction, and this function mainly does some math to construct a vector that points in the direction we intend to move the player in. So, how do we do this? Let us take a look at Quake's implementation:

```
512   fmove = pmove.cmd.forwardmove;
513   smove = pmove.cmd.sidemove;
514
515   forward[2] = 0;
516   right[2] = 0;
517   VectorNormalize (forward);
518   VectorNormalize (right);
```

dont have to worry about the z look , since our player's view is limited to the xy plane, wont have to set to 0 and normalize

So, starting at the beginning, we obtain values `smove` and `fmove`, which are values provided based on user input (the WASD keys). Then, we obtain the 3D vectors `forward` and `right`. `forward` is the vector that points in the direction the player is looking, and `right` is the vector orthogonal to `forward` on the xy-plane (there is also the vector `up`, orthogonal to both). We only care about the direction `forward` and `right` point in the xy-plane, so we

remove their z-coordinate, and then we must re-normalize them as removing this coordinate does not leave us with a normalized vector.

```
520   for (i=0 ; i<2 ; i++)
521     wishvel[i] = forward[i]*fmove + right[i]*smove;
522   wishvel[2] = 0;
523
524   VectorCopy (wishvel, wishdir);
525   wishspeed = VectorNormalize(wishdir);
```

Then, we sum these two vectors, with each vector in the sum scaled by the cvar associated with it. This makes sure that if one cvar, say the `cl_sidespeed` cvar, is smaller than `cl_forewardspeed`, it will contribute a smaller portion to this vector we are calculating. We then nuke the z-coordinate, just in case. Lastly, we copy `wishvel` into `wishdir`, and we set `wishspeed` to be the length of `wishdir`, and then normalize the vector. Thus, we have a value for the magnitude of our speed, and our intended direction.

```
527   //
528   // clamp to server defined max speed
529   //
530   if (wishspeed > movevars.maxspeed)
531   {
532     VectorScale (wishvel, movevars.maxspeed/wishspeed,
      ↪  wishvel);
533     wishspeed = movevars.maxspeed;
534   }
```

We now wish to check that our speed is within appropriate bounds. This piece of code simply checks if `wishspeed` exceeds our defined maximum speed, and if so, it scales `wishvel` by $\frac{\texttt{movevars.maxspeed}}{\texttt{wishspeed}}$, and sets `wishspeed` to be the maximum speed value.

```
539   if ( onground != -1)
540   {
541     pmove.velocity[2] = 0;
542     PM_Accelerate (wishdir, wishspeed,
      ↪  movevars.accelerate);
543     pmove.velocity[2] -= movevars.entgravity *
      ↪  movevars.gravity * frametime;
544     PM_GroundMove ();
545   }
```

Now, we check if we are grounded. If we are, we set our velocity's z-component to be 0, then we pass on `wishdir`, `wishspeed`, and `movevars.accelerate` to the function `PM_Accelerate`, which handles acceleration on the ground. We then apply gravity, although this does not really have an effect since we are grounded. Lastly, we move our player with a call to the `PM_GroundMove` function.

```
546 else
547   {  // not on ground, so little effect on velocity
548     PM_AirAccelerate (wishdir, wishspeed,
    ↪   movevars.accelerate);
549
550     // add gravity
551     pmove.velocity[2] -= movevars.entgravity *
    ↪   movevars.gravity * frametime;
552
553     PM_FlyMove ();
554
555   }
```

If we are not grounded, we simply pass the same information the the `PM_AirAccelerate` function instead, also append gravity to the z-coordinate of our velocity, and then move our player using the `PM_FlyMove` function.

3 PM_Accelerate

We want our player to accelerate in a nice fashion, and doing so is a little bit different than what you may imagine. Simply taking our intended direction vector scaled and adding a small portion of it to our current velocity every frame, with checks for a maximum velocity, will not yield nice feeling movement. Turning feels especially disgusting, and it does not respond very nicely to how we change our view angle as we move about. To make moving less awful, we want to give some leniency in exceeding the maximum velocity limit when turning. In particular, we want to give more leniency the more you turn, and enforce a more strict limit otherwise. This is how Quake chose to implement this idea:

```
400   currentspeed = DotProduct (pmove.velocity, wishdir);
401   addspeed = wishspeed - currentspeed;
402   if (addspeed <= 0)
403     return;
404   accelspeed = accel*frametime*wishspeed;
405   if (accelspeed > addspeed)
406     accelspeed = addspeed;
407
408   for (i=0 ; i<3 ; i++)
409     pmove.velocity[i] += accelspeed*wishdir[i];
```

First, we take the magnitude of our current velocity in the direction of `wishdir`, and store this as `currentspeed`. Then, we subtract this from `wishspeed`. This means that if we have a component of our velocity in the direction of `wishdir` that exceeds the cvar set max speed, this difference will be 0 or negative. If this difference is 0 or negative, we do not accelerate the player, and simply return.

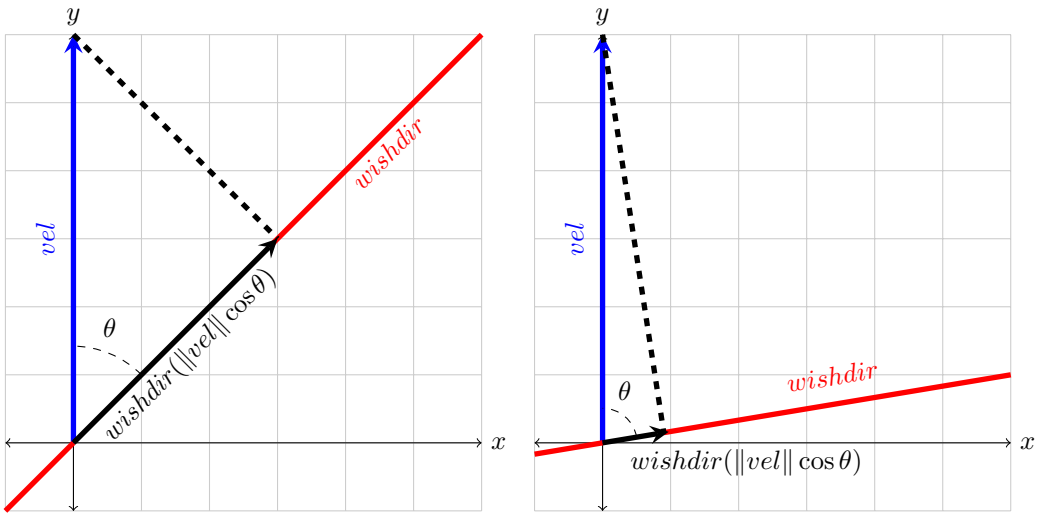


Figure 1: Projection of *vel* onto the line spanned by *wishdir*. The length of this projection is *currentspeed*. If this length exceeds *wishspeed*, we do not accelerate

Now, we find a constant `accelspeed` by scaling `wishspeed` by an acceleration constant, and factoring in the time since last frame. If this is larger than `addspeed`, then we set it to be `addspeed` instead. From here, we scale each component of `wishdir` by `accelspeed` and add it to our velocity.

