Ray tracing of spheres and capsules by calculus calculations

By Christopher Settles

# Introduction

My honors project this semester in Math223, Calculus with Analytical Geometry III, was to write a computer program that could draw a 3 dimensional world of predefined shapes (Spheres and capsules) by virtue of defining certain pixels on a screen and using calculus to decide if a certain shape gets draw or not on the screen. The program also allows the user to move forward and back, effectively zooming in and out on various shapes. This type of program is typically referred to as a game engine, as this is the basis to how video game graphics are rendered.

# C++ program – Drawing pixels at specific places

The computer allows the user to create a world of 3D objects, currently just consisting of spheres and capsules. The computer calculates the number of pixels it has to work with vertically and horizontally, and iterates through every pixel, treating each pixel as a ray. For every pixel (ray) the comptuer calls a Collideswith() function on every shape and if the function returns true, the pixel (ray) color changes to be the shape that it collided with.

EX:

for (int j = 0; j < width; j++) {

for (int k = 0; k < height; k++) {

Ray \*myRay = new Ray(Direction(userstartdirection.getx(), j - width / 2, -(k - height/2)), userstartpoint);

for (int i = 0; i < listofshapes.size(); i++) {

if (myRay->Collideswith(listofshapes[i]))

{

Set Pixel(mydc, j, k, listofshapes[i]->getColor());

}

}

}

}

Both Spheres and Capsules are set up as classes and what’s called polymorphic classes that derive from the abstract base class Shape. Shape has two pure virtual functions, bool Collideswith() and COLORREF getColor(). This means that if any new shape were to be programmed, all the programmer would have to do is to figure out how a ray will collide with it and store some color variable in it. Ray is also set up at a class and is defined as a direction (Direction) and a start point (Point). Direction and Point are also classes that are set up.

# Calculus – Collideswith() funciton

Essentially all of the calculus is performed in the collideswith() function. So as was mentioned earlier, every pixel (ray) on the screen does a calculation with every shape (Sphere or capsule) to see if that pixel (ray) collides with that shape. This is where the majority of the calculus lies.

The basic idea for developing the collideswith() algorithm for a sphere is that you want to calculate the minimum distance between the ray and the center point of the sphere. Then, if the minimum distance between the sphere and the ray is less than the radius of the sphere, the ray collides with that sphere and the corresponding pixel lights up. In order to have the computer calculate out that minimum distance for you, one should do the problem on paper but assume all numbers to be some type of variable. For example, instead of using the ray (1 + 2t, 5, 4t) one would have to write out (Ray.startpoint.x + Ray.Direction.x \* t, Ray.startpoint.y + Ray.Direction.y \* t, Ray.startpoint.z + Ray.Direction.z \* t).

To calculate the distance between two objects in 3 dimensions, one could use the distance formula

In our case, we want to calculate the distance between a ray and a point, so

Where ‘t’ is the time variable that allows the ray to move through space.

The next step is to distribute out these terms. The math here gets fairly ugly, so I will skip to the simplification of it.

If we set the entirety of the quantity that’s multiplied by to be instead (a), the entirety of the quantity that’s multiplied by to be instead (b), and the entirety of the quantity that’s multiplied by (not multiplied by t) to be instead (c), then we would get

For minimization purposes, we want to find the value of t that makes the value of d (the distance between the two objects) the lowest. To do this, we can take the derivative of the right side and set it equal to 0 to find the t value where the minimum distance is reached. When taking the derivative, we can ignore the square root sign because all we care about it ensuring that whatever is inside the square root sign is minimized. So

Taking the derivative we get

And solving for t we get

Which is the t value where the distance is minimized. Plugging that t value back into the distance formula, we get

Now since a, b, and c are all numbers that we have calculated previously, all we have to do is check to see if our d value is less than the radius of our sphere, and if it is, the pixel turns to the color of the sphere.

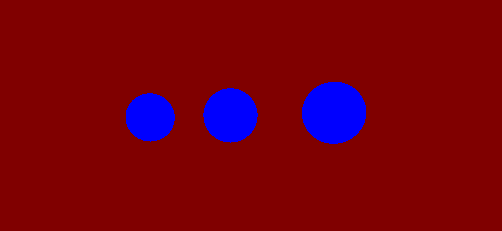


Figure 1: An example of 3 spheres getting draw in front of you. There all have the same radius, but they get closer to you as they come to the right.

This calculation is done for every pixel on the screen, where the pixels on the far left have a direction vector that points more towards the left and the pixels on the right side of the screen point towards the right. This logic also holds true for up and down pixels.

