Polar-Ray Traced Lighting

By Alexander S. Griffis

Abstract

Ray-tracing is a massive performance problem for game developers due to the mass amount of data that needs to be processed in order to achieve an adequate result. The use for ray-tracing varies from lighting to collisions to line of sight detection with enemies. On CPUs ray-tracing takes up a lot of clock cycles and stepping over textures, across arrays of tiles or through object space and looping through objects comes at a heavy cost of performance. Even with GPUs ray-tracing--where GPUs excel at texture sampling--is still performance heavy and leaves people with using more complex systems like SDFs (signed distance fields) to approximate ray-tracing. However there is a solution… Compute Shaders! Granted I've never used them, but you can simulate a compute shader within a fragment shader by treating the destination texture as a 2D array of 4 byte pieces of data rather than colors/fragments/pixels.

The solution to basic 2D ray-tracing is to define the minimum number of rays to be traced, map each ray to a 1D index, then map that two a 2D array and as such the texture being rendered. In the case of lighting, we can define the number of rays as equal to the circumference of a circle of the light at a given radius, map each ray to a polar coordinate (in 1D space) then map that to a 2D array. Save the output texture and pass it to a secondary shader that builds the light by having each pixel look up the closest ray at an angle from the pixel to the light center, then do distance checks to see if the pixel is within bounds (lit up) or out of bounds (dark) of the ray. This solution allows the minimal number of rays to be cast and minimal number of pixel texture samples to be made for any given light source at a given radius which reduces the search time for the ray-tracing effectively providing exponentially better performance and flexibility than most other standard methods.

How It Works

Why the hassle?
If you want shadow casting in a 2D scene you have several methods: sprite based shadow casting, signed distance fields (SDF), polar transforms, polygonal shadow casting, etc. etc. All of these methods can be fast, but can be very complicated to set up depending on the level of versatility you want in your scene. None of these methods allow for pixel-level lighting detail without a bunch of backend math, even then you're restricted to predefined shapes.

By utilizing Polar-Ray Tracing we can define a set number of rays to cast around the center of a point--in this case the center of a light point--and trace outwards away from the light until each ray hits a collision. This means we'll only trace the minimal necessary number of rays needed to trace out the final shape of the light. By doing this in a shader we can take advantage of the massive power of GPU parallel processing by rendering all of the rays in parallel for each light. The number of rays cast is actually similar to real life. In real life a light's radius is proportional to its energy output (number of photons). In 2D space the photons are our rays where each photon steps in a line from the light center to the light edge (at a given light radius). So then in 2D space the number of rays(photons) is the reverse of real-life, directly proportional to the light's radius: rays = 2piR. This way the number of rays scales with the circumference of the light to fill in gaps as the light grows in size.

Ray-tracing is slow, very slow as you need to iterate over every pixel between point A to another point B on a texture and perform texture sampling. Well GPUs excel at texture sampling so the solution of the problem is to minimize the amount of rays needed per light to achieve the desired result. That is exactly what this shader does and turns out… It's lightning fast, fast enough for real-time lighting. So let's look at how Polar-Ray Tracing works.

Coordinate Planes (Background Info)
The key to this method is the polar coordinate plane and transforming polar coordinates (1D angles), into cartesian coordinates (2D XY positions). Every 2D array or texture is a 1D array of data with a mathematical transform to split the 1D array into several segments that represent rows. To take this further polar coordinates are simply angles pivoted around the center of a pole and if you unwrap that you get a 1D array of angles.