This short implementation has some interesting consequences. For one, it does what we intend. If we begin with no velocity and simply hold w, we accelerate until our velocity has a component in the direction of `wishdir` that is as big as or exceeds the cvar set speed. As we are only moving straight forward, we stop accelerating exactly when our velocity is the max set velocity, which is what we want. From here, if we decide to turn to one side, we obtain a smaller component of our velocity in the direction of `wishdir`, and the more drastically we turn the smaller this component is. However, as we build speed by accelerating, this means that our overall velocity vector will gradually have a larger component in the direction of `wishdir` unless they are orthogonal, meaning that as we build speed we have a smaller range of `wishdir` angles that will give us this extra acceleration. This consequence is what we call (air)strafing.

4 PM_AirAccelerate

This function acts very similarly to `PM_Accelerate`:

```
422   if (wishspd > 30)
423     wishspd = 30;
424   currentspeed = DotProduct (pmove.velocity, wishdir);
425   addspeed = wishspd - currentspeed;
426   if (addspeed <= 0)
427     return;
428   accelspeed = accel * wishspeed * frametime;
429   if (accelspeed > addspeed)
430     accelspeed = addspeed;
431
432   for (i=0 ; i<3 ; i++)
433     pmove.velocity[i] += accelspeed*wishdir[i];
434 }
```

The main difference here is that our `wishspeed` is restricted to being within the bounds of 0 to 30, instead of our usual value (which is 400 by default for Quake). Thus, we only accelerate up to a tiny velocity if we hold w from rest (in the xy-plane) while airborne. This also means that we have a significantly reduced range of `wishdir` angles to gain speed with, and requires us to be very close to 90° to gain any extra acceleration.

5 PM_Friction

This function handles friction. It is pretty straight forward, so this section will be brief.

```
336   vel = pmove.velocity;
337
338   speed = sqrt(vel[0]*vel[0] +vel[1]*vel[1] +
      ↪  vel[2]*vel[2]);
339   if (speed < 1)
340   {
341     vel[0] = 0;
342     vel[1] = 0;
343     return;
344   }
```

To begin, we check if our speed is less than 1. If so, we simply come to a rest. This prevents friction from reducing our velocity with an asymptote where we never actually end up with 0 velocity, just an increasingly negligible small float number.

```
348 // if the leading edge is over a dropoff, increase
    ↪  friction
349   if (onground != -1) {
350     start[0] = stop[0] = pmove.origin[0] +
    ↪  vel[0]/speed*16;
```

```
351     start[1] = stop[1] = pmove.origin[1] +
    ↪   vel[1]/speed*16;
352     start[2] = pmove.origin[2] + player_mins[2];
353     stop[2] = start[2] - 34;
354
355     trace = PM_PlayerMove (start, stop);
356
357     if (trace.fraction == 1) {
358       friction *= 2;
359       }
360     }
```

This section simply determines if we are about to drop off of a ledge using a trace, which we will discuss later. If we find that we are about to move near a ledge, we double friction. The ledge needs to be at most a 34 unit drop for this to be taken into account.

```
364 if (waterlevel >= 2) // apply water friction
365     drop +=
    ↪   speed*movevars.waterfriction*waterlevel*frametime;
366   else if (onground != -1) // apply ground friction
367   {
368     control = speed < movevars.stopspeed ?
    ↪   movevars.stopspeed : speed;
369     drop += control*friction*frametime;
370   }
```

Now, we want to figure out how much to drop our velocity. Barring the water case because moving in water is fucking lame and we don't care about it, we find a variable `control` through the use of a ternary operator. If our speed is less than some minimum stop speed (this is 100 in Quake by default), then we use the stop speed, otherwise we continue using our speed. Then, we find the amount to drop our velocity as this result scaled down by friction, and our frame time factored in.

```
373 // scale the velocity
374   newspeed = speed - drop;
375   if (newspeed < 0)
376     newspeed = 0;
377   newspeed /= speed;
378
379   vel[0] = vel[0] * newspeed;
380   vel[1] = vel[1] * newspeed;
381   vel[2] = vel[2] * newspeed;
382 }
```

We find the difference between our current speed and the amount by which we intend to drop. If this is negative, we set our `newspeed` to 0. Otherwise, we

find the ratio of this `newspeed` to our old speed, and scale our velocity by this amount.

Part II
The Ungodly Prison That Is Collisions

6 Motivation

One big topic that is needed for any proper movement system is collisions. We want to know if we collide with something, and we also want to know information about the collision, and then what we want to do with this information. Generally, the information you need is if you collided or not, the depth of the collision, and the normal of the plane you collided with. Provided this information, we can accurately behave how we should when colliding with walls and floors. The reason we want this information is that we do not simply want to stop moving if we collide. If this were the case, hitting a wall at any angle or simply touching the floor would prevent us from moving. We instead want to remove the component of our velocity that is in the direction of the normal of collision. This way, when you touch ground, only gravity is removed from your velocity, thus, you stand on the object. If you walk into a wall at an angle, only the component of your velocity that is normal/perpendicular to the wall will be removed, meaning you will slide along the wall at a slower speed. It should also be noted that later into Quake's code, there is a check to see the slope of the plane you collide with below your player exceeds some magic number, and if so, we are simply considered airborne. The consequence of this check with properly implemented collisions and airstrafe is what is known as surf.

7 GJK Algorithm In A Nutshell

One popular collision algorithm that provides us a good start in detecting collisions is the Gilbert-Johnson-Keerthi algorithm, or GJK for short. This algorithm simply tells us "yes" or "no" for if we collide, but it is possible to write a more complex version of this algorithm that provides more detail. At its core, the GJK algorithm relies on an interesting observation: if we think about two shapes as simply an infinite collection of points that occupy the space of the shape, then we can do operations on these points. We can subtract the coordinates of each point in one shape from the other and generate a new shape, for example. In particular, it is interesting to note that if two shapes collide, then there exists some point inside one shape that is now also in the same coordinate position as a point in the other shape. Thus, the difference of the coordinates will be 0 (the origin).

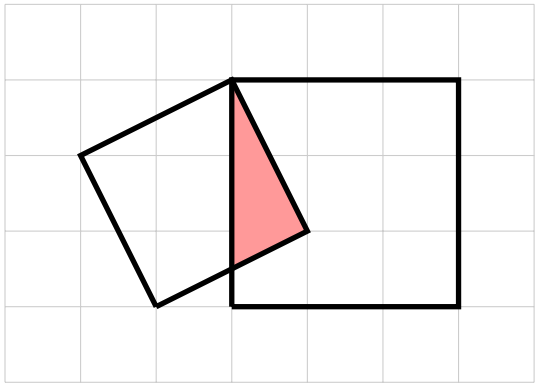


Figure 2: Two overlapping polygons. They share points with the same coordinates in the shaded region, which results in a Minkowski Difference containing the origin

This is the basis for GJK. We take two convex shapes, generate a shape by subtracting the points in one shape from another (which is usually called the Minkowski Difference), and if this new shape contains the origin, then the two original shapes must intersect. However, it is infeasible to compute the Minkowski Difference with an infinite number of points, and so instead we can simply work with the vertices of the shapes. We subtract each vertex of one shape from every other vertex of the other shape, and get a huge number of coordinates, most of which we actually do not care about. We only care about the "outer-most" coordinates of this Minkowski Difference, as this will generate a sort of "hull" or "outline" of the new shape.

