jason.today

Building Real-Time Global Illumination: Part 1

Jason McGhee

20–26 minutes

Naive GI: A Foundation

Part 2: Radiance Cascades is now available. But read this post first!

This is what we will build in **this post**.

Drag around inside. Want a bigger canvas?

Colors on the right - try toggling the sun!

*I'll be using three.js for this post and **everything** we discuss is written from scratch and entirely contained in the post / html. I wrote this with mdxish, so you can read this post in markdown as well, where all executed code is broken into grokable javascript markdown codeblocks. This post supports desktop and mobile.*

Global illumination is about how light interacts with surfaces. When you turn a light on in your room at night, it's not a small glowing orb. It spreads to the walls, the floor, and the ceiling, breathing life into the room. Desks, doors, and objects cast shadows, softening as they stretch away.

Simulating these interactions can make for some beautiful scenes - as we're more closely mimicking real life. Think Pixar, beautiful blender renders, and hyper-realistic voxel scenes. But historically, it required powerful hardware, time, and/or aesthetic compromises - like noise. There are some fantastic tutorials on various techniques, such as ray tracing (video), which can be used to simulate realistic lighting- but, not in real-time.

Over the last six months or so (at time of writing), a group of folks have been working hard on a new technique that enables real-time global illumination on consumer hardware, without the standard compromises. It's called Radiance Cascades. A fast, noiseless approach to global illumination.

And that's what we *will* (next post) be building, but doing so requires a foundation of code and conceptual understanding. So we're going to build up our repertoire and knowledge and first build "naive" global illumination. Next time, we'll effectively get the same end result, but much higher resolution and much more efficient.

Let's get started!

A drawable surface

If we're going to be messing around with lights and shadows, it's incredibly useful to be able to quickly and easily draw on the screen. We'll use it to test things out, understand limitations and issues, and build intuition around the algorithms and techniques we're going to use. If you are anxious to get to the global illumination part of this post, feel free to jump straight to raymarching.

In general, to get this kind of interface, we just need to place some pixels on the screen in a group and interpolate from the

previously drawn point, in case the brush moved within the last frame and is still drawing.

On the GPU side, we can do what we want using an SDF line segment.

As this is our first shader in the post - this is a fragment shader. The code it executes individually on every pixel. To draw a line (or any shape) you describe it in terms of distance away from the pixel currently being processed by the GPU. `p` here is the position of the pixel in coordinate space, meaning, if it's `vec2(20.0, 20.0)` it's intuitively 20 pixels towards height, and 20 pixels towards width. This is unlike `uv` - more on that in a sec.

What we need to do to represent a line with width (in this case, radius) is describe the distance from our pixel position, to the nearest point on the line, and if that's less than half the width (or `radius` here) set the pixel to be the chosen color.

```
// Draw a line shape!
float sdfLineSquared(vec2 p, vec2 from, vec2 to) {
  vec2 toStart = p - from;
  vec2 line = to - from;
  float lineLengthSquared = dot(line, line);
  float t = clamp(dot(toStart, line) /
lineLengthSquared, 0.0, 1.0);
  vec2 closestVector = toStart - line * t;
  return dot(closestVector, closestVector);
}
```

If you've never heard about SDFs (signed distance functions) you can [read about them on Inigo Quilez's blog] and / or read the section on . [drawing shapes on the GPU] in the Book of Shaders.

And I know, you're probably saying - but there are no calls to `distance` or `length` - what gives? This doesn't look like the examples. But a neat trick is that we can avoid a `sqrt`, if we use `dot` and keep everything we're comparing squared. To clarify, `distance(a, b) == length(a - b) == sqrt(dot(a - b, a - b))`. So if we're comparing to `radius` - we can instead just pass in `radius * radius` and not need `sqrt`. (`sqrt` is expensive).

At this point, we can draw our line.

```
void main() {
  vec4 current = texture(inputTexture, vUv);
  // If we aren't actively drawing (or on mobile) no-op!
  if (drawing) {
    vec2 coord = vUv * resolution;
    if (sdfLineSquared(coord, from, to) <= radiusSquared) {
      current = vec4(color, 1.0);
    }
  }
  gl_FragColor = current;
}
```

Any time you see `uv` (or `vUv`), it is the position of the current pixel, but on a scale of 0-1 in both dimensions. So `vec2(0.5, 0.5)` is always the center of the texture. You can pass in the size of the texture as a `uniform` if you want to be able to convert to pixel-space, which we do here.

ThreeJS uses `vUv`. I believe that first `v` is for [`varying` keyword](#).

If you want to add some extra polish you can make it feel a bit smoother / paintier by adding easing. Check the source to see

how I did it! Spoiler: I cheated and did it on the CPU.

Whichever engine you're using, be it a game engine, webgl, p5.js, three.js, webgpu, (and most others), there will be a way to say, "draw this rgb value at this location", and then it's just expanding that to a radius, from one point to another - or using SDFs as shown above.