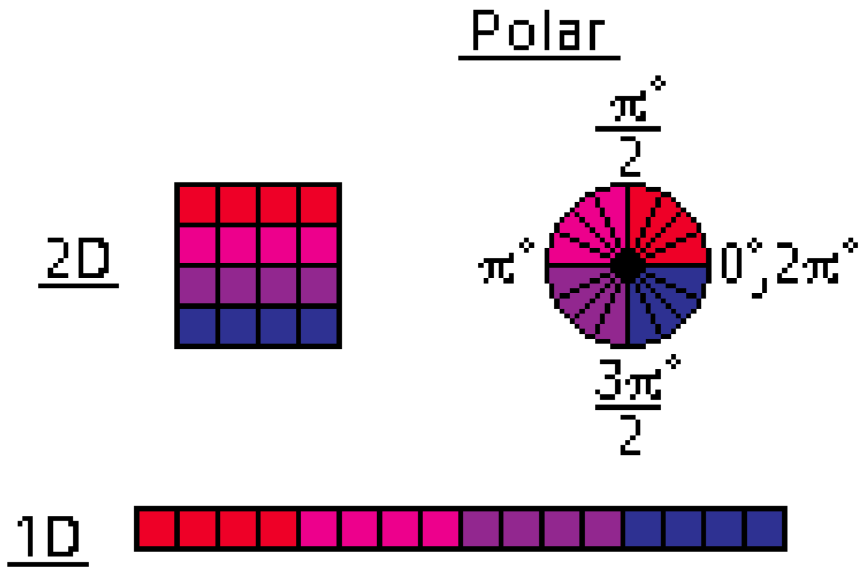

Figure 1.

As such we have a few formulas for converting back and forth between these coordinate spaces--I am only listing the ones relevant to this method:

2D to 1D:
Index = (X * W) + Y
1D to Polar:
Angle = 2pi * (Index/Count)
2D to Polar 2D:
Angle = 2pi * (Index/Count)
vec2(x,y) = cos(Angle), -sin(Angle)
2D to 1D given delta from XY to XY of circle center:
vec2(x,y) = XY - CircleXY
Index = round((Count * fract(atan2(-vec2.y, vec2.x)/2pi)));
1D to 2D:
x,y = mod(Index/Width), Index/Height

2pi can also be represented as Tau, which is what I pre-compute and use in the shaders.

Dual-Pass Shaders

The shader code is provided with this documentation along with a working example using GameMaker:Studio 2. See download link at the beginning of this document or attached ZIP file.

This methodology uses a dual-pass shader where the first shader outputs a texture containing pre-traced rays and the second shader uses that texture to lookup the pre-traced rays pointing in the direction of each pixel in the scene to do distance comparisons for the final light render. Let's take a look at this in steps from pass A to B.

Pass A: RayTracing Shader

Keep in mind the information provided on coordinate planes in the previous section. The first shader pass takes an input texture specifically sized to render the length of each ray to. This input texture is based sized using the following formula:

W,H = pow(2.,ceil(log2(sqrt(2.*PI*RADIUS_SIZE))));

For any square texture, the size of that texture is always the sqrt() of the number of components you want to fit in a texture (because inversely the number of components you can fit in a square is x^2. However GPUs prefer texture sizes in the size of powers of 2. So we get the sqrt() of the number of rays and find the log(2)--or power of 2 exponent of the number of rays--then square it. If the light radius is 128px then we have the following:

2pi*128px = sqrt(804 rays) = log2(28.3) = pow2(ceil(4.82)) = 32 x 32 texture size.

This is done to efficiently use only the minimally necessary texture size for any given set of rays to keep from processing too many extra blank pixels in the texture.

Given that texture size of 32 x 32 for 804 rays we can split the texture into two sections: 804 ray-pixels, 220 blank-pixels (from left to right, top to bottom). See figure 2.