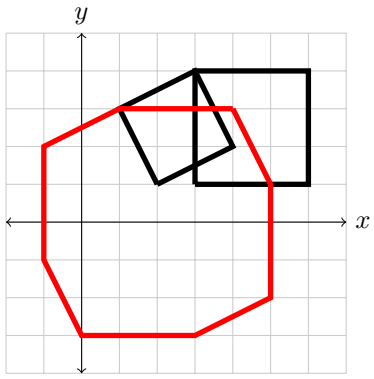
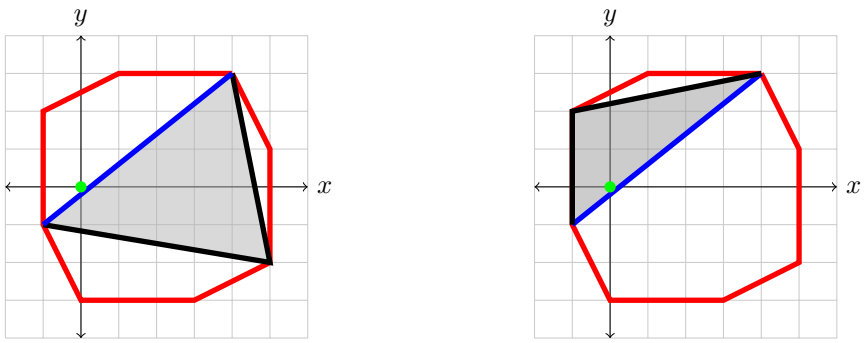


Figure 3: The Minkowski Difference hull. There are 16 total points we can calculate by finding the difference between each and every vertex, but only 8 of the resulting points act as vertices and form the hull, as all others fall somewhere within the hull

Then, we simply need to find if this hull contains the origin within it. GJK does this by working with something called a simplex, which is the bare minimum polytope that encloses a given point in $\mathbb{R}^n$. So for $\mathbb{R}^2$ this would be a triangle, as it is the bare minimum shape to enclose a point in 2 dimensions. If we are planning to implement this in 3 dimensions, our simplex is a tetrahedron. GJK works by finding a simplex bounded within the hull and doing a check if we contain the origin. If we do, we have collided and can move on. If we do not contain the origin, it finds the closest edge of the simplex to the origin, and generates a new simplex with that edge using points in the direction of the origin from that edge, as to progressively move towards the origin. If we do not contain the origin, and all the points in the direction of the origin are ones we have already used, then we know there is no such simplex bounded within our hull that will contain the origin, and thus we have not collided.



Does not contain origin

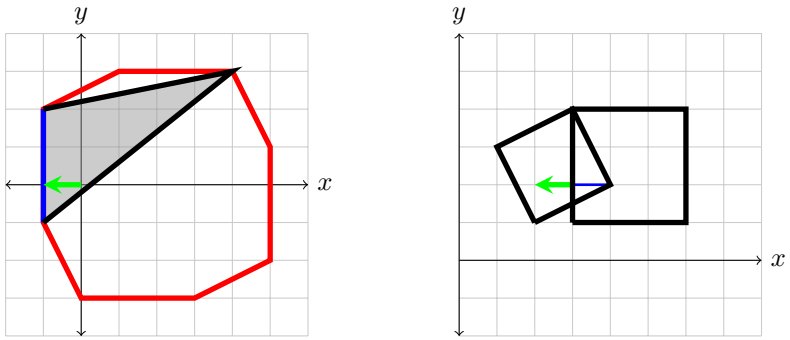
Does contain origin (Termination Simplex)

Figure 4: A simplex, with the closest edge to the origin coloured blue. The first simplex does not contain the origin, so we use this edge, and form another simplex using points in the direction of the origin from the closest edge

8 EPA Also In A Nutshell

Now, we need more information than just a simple yes or no answer to if we have collided. The Expanding Polytope Algorithm, or EPA, does just this. We take the termination simplex from GJK, and expand it using points from the Minkowski Difference hull. In particular, we want to find the edge of the polytope this forms that is closest to the origin. We can do this by looking only at the points on the hull in the direction of the normal of the closest edge on the termination simplex to the origin. This is because the termination simplex is generated by moving closer towards the origin in hopes to enclose it, and so the termination simplex should be in the region of the hull that has the origin

closest to its edges. Once we have determined which edge of the expanded polytope is in fact the closest of all the possible edges from the original hull, we can use this edge and the origin to determine penetration depth and the normal of collision. The normal of collision is the normal of the closest edge on our hull to the origin, and the depth of penetration is the distance between the origin and this edge. Thus, we can obtain our information about the collision normal and depth.



Termination Simplex from GJK *Collision normal & depth*

Figure 5: The closest edge of the hull to the origin is used for the collision normal and depth. The length of the green vector on the left is the depth of collision, and its direction gives the normal. On the right, the blue line shows the depth is indeed 1, and the green vector showing the normal does indeed point orthogonally to the surface

9 Bounded Volume Hierarchies

So now that we know how to tell if we have collided, how do we figure out what object we actually want to check against? We could simply check against every solid object that exists, however, we would prefer to not have our game run like molasses. Doing GJK-EPA is kind of expensive to do, and so the less we have to do it, the better. If we have 10,000 objects in our world, we do not want to do this computation 10,000 times. Thus, we have the Bounded Volume Hierarchy (BVH). This is a structure that organizes our world in a way that vastly reduces the time taken to figure out what we collide with. Essentially, the world is first split into two (or some small number of) gigantic volumes that both enclose everything. Then, inside each volume, we split the remaining space into smaller volumes, and we continue to divide up the space until the smallest space contains our desired number of objects. When we go to check what we collide with, we will collide with one of the two largest boxes. From here, we can immediately eliminate half of the objects in the scene from needing a check. Then we check again with the sub-boxes, and eliminate half of the remaining objects, and we

repeat checks until we have found the box containing the singular or few objects we wish to check directly against. Instead of needing to search n times, once per object that exists in your scene (which will run in $\mathcal{O}(n)$ time), our search runs in $\mathcal{O}(\log(n))$, which is a considerable improvement.

10 Binary Space Partitions

Binary Space Partitions (BSP) is like BVH in that is also an acceleration structure, but it mainly plays important roles in figuring out what areas of the map are visible to the player in given areas, which is used to figure out what we want to render at any given time without the need for real-time computations (as a table of what should be rendered is stored in memory, and generated with the compiling of the map). However, many games (such as Quake and games that run on the Source engine) use BSP for collision detection as well.