The capsule collideswith() function works similarly, however instead of just one t variable, you end up with two t variables (think a and b), one for the ray and one for the line segment that the capsule is made out of. As a result, you end up minimizing the distance function with both a and b, by taking the partial derivatives of the function with respect to both a and b. You should then end up with 2 equations with a and b in both of them. Then you go on to use Cramer’s rule with both of these equations to solve for both a and b. Then as long as your b variable (the time variable moving along the line segment) is within 0 and 1 (the length of the line segment), plugging in these a and b values to the original distance equation for a capsule will give you the minimum distance between a ray and a capsule. If the b variable is not minimized between 0 and 1 (the ray gets closest to the capsule at either of the end points of the capsule) then you would calculate the minimum distance the ray gets to that start point or end point of the capsule (whichever point is closest to the ray, the b value which one you are closer to). I did this by placing a sphere at each end of the capsule and calculating the collideswith() function on that sphere.

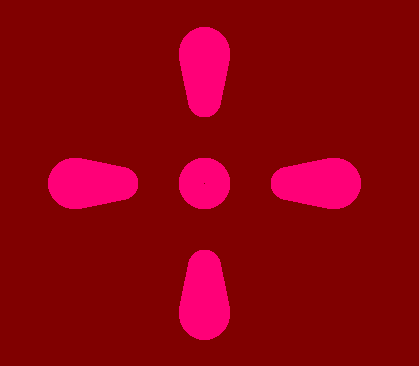


Figure 2 The 5 pillars above are all capsules that are directly in front of you and extent themselves away from you. Meaning: Assuming that you are looking in the x direction, the capsules do not travel at all in the y-z plane, they are only traveling away from you in the x plane. This image provides depth if you realize what it is. It is the equivalent to standing 5 pillars up on the ground in this fashion and looking down on those pillars. The center pillar looks like a circle because you are looking at it head on, but the other pillars you can see depth with because they get smaller as they travel away from you and larger as they travel towards you.

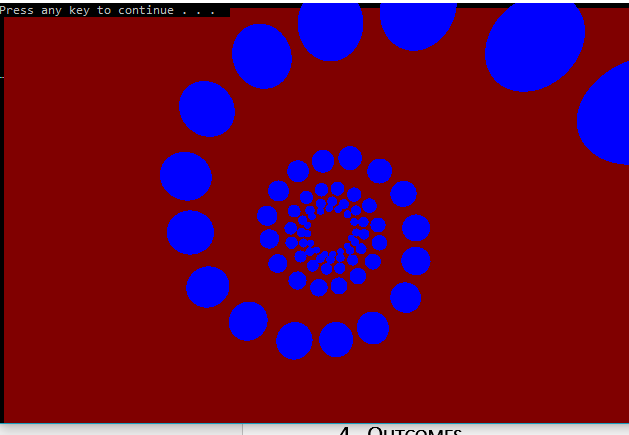


Figure 3: Falling through spirals

# Outcomes

I have to say, working on this project became addicting. You feel like the cool kid on the block when you realize that you just rendered 3d objects using nothing but things you learned in your calculus class. I’m actually very happy with 3 different aspects of this project:

1. The way I was able to engineer the software was perfectly correct; The Sphere and Capsule class are actually just derived from the Abstract Shape class. Using this, I was able to make a single array (vector) of shape pointers to both capsule and sphere objects. In the real world, this is how software engineering is really done. As a result of this design, programmers can add any kind of shape they want just as long as they overwrite the collideswith() and getColor() functions.
2. Figuring out how to draw 3d objects in an abstract 3d world that I create by just using a 2 dimensional screen. (Using calculus to find out if rays collide with objects, and more). I feel as though my math skills have improved as a result.
3. The cool graphics that I can create now!

One part of this project was to investigate the process through which a ray will collide with a cube, as done in the popular game of Minecraft, and other games. Chris Cunningham and I were unsuccessful at conjuring a way to develop the collideswith function for this type of object.

Source code for this project can be found at <https://github.com/ChristopherRSettles/RayTracingCalculus>

# Improvements

There are a few things that I wish to improve with this project, because obviously this isn’t a real game engine yet, but it has potential to be one.

Things to improve on:

* Speed
  + The computer has to do many calculations (For every pixel, it has to do a calculation for at least the number of objects in the world). When you start increasing the number of objects in the 3d world, the computer renders these objects must slower. Traditionally, this problem is resolved by sending the calculations that need to be ran to the graphics card and allowing the graphics card to do these types of calculations. So instead of the processor doing these calculations one after another, the graphics card can likely do all the calculations at the same time and it should be exponentially faster.
* Direction
  + Program it so that the user can look in whatever direction he/she wants to. Currently it is set up so the user can only look at things that are only in the –x direction. This type of math would involve the user of “quaternions" which are like a bigger badder version of complex numbers (a + bi + cj +dk instead of just a + bi)
* Lighting
  + Investigate the difficulty of having light sources added to the mix and if not so difficult then program it.
* Objects in front of each other
  + Ned to come up with logic that could tell you if one object is in front of another.
* Adjust field of view
* Have the canvas be somewhere other than the command prompt.
  + This one probably also ties in with switching it to the graphics card.