And after skimming over some details that aren't the focus of this post, we've got our drawable surface!

If you're interested in the plumbing, inspect this node in the browser, or go to this section in the markdown! Right above this dom is where the code that runs, lives.

Raymarching

At its core, global illumination is going through a scene, looking around at nearby lights, and adjusting how bright that area is based on what we see. But *how* you do that makes all the difference.

So we're going to start with a naive approach, but the core logic ends up being nearly identical to dramatically more efficient approaches, so we're not wasting time building it. It's called raymarching. *Link contains spoilers!*

Remember when ten seconds ago I said global illumination is going through a scene and looking around at nearby lights to determine your "radiance"? Let's try it out by telling the GPU (for every pixel) to walk in a handful of directions for a larger handful of steps and to average the colors of anything it hits - and to stop walking if it hits something.

Before we proceed, let's define a quick bounds check function

we'll use.

```
bool outOfBounds(vec2 uv) {
  return uv.x < 0.0 || uv.x > 1.0 || uv.y < 0.0 ||
uv.y > 1.0;
}
```

Ok, let's walk through what we just said above, but with glsl.

```
vec4 raymarch() {
```

So first we query our drawn texture. If it's something we drew, there's no need to process this pixel, as it can't receive light. If we were doing subsurface scattering, or reflective or transparent materials, we *would* need to process them, as they would interact with light.

```
vec4 light = texture(sceneTexture, vUv);

if (light.a > 0.1) {
  return light;
}
```

Now we know we're only dealing with pixels that can receive light. We'll declare two variables. `oneOverRayCount` which we'll use to take the average of all light we see. And `tauOverRayCount` which tells us how much of an angle there is between our rays - which are equally spaced in a circle around the pixel.

We'll also get a random number between 0-1 based on our pixel location. We'll use it to slightly offset the angle of each ray (the random offset is seeded based on our position), so that rays don't all line up across pixels. That means no repeating patterns!

If Radiance Cascades didn't exist, I'd probably bring up blue noise, but the goal is not using noise at all - so any old noise is

fine for our purposes.

Finally, we have some `radiance` that we'll add to, if we hit something.

```
float oneOverRayCount = 1.0 / float(rayCount);
float tauOverRayCount = TAU * oneOverRayCount;

// Distinct random value for every pixel
float noise = rand(vUv);

vec4 radiance = vec4(0.0);
```

Now we get to start firing rays!

```
for(int i = 0; i < rayCount; i++) {
```

For every ray, calculate the direction to head, and start walking.

Since we know our current position is empty, we can take our first step.

```
float angle = tauOverRayCount * (float(i) + noise);
vec2 rayDirectionUv = vec2(cos(angle), -sin(angle))
/ size;

// Our current position, plus one step.
vec2 sampleUv = vUv + rayDirectionUv;

for (int step = 0; step < maxSteps; step++) {
```

Now, for every step we take, if we walk out of bounds (or hit `maxSteps`), we're done.

```
if (outOfBounds(sampleUv)) break;
```

If we hit something, collect the radiance, and stop walking. Otherwise, take another step.

```
vec4 sampleLight = texture(sceneTexture, sampleUv);
if (sampleLight.a > 0.1) {
  radiance += sampleLight;
  break;
}

sampleUv += rayDirectionUv;
```

If we hit an end condition, return our collected radiance, averaged over all directions we walked / rays we fired!

```
      }
  }
  return radiance * oneOverRayCount;
}
```

And that's it!

Now we can tweak the code a bit (see source to see how) in order to provide controls over it. This way we can build some intuition around why we did what we did.

Enable Noise

Checkout what happens when we turn off the noise!

Accumulate Radiance

Checkout what happens when we don't accumulate radiance to see the rays traveling.

Max Raymarch Steps

Don't forget you can draw.

We've done it! Looks pretty similar to the final result at the top.

And we could increase the rays, replicate those features (grain and sun) and call it a day, but it's actually wildly inefficient. Maybe it doesn't even run well on whatever device you're using,

maybe it does. But we're only casting 8 rays per pixel and light can only spread 256 pixels away - also that canvas is absolutely tiny and can't be made much bigger and run smoothly, even on more powerful machines.

If we take a moment to think about the core logic of our raymarching, we're doing a lot of repeated work. Think about all the overlapping steps we're taking - hint they are mostly overlapping. So caching right? Well, most of what we're doing is in parallel per-pixel, so the values of other pixels aren't available yet. So we're going to have to come up with another approach.

Remember those distance fields from earlier? Where we were representing shapes in terms of their distance away from any given pixel? Well, if we had a way to tell how far away the nearest filled pixel was, from any given pixel, we'd know the maxiumum safe distance we could jump - in any direction. After we jumped, we could reassess, and again know the maximum safe distance we could jump. This would save a ton of computation during our raymarching process as we're getting to completely ignore large swathes of our surface.