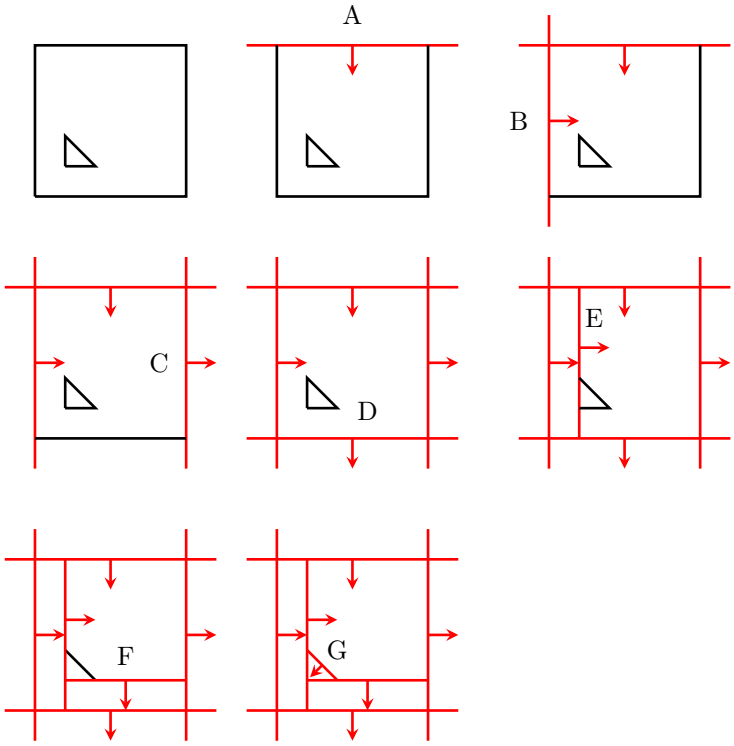


Figure 6: Example BSP partitioning

The basic premise is that we recursively slice up our map along each face, and store it in a particular manner to accelerate such searches for collisions and rendering. A binary tree is then organized with the first face at the top. Anything on one side of the slice along the face is then put onto one side of the

tree, and everything else is put onto the other side. We recursively slice each face in each leaf, and continue to separate the remaining portions of the world based on what falls on either side of the slice.

Once we have a full tree, we can make use of this for rendering purposes very easily. If we know our player sits on one side of a plane, we can immediately eliminate the leaf on the opposite side of the tree to where we are, which thereby eliminates all subsequent branches of the tree on that side. Each time we determine which side of a plane we are on we can drastically reduce the number of remaining planes to consider rendering, and efficiently build a table of which planes to render when in certain areas. A similar process can be used for collision purposes, although rendering was the main motivation for the implementation of this structure.

11 The Collision Hull

In Quake and subsequent games based on the Source engine, the player interacts with the world physically by means of an Axis-Aligned Bounding Box (AABB). An AABB is a box that remains aligned no matter what, so rotating the players view by looking around does not change the rotation of this box. This is often referred to as the collision hull, as it is used to calculate collisions (and also splash damage). Quake handles collisions via something called `pmtrace_t`. A trace essentially continuously moves a collision hull sized box along a path, and collects information about any solids it collides with along the way. It returns a fraction between 0 and 1, which represents the portion of its path that it was able to travel before collision. For example, if this fraction is 1, then it travelled all the way without colliding. If this fraction was 0.5, then it would have travelled half of its distance before colliding, and so on. It also returns the normal vectors for the planes it collided with. We can use GJK-EPA or some algorithm similar to it to determine this information. Thus, on each frame we can give the trace our current position as a starting point, and move it from our position to the position we intend to end up with our current velocity. This way, the trace will tell us the maximum step forward we can make in this direction that does not put us inside of a solid (using the trace fraction). As well, we want to remove the component of our velocity in the direction of the normals of collision. Explicitly, for a normal $\vec{n}$ and player velocity $\vec{v}$, we can edit our velocity to move parallel with $\vec{v} - (\vec{v} \cdot \vec{n})\vec{n}$. Thus, our velocity puts us along the surface, so we slide along it instead of entirely coming to a halt when impacting a surface. There is much more nuance to this, but most of it will be explained later during the discussion of the actual movement functions that Quake uses.

Part III
So Now We Can Actually Move, I Guess

12 PM_ClipVelocity

To begin working with collision detection, we need something that takes care of properly projecting our velocities along the planes we will collide with. For the unfamiliar, vector projection gives you the component/portion of one vector in the direction of another vector. One other important thing to note about this is that if two vectors have a 90° to each other, then they share no component in the direction of each other. We give this the special name of being 'orthogonal', as sometimes angles between vectors does not really make sense, but the notion of orthogonality remains useful, thus this abstracted term is nice to have.

```
72 int PM_ClipVelocity (vec3_t in, vec3_t normal, vec3_t
   ↪  out, float overbounce)
73 {
74   float  backoff;
75   float  change;
76   int    i, blocked;
77
78   blocked = 0;
79   if (normal[2] > 0)
80     blocked |= 1;    // floor
81   if (!normal[2])
82     blocked |= 2;    // step
83
84   backoff = DotProduct (in, normal) * overbounce;
85
86   for (i=0 ; i<3 ; i++)
87   {
88     change = normal[i]*backoff;
89     out[i] = in[i] - change;
90     if (out[i] > -STOP_EPSILON && out[i] < STOP_EPSILON)
91       out[i] = 0;
92   }
93
94   return blocked;
95 }
```

This function is passed a vector we want to project along a surface, the surface normal, the vector to copy the result into, and a float called `overbounce` which acts as yet another epsilon value. First, we determine if the normal

is a floor, which is if it has a z-component greater than 0, or if the surface normal is for a wall which would be a z-component of 0. Next, we find the component of our velocity in the direction of the normal, scaled a bit extra by the variable `overbounce`, and we subtract this from our input vector. If we have an `overbounce` value >1, then some portion of our velocity points back towards us, and thus we do not actually move parallel to the plane. Lastly, if any of the components of the output vector are within the bounds of |`STOP_EPSILON`|, we set the component to 0. This prevents us from having extremely small movements when against planes, and lets us snap to a stop when we collide with sufficiently slow speeds. The result is already stored in the vector `out`, and so we return the `blocked` flag. Interesting to note that this return value is never used, and later in Quake 3 this becomes a void function.