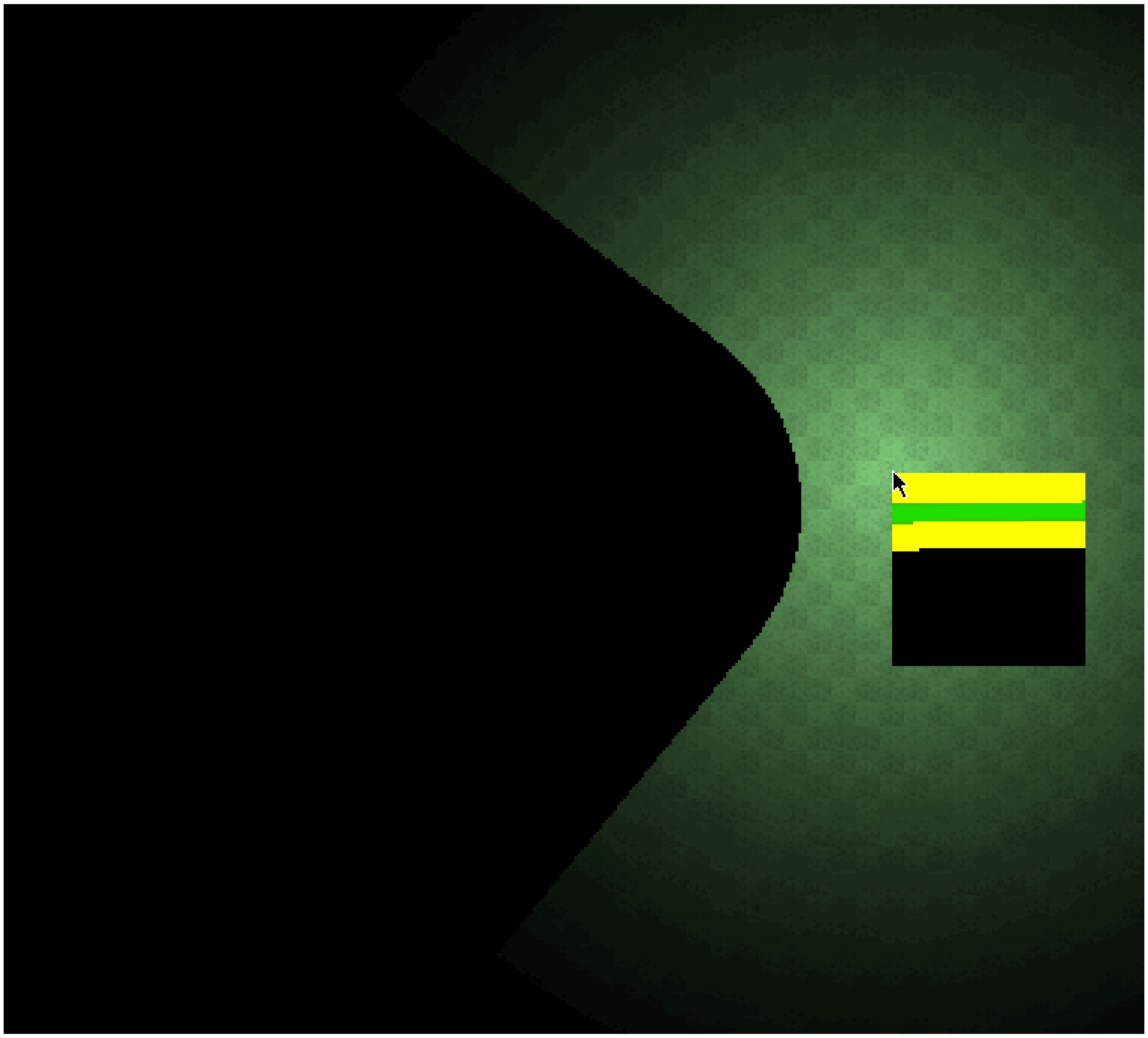

Figure 2.

As the light moves around the collision in the center of the image you can see how it's rays get shorter as they meet the collision area. You can then see how each pixel in the square by the mouse maps from top-left (0 degrees), top-middle (pi/2 degrees), bottom-middle (3pi/2 degrees) and finally bottom (2pi degrees).

Green shows shortened rays colliding.
Yellow shows maximum ray length (no collision).

This is because the shader only uses the R(red) and G(green) components of the texture for ray lengths of 2^16-1 length (65,535px max).

Each pixel on the texture in *Figure 2* is a ray with a given length stored as the gl_FragColor R,G components. Each pixel has a 1D index from 0 to texture width squared. For each pixel we compute it's index in the texture then convert that index to an index to the angle/ray that the pixel will represent:

```
RayCount = 2pi*Radius
Index = (x * w) + y
Angle = 2pi * Index/RayCount
```

Once we know the angle that is associated with the ray for this pixel we can iterate from the center of the light outwards to the edge of the light's perimeter until we hit a collision on the collision map. This iteration is started by finding the delta XY position at the associated angle, this is equivalent to GMS2's lengthdir_xy(dir,len) functions:

```
vec2(x,y) = cos(Angle), -sing(Angle)
```

The iteration is then performed as a for-loop from 0 to max radius.. You cannot have variable length for-loops in shaders, so we set a predefined max radius and cut short using the GLSL-ES step(val,val) function. We then multiply the iterator by the delta position to step across the ray. Then you can do a texture lookup at that position: (here we're only looking at the A (alpha) component because we're only using a black alpha texture where every pixel is black and collisions have an alpha of 1 and no collision alpha of 0:

```
Validate = step(index,count)
for(i=0;i<MaxRadius * Validate;i++) {
    Ray = vec2(x,y) * i
    Collision = texture2D(CollisionMap, light_xy + Ray).a
    If (Collision || Ray > Radius) break;
}
```

Once a collision is hit or we've reached the lights' perimeter the length of the ray from the center of the light to the ray end point is then stored in the gl_FragColor's R,G components by breaking down the length as two bytes. For any pixel with an index greater than the ray count we ignore and do not ray trace.

