A 2D raytracer for light and shadow, and field of vision calculations for 2D games

Nikolaj R. C.

2022 CE

Abstract

In this article, a Raytracing algorithm for fast and accurate light and shadow, and field of vision, calculations in 2 dimensions, intended for 2D games, is introduced. This algorithm does not take refraction, reflection or the wave-nature of light into account

No mathematical or computer-science education, should be required to understand this paper.

1 Fiat Lux

Say you have a decent 2D game, and now you want to make it look like an excellent game. Adding a dynamic lighting system is a good way of doing this, light is impressive, and will make the many different things making up your simulated world look like they belong together.

What is more, the same systems which can set up dynamic lighting can be used to calculate what the player, or a non player charcater, can see. After all, calculating what objects are in the direct line of sight from somewhere in the world is exactly the same as calculating what objects are directly lid by some objects.

This can, for instance, be used to set up a stealth system, or counter the greatest flaw I see with 2D gains: the players ability to see behind walls, by literally looking at the world from a perspective unknown to its inhabitants. By either completely blacking out anything not in the line of sight of the player, or by hiding key objects, such as npc's or key items, the player must actually explore the world, the same way their character would.

So how can we do this?

Well, really, if we want a physically accurate model we would need to solve Maxwells equations in whatever environment we are in, that can be done, which leads to some fascinating effects, such as how the very center of the shadow, of a circular object, has a tiny bright spot, which is exactly as bright as if the object wasn't even there. And if we really really wanted a physically accurate model, well we do know what the Hamiltonian for electromagnetic radiation in free space... But in a game, we don't need a physically accurate simulation, we need a good enough looking algorithm which can run fast.

One such good-enough approach is to say that light moves in straight lines (*rays*) from a point source, and can be blocked by any objects. This is arguably close to Newton's model of light. This is the model I wish to implement here.

2 The algorithm

2.1 Setting the stage

There are two kinds of objects which the algorithm needs to know about, *meshes* and *point sources*.

Point sources are single point sources sending *rays* of light out in all possible directions (or in some limited angle), and *rays* are straight lines starting at some point, and going in some direction.

Meshes are made up of points called *vertices* (singular *vertex*) connected to one another by straight , light blocking, edges.

It is simple to check if a particular ray intersects a particular edge in a mesh, and if yes, where. Using this, I have made a function `bool mesh2D::has_intersect(const vec2& A,const vec2& B)`, in my Mesh class, which returns true if a ray starting in the 2D point `A` and going through the point `B` intersects any edge in the mesh. If we want to know where a ray intersects the mesh, the function `bool get_intersect(const vec2& A,const vec2& B, vec2& Out, uint& V0_ID, uint& V1_ID , float& AB2)` still returns true if any intersection happens, and if so the intersection closest to `A` is written to `Out` and the function tell us that we hit the edge between vertex number `V0_ID` and `V1_ID` and `dist2` is now the distance from `A` to the intersection squared. (The `&` denotes a C++ reference, which means that any changes made to `Out`, sticks around after the function is done). A few examples of these functions is shown In Figure **??**.

OK actually these functions are not even remotely simple or easy to implement, especially if we want the code to run relatively fast. For instance, in my implementation I implemented Welzl's algorithm to calculate *bounding circles* around all meshes (See Figure **??**), so that we can skip checking all the edges of a mesh, if the ray does not come anywhere near the mesh, but implementing this doesn't really have anything to do with the main problem, so lets just start by assuming that we have these functions.

As illustrated in Figure **??**, the output is another mesh, formatted as a *triangle fan*, a list of points where the first point is the source, and the remaining points are used to built a number of triangles all with one vertex at the source. Anything inside these triangles is in light, and anything outside is in shadow. It is fairly easy to both render this triangle fan to create a light-map, which can be used for shading effects, or to test if a point is inside or outside the triangle fan.

2.2 Raycasting, the easy alternative

2.3 Issues

There are a number of rare situations, where the algorithm fails, these situations may be rare, but in a game where perhabs the player moves the lightsource around – or at the very least the player is a shadowcasting object – these rare errors will happen eventually.

When