13 PM_FlyMove

We actually want to implement a more general movement function first. `PM_FlyMove` handles translating the player in all 3 dimensions, and later the function `PM_GroundMove` handles moving when on the ground in particular. This is because we want to include specific extra math for handling stairs and such, and thus separating the general collision detection and movement into one function allows us to just call this function when we may need it in the more specific function later on. Thus, understanding `PM_FlyMove` is important for movement here on out. Let us take a look at how it is written:

```
128   time_left = frametime;
129
130   for (bumpcount=0 ; bumpcount<numbumps ; bumpcount++)
131   {
132     for (i=0 ; i<3 ; i++)
133       end[i] = pmove.origin[i] + time_left *
      ↪  pmove.velocity[i];
134
135     trace = PM_PlayerMove (pmove.origin, end);
136
137     if (trace.startsolid || trace.allsolid)
138     {  // entity is trapped in another solid
139       VectorCopy (vec3_origin, pmove.velocity);
140       return 3;
141     }
142
143     if (trace.fraction > 0)
144     {  // actually covered some distance
145       VectorCopy (trace.endpos, pmove.origin);
146       numplanes = 0;
147     }
148
```

```
149     if (trace.fraction == 1)
150        break;    // moved the entire distance
```

This mainly does what we described the collision hull to do in the section above. It manually compiles a list of every plane it intersects with and does a variety of tricks to determine how we should move.

```
137 if (trace.startsolid || trace.allsolid)
138     {  // entity is trapped in another solid
139       VectorCopy (vec3_origin, pmove.velocity);
140       return 3;
141     }
142 
143     if (trace.fraction > 0)
144     {  // actually covered some distance
145       VectorCopy (trace.endpos, pmove.origin);
146       numplanes = 0;
147     }
148 
149     if (trace.fraction == 1)
150        break;    // moved the entire distance
```

The first of these checks is a basic check to determine if our trace begins already inside an object, in which case we are probably stuck inside an object. The next check takes a look at the trace fraction. If it is greater than 0, then we know we covered some distance without colliding, and so we should move some amount based on this fraction (which we will do later). If it is 1, then we have moved the entire distance without colliding, and so we do not have to worry about editing our path.

```
156 if (trace.plane.normal[2] > 0.7)
157     {
158       blocked |= 1;    // floor
159     }
160     if (!trace.plane.normal[2])
161     {
162       blocked |= 2;    // step
163     }
```

The next check that Quake makes is in relation to sloped surfaces. We want to move along surfaces that do not slope very much as though we were on normal ground, and we want to slide down surfaces with sufficiently high slopes. This very simple check does not need to compute any angles or dot products to determine the slope, but instead references a nice observation: any normalized vector that has a component in the z direction greater than 0.7 will have a slope less than 45°. This is because if we consider a unit vector that points directly upward, and begin to rotate it towards the xy-plane, it will progressively

decrease in its z-component. The magic number for a slope of roughly 45° is when it has a component of 0.7 in the z-axis. Any less, and the normal of collision points at a shallow angle from the xy-plane, which means the actual surface is at a larger angle. The second check is to make sure we even have a z-component. If we do not, then we know the surface normal must be perfectly at 90° to the xy-plane, meaning we are colliding with a flat wall and not a slope of some sort. In this case, we would want to additionally run a check to see if we can step up a stair height and move forward, as we do not want to simply slam into a 1-unit tall piece of geometry and stop moving. If it was slanted we would move up it, if it is a wall it has the potential to be a step.

```
165 time_left -= time_left * trace.fraction;
166
167   // cliped to another plane
168     if (numplanes >= MAX_CLIP_PLANES)
169     {  // this shouldn't really happen
170       VectorCopy (vec3_origin, pmove.velocity);
171       break;
172     }
```

The first line here finds the time we spent travelling before collision, and subtracts that from the total time we have between frames, leaving us with the amount of time that we would spend moving inside the object we collide with. Next, we do a quick check to see if we collide with more planes than the cvar `MAX_CLIP_PLANES`, which by default is 4. If we collide with more than 4 planes, we do not do anything.

```
178 // modify original_velocity so it parallels all of the
    ↪  clip planes
179 //
180     for (i=0 ; i<numplanes ; i++)
181     {
182       PM_ClipVelocity (original_velocity, planes[i],
    ↪  pmove.velocity, 1);
183       for (j=0 ; j<numplanes ; j++)
184         if (j != i)
185         {
186           if (DotProduct (pmove.velocity, planes[j]) < 0)
187             break;  // not ok
188         }
189       if (j == numplanes)
190         break;
191     }
192
193     if (i != numplanes)
194     {  // go along this plane
195     }
```

```
196         else
197         {  // go along the crease
198           if (numplanes != 2)
199           {
200 //            Con_Printf ("clip velocity, numplanes ==
    ↪  %i\n",numplanes);
201             VectorCopy (vec3_origin, pmove.velocity);
202             break;
203           }
204           CrossProduct (planes[0], planes[1], dir);
205           d = DotProduct (dir, pmove.velocity);
206           VectorScale (dir, d, pmove.velocity);
207         }
```

Now that we have determined generally what it is that we have collided with, such as a wall or ramp, we can move accordingly. For each plane that we intersect, we want to project our velocity along the plane using its normal. This is where we use `PM_ClipVelocity`. As well, we consider the dot product of this adjusted velocity against the normal of the other planes we collide with. If this dot product is positive, then we have some component of our velocity in the same direction as the normal, meaning we are moving into the plane. We do not want to do this, so we break. Otherwise, we want to do a check to see if we collide with 1 or more planes. If we collide with 1 plane, we just move along it. If we collide with 2, we take the cross product of the two planes we collide with, and project our velocity onto the cross product. This may seem odd, but it makes sense if we consider a few examples:

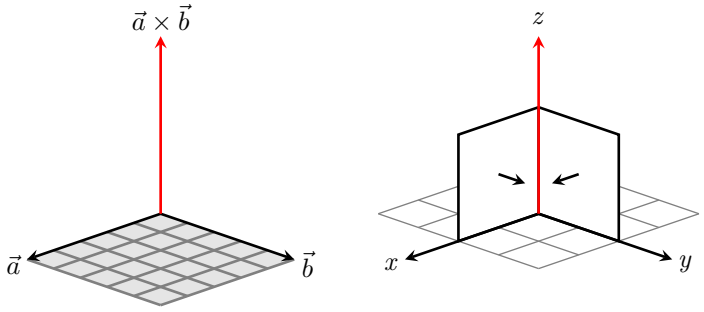


Figure 7: Left: The cross product of a vector $\vec{a}$ with a vector $\vec{b}$, denoted in red. The length of the cross product is the area of the shaded area. Right: An example of two planes forming a corner. The planes normals are shown, and the cross product points straight up along the z-axis

If we collide with two walls at once, we are in a corner and thus do not want to move horizontally. The cross product of two normals that point straight out towards us must give a vector that points directly up or down, meaning that

unless we are falling or jumping, any horizontal components of our velocity will be nuked in the projection onto the cross product, giving us a velocity of 0 as we intend.