That's it, pretty simple execution. This method will scale with any texture size and ray count just make sure that you've checked your math on your texture sizes.

Pass B: Light Sampling Shader
The second pass is a bit simpler: we pass in a texture that will fit the size of the light that we want to render, then look up the associated ray from the ray-texture in Pass A and check the distance from the pixel to the light center vs the length of the ray. If the distance is greater than the ray length the pixel is dark, if shorter the pixel is lit up.

This is done by finding the delta XY from the current pixel to the light center. Then calculating the ray count for the light:

> Delta = vec2(x,y) = XY - LightXY
> RayCount = 2piR

Once we have both the Delta XY and RayCount we can use atan2(y,x) to find the angle using atan2()... however it's not so easy considering the angle layout of atan2() being from 0 to pi (0-90 degrees) then -pi to 0 (180 to 356 degrees)....

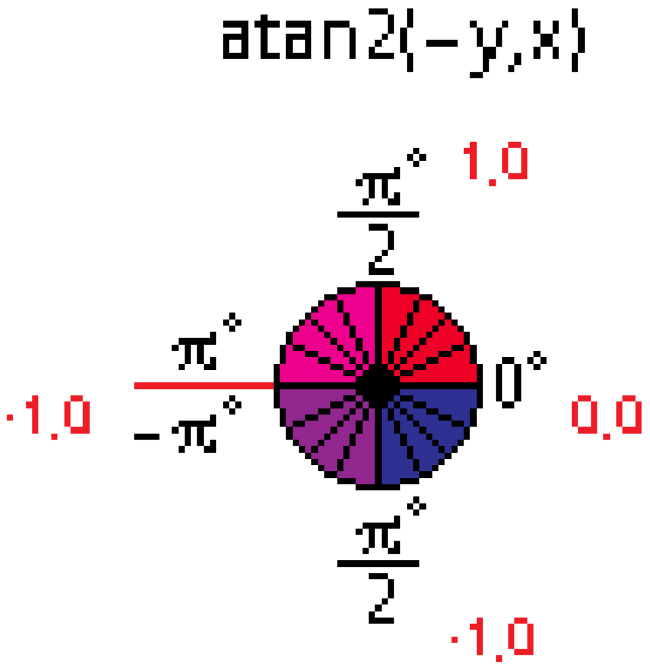

Figure 3.

So if you've ever been confused using atan2() then this is why. We could use atan(y/x) which uses a range from 0 to 2pi, however atan(y/x) doesn't account for when x = 0 and you end up with divide by zero or undefined behavior.

So again once we have the delta position and ray count we can find the angle of the pixel to the light using atan2(y,x):

> *Note: if you're using GameMaker then your y must be -y when using any sort of function dealing with angles because GameMaker uses an inverted y-axis.*

```
Angle = round(RayCount * fract(atan2(y,x)/2pi))
```

How does this work? Well we take our angle atan2(y,x) and we get a value between -pi and pi when we divide by 2pi we get the normalized value of the angle as angle/max_angle in the range of 0.0 to 1.0. When we multiply by the RayCount we get the current pixel's ray index from 0 to RayCount. Once you know the ray index you can convert the ray index to the 2D texture coordinate in the ray texture from Pass A:

```
X,Y = mod(Index/RayTexWidth), Index / RayTextureHeight
```

From there we can read out the ray from the texture:

```
texture2D(RayTexture, XY).rg
```

Then convert the R,G components back to the original ray length. Finally we compare the length of the ray to the distance of the current pixel from the light center and only light up the pixel if the distance is shorter than the ray. From there you apply your tonemap and color and voila, ray-traced lighting. The color blending can be handled in another shader, but since I am using GameMaker I am using GameMaker's built in blend modes for that. Final result: