Graphical Ray Tracing: Made from scratch

Christopher Settles, Waubonsee Community College

# Abstract

This project addresses the graphical rendering of a world filled with mathematical shapes such as spheres, capsules, planes, and prisms by using a mathematical method called Ray Tracing. That is, the project approaches rendering 3D graphics mathematically by using ideas of vectors, optimization, and linear algebra techniques. The purpose was to apply tools from math disciplines towards a computer-graphics related application. With these mathematical tools, algorithms were developed that would allow for a computer to ray trace 3-dimensional objects onto a 2-dimensional screen.

# Implementation – C++ Abstract Base Class “Shape” and drawing pixels to the screen

All shapes defined in the world are polymorphic classes that derive from the abstract base class Shape. Shape has two pure virtual functions, boolean Collideswith() and COLORREF getColor(). This means that any new shape created that inherits from the abstract base class Shape would need to implement the methods getColor() and construct the algorithm for Collideswith(), which is a collision detection algorithm for that shape with a ray. A ray is set up as a class and is defined as a direction (Direction object) and a start point (Point object).

The program is given a screen to draw the world. Rays shoot radially out from the starting point at an imaginary plane (computer screen). Every ray has the same start point (some distance away from the screen) and travels in the direction of its corresponding pixel’s location on the screen. The program iterates through every pixel on the screen and generates a ray that belongs to that pixel.

# Calculus applied – Collision detection for Spheres and capsules

A sphere can be defined as a single point in R3 and a radius. The basic idea for developing the Collideswith() algorithm (Collision detection) for the sphere is to calculate the minimum distance that the ray gets to the center point of the sphere. (It is important to understand that the ray is constantly moving, and there will be only one time in which the ray is at its closest point to the sphere.) Since the ray position is a function of time, the distance function for the distance between the center of the sphere and the ray is also a function of time. Calculus can be used to minimize this function to calculate the minimum distance that the ray gets to the center of the sphere by finding the time value that minimizes the function. Then, if the minimum distance between the sphere and the ray is less than the radius of the sphere, it can be deduced that the ray collides with that sphere, and the pixel corresponding to that ray lights up to be the color of the sphere it collided with.

To calculate the distance between two objects in 3 dimensions, the distance formula could be utilized:

(plus the y and z components under the square root). Minimizing, we get

Where a is the quantity multiplied by t2, b is the quantity multiplied by t, and c is time independent in the distance function.

A capsule is constructed with a line segment and a sphere at each end of the segment. The capsule collision detection algorithm works similarly; however, instead of the distance being calculated between a ray and a point, the distance is calculated between a ray and the line segment that constructs the capsule.

The algorithm will first check to see if the ray collides with the capsule’s line segment, and if not the algorithm will check if the ray collides with either end point of the capsule. The distance function between the ray and the line segment is also calculated using the before mentioned distance formula, however, this time both the ray and the line have a position function that is dependent on their own time variable. As

a result, the distance function between the ray and the line contains two different time variables, and the minimization of this distance function involves the partial derivative with respect to both time variables. The set of equations

model how these new partially differentiated functions appear, with g, h, and c as constants. Cramer’s rule can be applied to now solve for a and b. Doing so, we get

for the final values of a and b. The algorithm then checks if a is between 0 and 1 (in this case, the minimum distance between the capsule and the ray is the minimum distance between the ray and the line). If a is not between 0 and 1, this means that the ray gets closest to the capsule at one of the capsule’s end points, so the Collideswith() function then just gets called on a sphere placed at the end of the line segment which the ray comes closest to.

# Linear algebra applied – Planes and Boxes

A plane can be defined as a start point and two vectors that create the subspace of R3 in which the vectors exist. Since the collision detection algorithm only wants to check to see if the ray collides with a limited version of the plane (a rectangle), the magnitude of each vector can be used to determine the length in which the plane extends in that direction. To detect if a ray collides with a plane, the algorithm asks “where” the ray would collide with the plane, if the plane did stretch infinitely. (This is safe to do as long as the algorithm first checks if the plane is parallel to the ray, and if so, returns false). The algorithm does this by setting the position function of the ray equal to the position function of the plane. Once the point of intersection between the plane and ray is found, the algorithm determines if the point of intersection is within the plane’s defined bounds, and if so, the ray collides with that plane.

A box is defined as a start point and vectors. The collision detection for the box simply creates with the box’s vectors, six planes that cover the six different faces of the box and then computes the collision detection on each plane.