If we also consider sloped surfaces, it makes sense. Consider falling into a V-shaped wedge between two ramps. The cross product of the two normals would produce a vector that points along the crease, meaning our velocity moves us along the surface of the two ramps, as we'd expect.

```
210 // if original velocity is against the original velocity,
    ↪  stop dead
211 // to avoid tiny occilations in sloping corners
212 //
213     if (DotProduct (pmove.velocity, primal_velocity) <=
    ↪  0)
214     {
215       VectorCopy (vec3_origin, pmove.velocity);
216       break;
217     }
218   }
219
220   if (pmove.waterjumptime)
221   {
222     VectorCopy (primal_velocity, pmove.velocity);
223   }
224   return blocked;
225 }
```

The last portion of `PM_FlyMove` takes care of the case when our updated velocity points orthogonal or backwards to the original velocity, and if so, we halt. This ensures we do not get stuck in a weird cycle of moving some super small amount towards a wall, getting moved back a bit, and then able to move forwards that bit again, and repeatedly oscillating between hitting a wall and being pushed back.

14 PM_GroundMove

This function takes care of more of the specific motion when on the ground, such as stairs. Some of it may seem familiar:

```
241   pmove.velocity[2] = 0;
242   if (!pmove.velocity[0] && !pmove.velocity[1] &&
    ↪  !pmove.velocity[2])
243     return;
244
245   // first try just moving to the destination
```

```
246   dest[0] = pmove.origin[0] +
    ↪  pmove.velocity[0]*frametime;
247   dest[1] = pmove.origin[1] +
    ↪  pmove.velocity[1]*frametime;
248   dest[2] = pmove.origin[2];
249
250   // first try moving directly to the next spot
251   VectorCopy (dest, start);
252   trace = PM_PlayerMove (pmove.origin, dest);
253   if (trace.fraction == 1)
254   {
255     VectorCopy (trace.endpos, pmove.origin);
256     return;
257   }
```

First, we set the z-component of our velocity to 0, and then check if we have a non-zero velocity in the xy-plane. If we are moving extremely slow, we do not need to move at all, and so we return. Otherwise, we continue with setting the vector `dest` to be where we will end up with our current velocity after the current frame. If our trace does not hit anything from our position to destination we can move the whole distance, and so we set our origin to the destination.

```
259 // try sliding forward both on ground and up 16 pixels
260   // take the move that goes farthest
261   VectorCopy (pmove.origin, original);
262   VectorCopy (pmove.velocity, originalvel);
263
264   // slide move
265   PM_FlyMove ();
266
267   VectorCopy (pmove.origin, down);
268   VectorCopy (pmove.velocity, downvel);
269
270   VectorCopy (original, pmove.origin);
271   VectorCopy (originalvel, pmove.velocity);
```

First, it should be noted that the comment here references the piece of code *following* this block of code, which is the height for if we can step up a stair height and move forward. For now, this block of code simply uses the pre-existing `PM_FlyMove` to take care of collisions that we encounter, which is mainly making sure we project our velocity properly with everything we collide with. Then, we just store the step that this takes. Then, we check stair height motion:

```
273 // move up a stair height
274   VectorCopy (pmove.origin, dest);
275   dest[2] += STEPSIZE;
```

```
276   trace = PM_PlayerMove (pmove.origin, dest);
277   if (!trace.startsolid && !trace.allsolid)
278   {
279     VectorCopy (trace.endpos, pmove.origin);
280   }
281
282 // slide move
283   PM_FlyMove ();
284
285 // press down the stepheight
286   VectorCopy (pmove.origin, dest);
287   dest[2] -= STEPSIZE;
288   trace = PM_PlayerMove (pmove.origin, dest);
289   if ( trace.plane.normal[2] < 0.7)
290     goto usedown;
291   if (!trace.startsolid && !trace.allsolid)
292   {
293     VectorCopy (trace.endpos, pmove.origin);
294   }
295   VectorCopy (pmove.origin, up);
```

This adds a stair height to our stored destination, and attempts to move us from our position to our destination offset vertically by a stair height. This takes care of moving horizontally into the stairs. Now that we end up somewhere either directly on the stair or slightly above it, we want to move our player downwards to rest on top of the stairs. Thus, we once more do a trace down to determine how far down we have to go until we land on the step. If we land on a valid ground surface, we skip ahead to later in the code. Otherwise, so long as we do not start or end inside a solid, we move to our new destination with the vertical offset included.

```
297 // decide which one went farther
298   downdist = (down[0] - original[0])*(down[0] -
    ↪  original[0])
299     + (down[1] - original[1])*(down[1] - original[1]);
300   updist = (up[0] - original[0])*(up[0] - original[0])
301     + (up[1] - original[1])*(up[1] - original[1]);
302
303   if (downdist > updist)
304   {
305 usedown:
306     VectorCopy (down, pmove.origin);
307     VectorCopy (downvel, pmove.velocity);
308   } else // copy z value from slide move
309     pmove.velocity[2] = downvel[2];
```

This does exactly as commented. It checks if we went further using `PM_FlyMove`

or the stair height additional move, and we use whichever made us move further. After this, there is an additional comment relating to nudging the player, however there actually is not any code after this point.

15 PM_CatagorizePosition

The last check we have to do in our movement is a more complex check for if we are grounded or not, to be able to determine what functions to use when moving. This is the job of `PM_CatagorizePosition`, and yes they did spell Categorize incorrectly. There are many spelling errors in the code's comments as well, which is quite funny. Below is the implementation:

```
580 // if the player hull point one unit down is solid, the
    ↪  player
581 // is on ground
582
583 // see if standing on something solid
584   point[0] = pmove.origin[0];
585   point[1] = pmove.origin[1];
586   point[2] = pmove.origin[2] - 1;
587   if (pmove.velocity[2] > 180)
588   {
589     onground = -1;
590   }
591   else
592   {
593     tr = PM_PlayerMove (pmove.origin, point);
594     if ( tr.plane.normal[2] < 0.7)
595       onground = -1;  // too steep
596     else
597       onground = tr.ent;
598     if (onground != -1)
599     {
600       pmove.waterjumptime = 0;
601       if (!tr.startsolid && !tr.allsolid)
602         VectorCopy (tr.endpos, pmove.origin);
603     }
604
605     // standing on an entity other than the world
606     if (tr.ent > 0)
607     {
608       pmove.touchindex[pmove.numtouch] = tr.ent;
609       pmove.numtouch++;
610     }
611   }
```

Essentially, we first store our position into the vector `point`, and subtract 1 unit from our height, which will be used as a sort of destination for our trace to use so we can check for planes below us. If we have an upward component in our velocity that exceeds 180 units per second, then we are considered airborne. This has the consequences of allowing us to slide up ramps that are less than 45° from the ground (the ramps we would normally just walk up). When we hit the ramp, and our velocity is projected along the ramp, some of our horizontal velocity is given to our vertical component during the projection, and provided our velocity is large enough, we can achieve an upwards component of greater than 180 units per second.

Next, we check if we collide with anything 1 unit below us. If we do, we check if it is too steep. If it is sloped less than 45°, then we consider ourselves grounded. Otherwise, we are airborne. Lastly, if we are found to be standing on an entity, we record what entity we are standing on, as this is used to trigger some events such as waking up enemies if we stand on their head.

