2D case

Using the algorithm developed above, which, for the initial position $(0, b)$, $0 < b < 1$ and for any initial velocity returns the integer coordinates of the obstacle of the first collision, we make the function `collisions` that calculates the trajectory of the particle in the 2D plane. In order to calculate the exact collision point and the velocity after the collision, we need to use the classical collision function `collide`.

```
function collide(q, p, x, y, vx, vy, r)
      r0 = [x, y]
      v0 = [vx, vy]
      v0 /= norm(v0)
      R = [q, p]
      crossz(x, y) = x[1]*y[2] - x[2]*y[1]
      discr = norm(v0)^2*r^2 - (crossz(v0, r0-R))^2
      t1 = (-dot(v0, r0-R) - sqrt(discr))/norm(v0)^2
      N0 = r0 + v0*t1 - R
      N = N0/norm(N0)
      v1 = v0 - 2*dot(v0, N)*N
      r1 = r0 + v0*t1
      return r1[1], r1[2], v1[1], v1[2]
end
```

At every step of the trajectory, we create an array of the integer corners of the square where the particle is, and then, given the line of motion of the particle, we determine whether it will leave the square or experience a collision in the same square; also, we make sure that the corner which is behind the particle (i.e., is at the direction opposite to the motion) does not get counted as a collision. There is a small problem at the first step: in order for the function to work, we put a dummy point into the array of the coordinates of the collisions (usually a point very far from the origin), and at the first step it may be counted as a collision, so we delete it. If the particle exits the square, we determine the side through which it exits: if it is one of the two vertical sides, we simply apply the efficient algorithm and obtain the place of the next collision; if it is one of the two horizontal sides, we rotate the coordinates, apply the efficient algorithm, and rotate the coordinates back, thus obtaining the place of the next collision. If the particle does not exit the square, we use the classical collision function to determine the new point of collision and the new velocity.

Below are the functions `frac`, `efficient_algorithm` and `first_collision`, which comprise the implementation of the efficient algorithm, and the function `collisions` which calculates the trajectory.

```
function frac(x, epsilon)
    h1, h2 = 1, 0
    k1, k2 = 0, 1
    b = x
    while abs(k1*x - h1) > epsilon
        a = ifloor(b)
        h1, h2 = a*h1 + h2, h1
        k1, k2 = a*k1 + k2, k1
        b = 1/(b - a)
    end
    return k1, h1
end


function efficient_algorithm(m, b, epsilon)
```

```
    kn = 0
    while b > epsilon && 1 - b > epsilon
        if b < 0.5
            (q, p) = frac(m, 2b)
        else
            (q, p) = frac(m, 2*(1 - b))
        end
        b = mod(m*q + b, 1)
        kn += q
    end
    q = kn
    p = ifloor(m*q) + 1
    return (q, p)
end


function first_collision(x, y, vx, vy, delta)
    # Normalize velocity if it wasn't normalized
    v = sqrt(vx^2 + vy^2)
    vx1 = vx/v
    vy1 = vy/v
    vx = vx1
    vy = vy1
    m = vy/vx
    b = y - m*x

    if b > delta && 1 - b > delta

        if vx > 0 && vy > 0
            (q, p) = efficient_algorithm(m, b, delta)
            p = ifloor(m*q) + 1
        elseif vx < 0 && vy > 0
            m = -m
            (q, p) = efficient_algorithm(m, b, delta)
            p = ifloor(m*q) + 1
            m = -m
            q = -q
        elseif vx < 0 && vy < 0
            b = 1 - b
            (q, p) = efficient_algorithm(m, b, delta)
            b = 1 - b
            p = -ifloor(m*q)
            q = -q
        elseif vx > 0 && vy < 0
            b = 1 - b
            m = -m
            (q, p) = efficient_algorithm(m, b, delta)
            b = 1 - b
            p = -ifloor(m*q)
        end

    # Added code for cases when b or 1-b < delta
    # The problem with efficient_algorithm() is that whenever b<delta or
1-b<delta, it always outputs (0, 1): even though the speed is negative in both
directions (starting point is (0, b)) and the first collision is (-2, -2), the
algorithm outputs (0, 1), which is in the other direction.
    # We have to make special cases to fix it

    elseif b <= delta
```

```
            # Four situations possible: leaves through top, leaves through 
right/left, hits (1, 1)/(-1, 1), hits (0, 1)
            r = abs(delta*vx)
            # Three critical values of m, ascending
            m1 = (b - 1 + r*sqrt(2 + b*(b - 2) - r^2))/(r^2 - 1)
            m2 = (r - (b - 1)*sqrt(2 + b*(b - 2) - r^2))/((b - 1)*r + sqrt(2 + b*(b 
- 2) - r^2))
            m3 = sqrt(((b - 1)/r)^2 - 1)

            # Leaves through top
            if abs(m) < m3 && abs(m) > m2
                  if vx > 0 && vy > 0
                        (q, p) = efficient_algorithm(1/m, (1-b)/m, delta/m)
                        q, p = p, q
                        p += 1
                  elseif vx < 0 && vy > 0
                        m = -m
                        (q, p) = efficient_algorithm(1/m, (1-b)/m, delta/m)
                        q, p = p, q
                        p += 1
                        q = -q
                  elseif vy < 0
                        q = 0
                        p = 0
                  end
            elseif abs(m) > m3
                  q = 0
                  p = 1
            elseif m < m2 && m > m1
                  q, p = 1, 1
            elseif m > -m2 && m < -m1
                  q, p = -1, 1
            elseif abs(m) < m1
                  if vx > 0 && vy > 0
                        (q, p) = efficient_algorithm(m, m + b, delta)
                        q += 1
                  elseif vx < 0 && vy > 0
                        m = -m
                        (q, p) = efficient_algorithm(m, m + b, delta)
                        q += 1
                        q = -q
                  elseif vy < 0
                        q = 0
                        p = 0
                  end
            end

      elseif 1 - b <= delta

            # It is just transformation y -> 1 - y, vy -> -vy and the previous 
situation applies

            m = -m
            b = 1 - b
            vy = -vy

            r = abs(delta*vx)
            # Three critical values of m, ascending
            m1 = (b - 1 + r*sqrt(2 + b*(b - 2) - r^2))/(r^2 - 1)
```

```
            m2 = (r - (b - 1)*sqrt(2 + b*(b - 2) - r^2))/((b - 1)*r + sqrt(2 + b*(b
- 2) - r^2))
            m3 = sqrt(((b - 1)/r)^2 - 1)

            # Leaves through top
            if abs(m) < m3 && abs(m) > m2
                  if vx > 0 && vy > 0
                        (q, p) = efficient_algorithm(1/m, (1-b)/m, delta/m)
                        q, p = p, q
                        p += 1
                  elseif vx < 0 && vy > 0
                        m = -m
                        (q, p) = efficient_algorithm(1/m, (1-b)/m, delta/m)
                        q, p = p, q
                        p += 1
                        q = -q
                  elseif vy < 0
                        q = 0
                        p = 0
                  end
            elseif abs(m) > m3
                  q = 0
                  p = 1
            elseif m < m2 && m > m1
                  q, p = 1, 1
            elseif m > -m2 && m < -m1
                  q, p = -1, 1
            elseif abs(m) < m1
                  if vx > 0 && vy > 0
                        (q, p) = efficient_algorithm(m, m + b, delta)
                        q += 1
                  elseif vx < 0 && vy > 0
                        m = -m
                        (q, p) = efficient_algorithm(m, m + b, delta)
                        q += 1
                        q = -q
                  elseif vy < 0
                        q = 0
                        p = 0
                  end
            end

            p = 1 - p

      end
      return q, p
end


dist_point_line(x, y, k, b) = abs(y - k*x - b)/sqrt(k^2 + 1)


function collisions(x, y, vx, vy, r, maxsteps, prec::Integer=64)
      set_bigfloat_precision(prec)
      x = BigFloat("$x"); y = BigFloat("$y"); vx = BigFloat("$vx"); vy =
BigFloat("$vy"); r = BigFloat("$r");

      # Normalize velocity if it wasn't normalized
      v = sqrt(vx^2 + vy^2)
```

```
      vx1 = vx/v
      vy1 = vy/v
      vx = vx1
      vy = vy1

      steps = 1
      places = Vector{BigInt}[]
      coords = Vector{BigFloat}[]
      speeds = Vector{BigFloat}[]
      # Push a dummy place to "places" - it cannot be empty for array_corners; 
also, it can't be near [x, y] and it can't be NaN or Inf
      push!(places, [-10^9, -10^9])

      push!(coords, [x, y])
      push!(speeds, [vx, vy])

      while steps <= maxsteps

            # Will the particle exit the square?
            n = ifloor(x)
            m = ifloor(y)
            k = vy/vx
            b = - k*x + y

            # If exits, not counting the obstacle that has just experienced 
collision (of course distance to it < r)
            # Make an array of corners and mark the corner where the collision just 
happened (which is the last item in "places" array)
            array_corners = Array{Int, 1}[]
            push!(array_corners, [n, m], [n, m+1], [n+1, m], [n+1, m+1])

            d(i) = dist_point_line(array_corners[i][1], array_corners[i][2], k, b)

            j = 0
            for i = 1:length(array_corners)
                  if array_corners[i] == places[length(places)]
                        j = i
                  end
            end


            # Array of the rest of the corners (3 other which may or may not 
experience collision); does not work at 1st step