16 NudgePosition

This function serves the purpose of quite literally nudging the player out of a solid object if they are stuck in it. It utilizes a function `PM_TestPlayerPosition` which essentially just checks if the collision hull intersects some other entity in the world such as world geometry. Such a function is built on a fair amount of helper functions that do a lot of very funny math and if I opted to include it all here to explain precisely how it works this paper would be twice as long. It suffices for our purposes to know that all of this just checks if we intersect something. With that in mind, here is the implementation for nudging players out of solids:

```
742   VectorCopy (pmove.origin, base);
743
744   for (i=0 ; i<3 ; i++)
745     pmove.origin[i] = ((int)(pmove.origin[i]*8)) * 0.125;
746 //  pmove.origin[2] += 0.124;
747
748 //  if (pmove.dead)
749 //    return;    // might be a squished point, so don'y
    ↪  bother
750 //  if (PM_TestPlayerPosition (pmove.origin) )
751 //    return;
752
753   for (z=0 ; z<=2 ; z++)
754   {
755     for (x=0 ; x<=2 ; x++)
756     {
757       for (y=0 ; y<=2 ; y++)
```

```
758         {
759           pmove.origin[0] = base[0] + (sign[x] * 1.0/8);
760           pmove.origin[1] = base[1] + (sign[y] * 1.0/8);
761           pmove.origin[2] = base[2] + (sign[z] * 1.0/8);
762           if (PM_TestPlayerPosition (pmove.origin))
763             return;
764         }
765       }
766     }
767     VectorCopy (base, pmove.origin);
768   //  Con_DPrintf ("NudgePosition: stuck\n");
769   }
```

After setting up some initial values for our tests, we essentially loop through each coordinate axis and try adding or subtracting some small delta value, approximately $\frac{1}{8}$ of a unit, until we can get a combination of nudges forward or backward on each axis that gives us a valid position. The values x, y, z in the `for` loops determine if the array `sign` gives us 0, -1, or 1, which gives us either no nudge, a nudge by $\frac{1}{8}$-th of a unit forward, or backward on that axis. Thus we can achieve every possible combination of nudges with these nested `for` loops.

17 Tying It All Together: MovePlayer

Now that we have all of the necessary functions to handle user input, velocity, acceleration, collisions, ground state, and more, we can finally tie them all together into one function that handles the order of execution of each and passes information between them. This is the `MovePlayer` function:

```
862 void PlayerMove (void)
863 {
864   frametime = pmove.cmd.msec * 0.001;
865   pmove.numtouch = 0;
866
867   AngleVectors (pmove.angles, forward, right, up);
868
869   if (pmove.spectator)
870   {
871     SpectatorMove ();
872     return;
873   }
874
875   NudgePosition ();
876
877   // take angles directly from command
878   VectorCopy (pmove.cmd.angles, pmove.angles);
879
```

```
880   // set onground, watertype, and waterlevel
881   PM_CatagorizePosition ();
882
883   if (waterlevel == 2)
884     CheckWaterJump ();
885
886   if (pmove.velocity[2] < 0)
887     pmove.waterjumptime = 0;
888
889   if (pmove.cmd.buttons & BUTTON_JUMP)
890     JumpButton ();
891   else
892     pmove.oldbuttons &= ~BUTTON_JUMP;
893
894   PM_Friction ();
895
896   if (waterlevel >= 2)
897     PM_WaterMove ();
898   else
899     PM_AirMove ();
900
901   // set onground, watertype, and waterlevel for final
      ↪  spot
902   PM_CatagorizePosition ();
903 }
```

We begin by orienting our space to be relative to our camera's perspective using the `AngleVectors` function, which is discussed in the math function reference section. This just makes sure that user input remains relative to the direction we are looking in. Then, we check if we are dead or in spectator, to either not move (dead) or use the spectator movement function, which I have chosen to exclude (it is fairly straight forward movement without any collision checks). Next, we call `NudgePosition` in case we managed to get stuck inside an object on the last frame. We then check our ground state, and consider if we can jump or not based on user input and ground state. I have chosen not to include the function `JumpButton` as a majority of it is water-related, and the only significant portion of it simply adds 270 units per second upward to the player's velocity, and sets us to not be grounded. After this, we apply friction, and execute `PM_AirMove` to figure out our intended direction to move in. It is within this function that we determine if we should use the air or grounded acceleration and movement functions, and thus, we do not need to call it here. Finally, we determine if our new position is grounded or not via a call to `PM_CatagorizePosition`. With all of this, we have successfully moved some distance over the course of 1 singular frame.

18 Closing Remarks

I have chosen to ignore several functions included in the `pmove.c` file as the main purpose of this document is about normal movement. I have excluded water movement and spectator movement, and where there was a significant portion of code relating to these, I have elected to remove them for sake of clarity and page count. As well, I have chosen to excluded the `JumpButton` function as well, as a huge majority of its code is in relation to water, and the main few lines were trivial.

Part IV
Brief Reference for Math Functions Used

19 About This Section

Throughout this code, we make extensive use of various helper math functions. I have decided to include their code, along with a brief description of what it does both mathematically and geometrically where applicable. These functions are written in the order they appear in the `pmove.c` file, and all can be found within the `mathlib.c` file in the same directory.

20 DotProduct

```
335 vec_t _DotProduct (vec3_t v1, vec3_t v2)
336 {
337   return v1[0]*v2[0] + v1[1]*v2[1] + v1[2]*v2[2];
338 }
339
```

Quite simply, this returns a scalar that is the sum of the component-wise multiplication of two vectors. Similarly, this can be calculated with $\| \vec{a} \| \| \vec{b} \| \cos\theta$, where $\vec{a}, \vec{b} \in \mathbb{R}^2$ and θ is the angle between them. It is both faster and more general to higher dimension spaces or spaces with no general notion of angles to compute the inner product, which is essentially the same thing, but does not involve cosines or angles. This is $\sum_{i=1}^{n} a_i b_i$, where $\vec{a}, \vec{b} \in \mathbb{R}^n$ and n is the dimension of the space. This formula is what is explicitly written here.

The notable properties of the dot product are that when the angle between them is 90°, the dot product is 0 (or more generally we say that the vectors are *orthogonal*.) As well, if the two vectors point greater than 90° from each other, the dot product is negative. We can then use this check of whether the result is negative, 0, or positive to determine a lot of key information about the direction between two vectors without ever needing to explicitly find angles.

21 VectorCopy

```
354 void _VectorCopy (vec3_t in, vec3_t out)
355 {
356   out[0] = in[0];
357   out[1] = in[1];
```

```
358   out[2] = in[2];
359 }
```

This quite literally copies the contents of the vector `in` into the vector `out`. Nothing much more to say.