            # The first two numbers are the corner coords, and the third is 
true/false (1/0) whether it leaves square
            array_rest_corners = Array{Int, 1}[]
            for i = 1:4
                  if i != j
                        push!(array_rest_corners, [array_corners[i], !(d(i) < r && 
dot([vx, vy], array_corners[i] - [x, y]) > 0)]) # The dot() added to prevent 
counting a backward ball
                  end
            end

            # There is a difficulty: at the first step, we have a dummy place in 
places, which is obviously not the previous collision, and the corner which is 
behind the initial position of the particle doesn't get deleted and the algorithm 
may mistakenly count it as the first collision. We have to detect and delete it.
            # 1st step: delete the obstacle that is the closest backwards
```

```
        # Determine through which wall it would exit if moved backwards
        # Times to each wall (vertical, horizontal)
        if steps == 1

              tv1 = (n - x)/vx
              tv2 = (n - x + 1)/vx
              th1 = (m - y)/vy
              th2 = (m - y + 1)/vy

              array_times_back = BigFloat[]
              push!(array_times_back, tv1, tv2, th1, th2)

              # Extract the maximum negative time value (closest wall
backwards)
              maxneg = -Inf
              number = 0
              for i = 1:length(array_times_back)
                    if array_times_back[i] > maxneg && array_times_back[i] < 0
                          maxneg = array_times_back[i]
                          number = i
                    end
              end
              # wall number: 1 = left, 2 = right, 3 = bottom, 4 = top
              # corner number: 1 = bottom left, 2 = top left, 3 = bottom right,
4 = top right
              if number == 1
                    if k*n + b < m + 0.5 && dot([n, m] - [x, y], [vx, vy]) < 0
# Make sure that the time to the wrong ball is also negative
                          deleteat!(array_rest_corners, 1)
                    else
                          deleteat!(array_rest_corners, 2)
                    end
              elseif number == 2
                    if k*(n + 1) + b < m + 0.5 && dot([n + 1, m] - [x, y], [vx,
vy]) < 0
                          deleteat!(array_rest_corners, 3)
                    else
                          deleteat!(array_rest_corners, 4)
                    end
              elseif number == 3
                    if (m - b)/k < n + 0.5 && dot([n, m] - [x, y], [vx, vy]) <
0
                          deleteat!(array_rest_corners, 1)
                    else
                          deleteat!(array_rest_corners, 3)
                    end
              elseif number == 4
                    if (m + 1 - b)/k < n + 0.5 && dot([n, m + 1] - [x, y], [vx,
vy]) < 0
                          deleteat!(array_rest_corners, 2)
                    else
                          deleteat!(array_rest_corners, 4)
                    end
              end
        end

        # Whether the particle leaves square or collides in the same square
        leaves_square = array_rest_corners[1][3] == 1 && array_rest_corners[2]
[3] == 1 && array_rest_corners[3][3] == 1
```

```
            # If the particle exits the unit square without another collision
            if leaves_square

                  # determine through which wall it will exit
                  # times to each wall (vertical, horizontal)
                  tv1 = (n - x)/vx
                  tv2 = (n - x + 1)/vx
                  th1 = (m - y)/vy
                  th2 = (m - y + 1)/vy

                  array_times = BigFloat[]
                  push!(array_times, tv1, tv2, th1, th2)

                  # Extract the minimum positive time value
                  minpos = Inf
                  number = 0
                  for i = 1:length(array_times)
                        if array_times[i] < minpos && array_times[i] > 0
                              minpos = array_times[i]
                              number = i
                        end
                  end
                  # number: 1 = left, 2 = right, 3 = bottom, 4 = top

                  # If the walls are vertical, we don't need to rotate coordinates
for first_collision(). We just run first_collision(), get the next obstacle and
collide()
                  if number == 1 || number == 2

                        if number == 1
                              posx = n
                              posy = m
                              x1 = 0
                              y1 = y + k*(n - x) - m
                        elseif number == 2
                              posx = n + 1
                              posy = m
                              x1 = 0
                              y1 = y + k*(n + 1 - x) - m
                        end

                        q, p = first_collision(x1, y1, vx, vy, abs(r/vx))
                        push!(places, [q + posx, p + posy])

                        x, y, vx, vy = collide(q + posx, p + posy, x1 + posx, y1 +
posy, vx, vy, r)
                        push!(coords, [x, y])
                        push!(speeds, [vx, vy])
                        # And we obtain coords of collision and new velocity, and
then cycle continues

                  elseif number == 3 || number == 4
                        # Here we have to rotate

                        if number == 3
                              posx = m
                              posy = n
                              y1 = x + (m - y)/k - n
```

```
                    x1 = 0
                elseif number == 4
                    posx = m + 1
                    posy = n
                    y1 = x + (m + 1 - y)/k - n
                    x1 = 0
                end

                # Now the coordinates are rotated. In this rotated system,
find the coordinates of the next obstacle
                q, p = first_collision(x1, y1, vy, vx, abs(r/vy))
                # Record the coordinates rotated back
                push!(places, [p + posy, q + posx])
                # Find the next point of collision
                x, y, vx, vy = collide(p + posy, q + posx, y1 + posy, x1 +
posx, vx, vy, r)
                push!(coords, [x, y])
                push!(speeds, [vx, vy])

            end

        # Now what if the particle doesn't exit the square? Obtain where it
doesn't (coords of obstacle), collide there and continue cycle

        elseif array_rest_corners[1][3] == 0
            place = [array_rest_corners[1][1], array_rest_corners[1][2]]
            push!(places, place)
            x, y, vx, vy = collide(place[1], place[2], x, y, vx, vy, r)
            push!(coords, [x, y])
            push!(speeds, [vx, vy])
        elseif array_rest_corners[2][3] == 0
            place = [array_rest_corners[2][1], array_rest_corners[2][2]]
            push!(places, place)
            x, y, vx, vy = collide(place[1], place[2], x, y, vx, vy, r)
            push!(coords, [x, y])
            push!(speeds, [vx, vy])
        elseif array_rest_corners[3][3] == 0
            place = [array_rest_corners[3][1], array_rest_corners[3][2]]
            push!(places, place)
            x, y, vx, vy = collide(place[1], place[2], x, y, vx, vy, r)
            push!(coords, [x, y])
            push!(speeds, [vx, vy])
        end
        steps += 1
    end
    # Delete the dummy place at position 1
    deleteat!(places, 1)
    return places, coords, speeds
end
```

3D case

The calculation of the trajectory in the 3D case is based on the 2D projections. We estimate the time to the first collision in the xy-plane and the yz-plane, using the corresponding projections of the coordinates and velocities on these planes. If the times are not approximately equal, it means that there is no real collision in 3D space. We select the lesser of the two times and advance to the next 2D obstacle determined by the efficient algorithm, and so on until the times are approximately equal. At that moment we obtain the 3D "candidate" for the valid collision. Then, using the classical 3D function `collide3d`, which calculates the coordinates of the collision, given the initial conditions of the particle, the coordinates of the center of the obstacle and its radius (and which outputs `false` when there is no collision) we determine if the collision is real. If it is, we record the obstacle, the coordinates of the collision and the new velocity and go to the next step. If it is not, we keep going with the same velocity and repeat this step.

The function measuring the approximate time to the circle (2D) may be expressed in a straightforward formula dividing the distance between the two points and the speed, which involves square roots. To make it slightly more efficient, we developed another method which avoids square roots and employs only the four arithmetic operations and elementary functions such as sign, absolute value and maximum/minimum. The idea is as follows. We measure the times of the displacement to the integer x- and y-coordinates (or the corresponding two coordinates on the plane; here we denote them x and y without the loss of generality) of the center of the circle. From Fig. 1 we can see that if the angle is about $\pi/4$ (or a multiple of it, not including 0), then these times are not very different from each other.

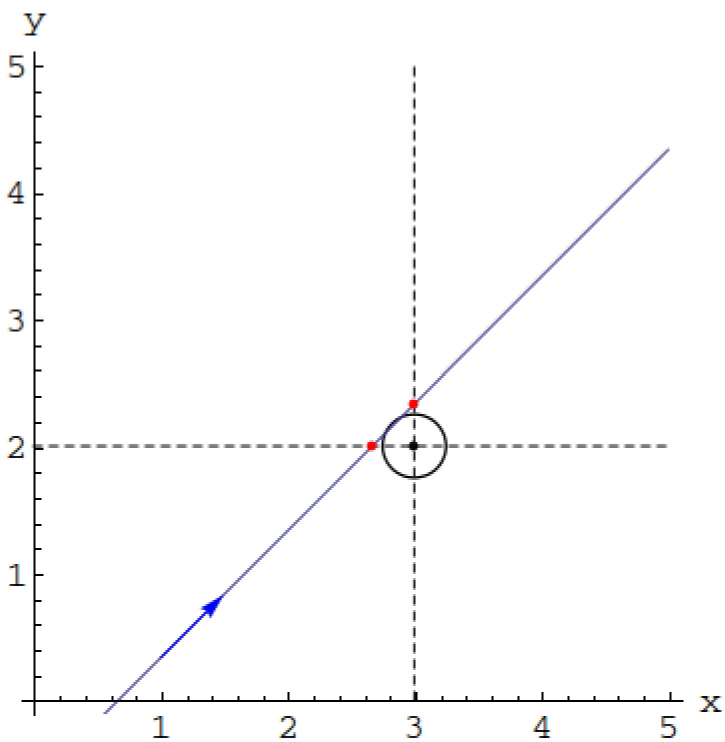

Fig. 1. Times to the integer coordinates of the circle in the case of the velocity angle close to $\pi/4$.

However, when the angle is close to $\pi/2$ (or a multiple of it, including 0), then the two times can be significantly different (see Fig. 2 below).

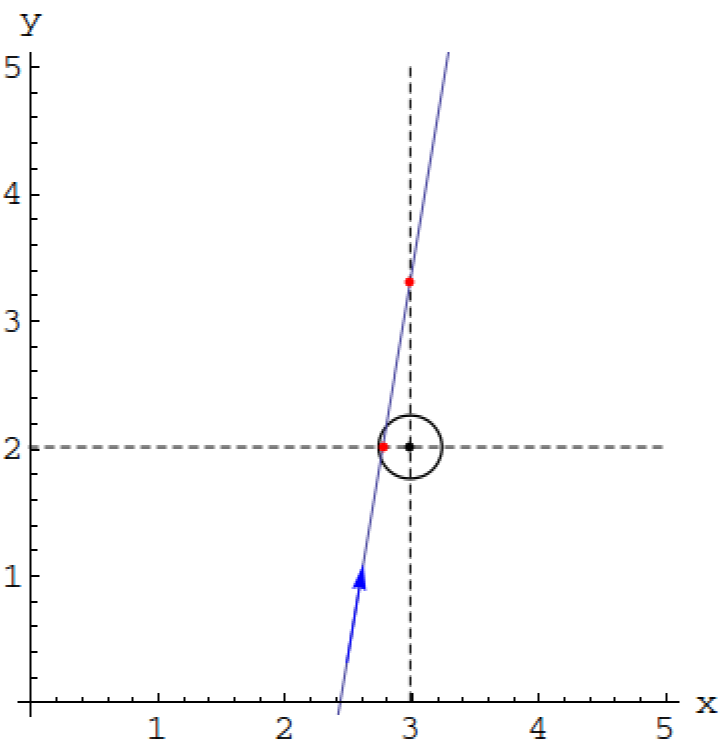

Fig. 2. Times to the integer coordinates of the circle in the case of the velocity angle close to π/2.

Moreover, whether the greater or the lesser time of the two is closer to the approximate time to the circle depends on whether the velocity vector has a greater angle $\varphi = \mathrm{arctg}(v_y/v_x)$ than the vector from the initial point to the center of the circle. If it is greater (as in Fig. 1 or Fig. 2), then, in the case presented in Fig. 2 the minimum of the two times corresponds to the approximate value. If the velocity angle is less than that of the vector to the center (for example, if the trajectory in Fig. 2 had the same slope but lesser y-intercept, i.e., would go "under" the circle), then the maximum of the two times would occur near the circle and thus correspond to the approximate time. The condition on the two angles can be found measuring the sign of the z-component of the cross product of the velocity and the vector to the center.

The first condition (whether the angle is close to a multiple of π/2 and which one – basically, in which of the 8 octants of the plane separated by the multiples of π/4 the angle is) can be found simply by evaluating the sign of the product $v_x v_y$ and whose absolute value (of v_x and v_y) is greater. By employing the two conditions, the approximate time is found as the maximum or the minimum of the two times, depending on the conditions.

This function `time_to_circle` is presented below in the function `collisions3d_time`, which implements the technique described in the first paragraph of this section. We also use the classical functions `collide3d`, to calculate the place where the particle will collide if the obstacle has center *x2* and radius *r*, and the particle has velocity *v* and initial position *x1*, and `v_new`, to calculate the velocity after the collision at the point *x1*, where *x2* is the center of the sphere.

```
function collide3d(x1, x2, v, r)
    b = dot(x1 - x2, v)/norm(v)^2
    c = norm(x1 - x2)^2 - r^2
    if b^2 - c < 0   # if there is no collision, return false
        return false
```

```
    end
    t = -b - sqrt(b^2 - c)
    x = v*t + x1
    return x
end


function v_new(x1, x2, v)
      n = x1 - x2
      n /= norm(n)
      vn = dot(n, v)*n
      v -= 2vn
      v /= norm(v)
      return v
end


function collisions3d_time(x, v, r, maxsteps, prec::Integer=64)
      set_bigfloat_precision(prec)

      x = big(x); v = big(v); r = big(r)
      v /= norm(v)

      places = Vector{BigInt}[]
      coords = Vector{BigFloat}[]
      speeds = Vector{BigFloat}[]

      approx_equal(x, y) = abs(x - y) < 0.4 # Was 0.3; setting it to 0.2 leads to
errors for r = 0.1; set 0.4 for new time_to_circle to work

      steps = 1

      first(x, v, d1, d2, r, prec) = collisions(x[d1], x[d2], v[d1], v[d2], r, 1,
prec)[1][1]

      # Supposedly efficient function time_to_circle without sqrt
      function time_to_circle(x, v, coord1, coord2, d1, d2)
          p = (coord1 - x[d1])*v[d2] - (coord2 - x[d2])*v[d1]
          tx = abs((coord1 - x[d1])/v[d1])
          ty = abs((coord2 - x[d2])/v[d2])
          if (sign(v[d1]*v[d2]) == 1 && abs(v[d1]) > abs(v[d2])) ||
(sign(v[d1]*v[d2]) == -1 && abs(v[d1]) < abs(v[d2]))
              if p > 0
                  return max(tx, ty)
              else
                  return min(tx, ty)
              end
          elseif (sign(v[d1]*v[d2]) == 1 && abs(v[d1]) < abs(v[d2])) ||
(sign(v[d1]*v[d2]) == -1 && abs(v[d1]) > abs(v[d2]))
              if p > 0
                  return min(tx, ty)
              else
                  return max(tx, ty)
              end
          end
      end

    while steps <= maxsteps
```

```
		x1, y1 = first(x, v, 1, 2, r, prec)
		y2, z2 = first(x, v, 2, 3, r, prec)

		t1 = time_to_circle(x, v, x1, y1, 1, 2)
		t2 = time_to_circle(x, v, y2, z2, 2, 3)

		int_steps = 0

		while !approx_equal(t1, t2)
			t = min(t1, t2)
			x_ahead = x + v*(t + 0.2)

			if t == t1
				x1, y1 = first(x_ahead, v, 1, 2, r, prec) #x, y
				t1 = time_to_circle(x, v, x1, y1, 1, 2)

			elseif t == t2
			y2, z2 = first(x_ahead, v, 2, 3, r, prec) #y, z
				t2 = time_to_circle(x, v, y2, z2, 2, 3)

			end

			int_steps += 1
		end

		ball = [x1, y1, z2]

		x_new = collide3d(x, ball, v, r)

		if x_new != false
			# Definitely a collision
			v = v_new(x_new, ball, v)
			x = x_new
			push!(places, ball)
			push!(coords, x)
			push!(speeds, v)
			steps += 1
		# If x_new returns false, continue moving from the farthest point
		else
			t = min(t1, t2)
			x += v*(t + 0.01)

		end

	end

	return places, coords, speeds
end
```