22 CrossProduct

```
361 void CrossProduct (vec3_t v1, vec3_t v2, vec3_t cross)
362 {
363   cross[0] = v1[1]*v2[2] - v1[2]*v2[1];
364   cross[1] = v1[2]*v2[0] - v1[0]*v2[2];
365   cross[2] = v1[0]*v2[1] - v1[1]*v2[0];
366 }
```

This product can also be expressed with two different formulas. The one used here does not involve angles or sine, and is thus a lot nicer to compute. A slightly visually nicer way of seeing this computation is:

$$\begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{bmatrix}$$

for some two input vectors $\vec{a}, \vec{b} \in \mathbb{R}^n$, which yields a vector $\vec{c} \in \mathbb{R}^n$. Geometrically, this outputs a vector orthogonal to both input vectors, with a length equal to the area given by the parallelogram with both input vectors for sides. Thus, if they point in the same direction or opposite, this area would be 0, and so the cross product would be the zero vector. We only use this in `PM_FlyMove` to find a vector orthogonal to both planes we collide with.

23 VectorScale

```
409 void VectorScale (vec3_t in, vec_t scale, vec3_t out)
410 {
411   out[0] = in[0]*scale;
412   out[1] = in[1]*scale;
413   out[2] = in[2]*scale;
414 }
```

This simply takes in a vector and a scalar, and returns a vector that is the input scaled by the scalar.

24 VectorNormalize

```
383 float VectorNormalize (vec3_t v)
384 {
```

```
385   float  length, ilength;
386
387   length = v[0]*v[0] + v[1]*v[1] + v[2]*v[2];
388   length = sqrt (length);    // FIXME
389
390   if (length)
391   {
392     ilength = 1/length;
393     v[0] *= ilength;
394     v[1] *= ilength;
395     v[2] *= ilength;
396   }
397
398   return length;
399
400 }
```

Normalizing a vector means to scale it so that it is of length 1. We can do this by first computing the length of our vector, and then scaling our vector by the reciprocal of the length. The first section that calculates `length` takes the dot product of the vector with itself, and then takes the square root of that length. The reason this works is because if we instead consider the formula that uses lengths and angles between vectors, mentioned previously for the dot product, a dot product of a vector with itself will have an angle of $0°$ between itself, and thus $\cos 0 = 1$, giving us $\|\vec{a}\|\|\vec{a}\| \cdot 1$, which is the length of $\vec{a}$ squared. Thus, square rooting the dot product of a vector with itself gives the length. Using the formula that does not require angles computes the same answer, and so we end up with the length squared without the use of lengths or angles. Now that we have our vector's length, we scale our vector by its reciprocal which is done component wise, and then the answer is returned.

25 VectorMA

```
327 void VectorMA (vec3_t veca, float scale, vec3_t vecb,
    ↪  vec3_t vecc)
328 {
329   vecc[0] = veca[0] + scale*vecb[0];
330   vecc[1] = veca[1] + scale*vecb[1];
331   vecc[2] = veca[2] + scale*vecb[2];
332 }
```

This odd function seems to just combine adding and scaling two vectors into one function. It takes in a vector `veca`, a scalar/float `scale`, and another vector `vecb`. It scales `vecb` by `scale`, and adds it to `veca`, then returns the result. There already exists functions for vector scaling, copying, and addition,

so the combination here seems rather arbitrary. As well, the name seems to suggest "Vector Magnitude", but seems to be derived from "Vector Multiplication Addition", which could be confusing if one was not familiar with what it did.

26 Length

```
370 vec_t Length(vec3_t v)
371 {
372   int    i;
373   float  length;
374 
375   length = 0;
376   for (i=0 ; i< 3 ; i++)
377     length += v[i]*v[i];
378   length = sqrt (length);    // FIXME
379 
380   return length;
381 }
```

As mentioned previously in `VectorNormalize`, we can find the length of a vector by taking the dot product of a vector with itself, and applying the square root to the result. This is exactly what this does, but it makes use of a `for` loop to do this instead of just manually computing it all in one line, which is an interesting difference. Then, we take the square root this result, and return it.

27 AngleVectors

```
290 void AngleVectors (vec3_t angles, vec3_t forward, vec3_t
    ↪  right, vec3_t up)
291 {
292   float    angle;
293   float    sr, sp, sy, cr, cp, cy;
294 
295   angle = angles[YAW] * (M_PI*2 / 360);
296   sy = sin(angle);
297   cy = cos(angle);
298   angle = angles[PITCH] * (M_PI*2 / 360);
299   sp = sin(angle);
300   cp = cos(angle);
301   angle = angles[ROLL] * (M_PI*2 / 360);
302   sr = sin(angle);
303   cr = cos(angle);
304 
305   forward[0] = cp*cy;
306   forward[1] = cp*sy;
```

```
307     forward[2] = -sp;
308     right[0] = (-1*sr*sp*cy+-1*cr*-sy);
309     right[1] = (-1*sr*sp*sy+-1*cr*cy);
310     right[2] = -1*sr*cp;
311     up[0] = (cr*sp*cy+-sr*-sy);
312     up[1] = (cr*sp*sy+-sr*cy);
313     up[2] = cr*cp;
314 }
```

This gives the column vectors of a 3D rotation matrix, as the columns of such a matrix are the axes that form the rotated space. Thus, the first column gives a vector that points in the 'forward' direction in this rotated space, the second gives an 'upward' vector, and so on. Noticeably, the second column has a -1 factored in to each line. It is visually nicer to see this as the matrix:

$$R = \begin{bmatrix} \cos y \cos p & -1(\cos y \sin p \sin r - \sin y \cos r) & \cos y \sin p \cos r + \sin y \sin r \\ \sin y \cos p & -1(\sin y \sin p \sin r + \cos y \cos r) & \sin y \sin p \cos r - \cos y \sin r \\ -\sin p & -1(\cos p \sin r) & \cos p \cos r \end{bmatrix}$$

With p, y, r as some angle in radians for pitch, yaw and roll. If you were to apply this transformation to the standard unit vectors for $\mathbb{R}^3$ ($\vec{e}_1, \vec{e}_2, \vec{e}_3$, or if you prefer, $\hat{i}, \hat{j}, \hat{k}$), you would obtain the columns of this matrix, which is another way of thinking about how we get these specific vectors. Thus we do not need to compute a matrix and apply it to the unit vectors, we can directly skip to storing the columns of this matrix.

Part V
Special Thanks & Works Cited

I would like to thank Waldo, Lumia, PeenScreeker, ficool2, BlueRaven, hex, ILDPRUT, firestabber, Frid, and CaWU for help and sanity checking my understanding of Quake's engine. Your patience and expertise is appreciated greatly.

References

[1] id Software, *Quake*, (1996), GitHub repository, pmove.c

[2] ILDPRUT, *Review*, (2022), Dropbox, review.pdf

[3] Jelvan1, *q3df_ doc*, (2022), GitHub repository, main.pdf

[4] Winterdev, *EPA: Collision Response Algorithm in 2D/3D*, (2020), Winter's Blog

[5] Winterdev, *GJK: Collision Detection Algorithm in 2D/3D*, (2020), Winter's Blog