Now, you might be thinking, alright - so we need a map that shows the nearest filled pixel for every pixel - what are we going to do, shoot rays in all directions and take steps in every direction find the first one? That can't be much more efficient than our current raymarching process! And you'd be right.

But maybe we can get more creative.

Jump Flood Algorithm

The loose idea here is, if we get the pixel coordinates of every filled pixel in our input / surface texture, and spread them around in the right way, we'll end up with a texture where every pixel

from our original image is still just its `uv`, but all the other pixels are whichever `uv` from the original image is the least distance away. If we manage to do that - all we need to do is calculate the distance between the original and newly smeared pixel locations, at each pixel, and we've got a distance field.

For the "smearing" bit - we can just hierarchically repeat our `vUv` transformed `seed` texture, on top of itself a handful of times.

So first, let's make that `seed` texture, which you can see by going to "1 pass" in the slider. We just multiply the alpha of our drawing texture by the `uv` map. You can see the `uv` map by moving the slider to "No passes". You can also see this as the result of a fragment shader by setting the pixel color to be `vec4(vUv.x, vUv.y, 0.0, 1.0)` - which is frequently used for debugging shaders.

```
float alpha = texture(surfaceTexture, vUv).a;
gl_FragColor = vec4(vUv * alpha, 0.0, 1.0);
```

The Jump Flood Algorithm, JFA, denotes we should use `log(N)` of the largest dimension of our texture. This makes sense given the branching nature of what we're doing.

```
const passes =
Math.ceil(Math.log2(Math.max(this.width,
this.height)));
```

And then we can execute our JFA. We're using the "ping-pong" technique of having two render targets and swapping between them in order to accumulate the final JFA texture we want. So render, swap, render, swap, etc. `passes` times. We can't just use the same texture / render target as this is all happening in parallel, so you'd be modifying pixels that hadn't been handled yet elsewhere, causing inconsistencies.

```
let currentInput = inputTexture;
let [renderA, renderB] = this.jfaRenderTargets;
let currentOutput = renderA;

for (let i = 0; i < passes; i++) {
  plane.material.uniforms.inputTexture.value =
currentInput;
  plane.material.uniforms.uOffset.value =
Math.pow(2, passes - i - 1);

  renderer.setRenderTarget(currentOutput);
  render();

  currentInput = currentOutput.texture;
  currentOutput = (currentOutput === renderA) ?
renderB : renderA;
}
```

So what about the actual shader? Well, we need to keep track of the nearest filled pixel to us and how far away it is. (I used -2.0 to make sure it was completely out of frame) and a big number for distance.

```
vec4 nearestSeed = vec4(-2.0);
float nearestDist = 999999.9;
```

Then, we just look all around us, offset by `uOffset` over `resolution` (we could totally precalculate that and pass it in) and if it's a filled pixel that's closer to us than what we've found so far, great! And note that we can't break early, because a pixel is "closer" based on the color it represents, not based on how far we've traveled, or something.

```
for (float y = -1.0; y <= 1.0; y += 1.0) {
```

```
  for (float x = -1.0; x <= 1.0; x += 1.0) {
    vec2 sampleUV = vUv + vec2(x, y) * uOffset /
resolution;
```

Quick bounds check, then we can get the distance based on the red and green pixel values currently stored there, compared with our own. I could have used `distance` but chose to use `dot` because why use the extra `sqrt`? We're only comparing relative distances.

```
if (sampleUV.x < 0.0 || sampleUV.x > 1.0 ||
sampleUV.y < 0.0 || sampleUV.y > 1.0) { continue; }

vec4 sampleValue = texture(inputTexture, sampleUV);
vec2 sampleSeed = sampleValue.xy;

if (sampleSeed.x != 0.0 || sampleSeed.y != 0.0) {
  vec2 diff = sampleSeed - vUv;
  float dist = dot(diff, diff);
  if (dist < nearestDist) {
    nearestDist = dist;
    nearestSeed = sampleValue;
  }
}
```

And we're done - set ourselves as the `uv` of the nearest filled pixel, ready to be compared to in the next pass (if applicable).

```
  }
}

gl_FragColor = nearestSeed;
```

Let's check out what this looks like in practice!

No Passes 9 Passes

I'm also showing "No Passes" here, which is the UV map of all pixels, without the drawing.

If you haven't been experimenting... you can draw regardless of the settings, and it will appropriately render what *would* have been rendered if you had drawn on a canvas and that was passed to the GPU. This is how all these canvases are instrumented.

So... we're pretty much there on our precomputed nearest-filled-pixel distance lookup. We just need to do a quick pass with a shader to convert the texture.

Distance Field

We need to take our JFA texture - which is the `uv` of our filled pixels smeared about hierarchically, and just measure the distance between the stored `rg` (red, green) value and the `uv` itself!

So let's go ahead and do it.

```
vec2 nearestSeed = texture(jfaTexture, vUv).xy;
// Clamp by the size of our texture (1.0 in uv
space).
float distance = clamp(distance(vUv, nearestSeed),
0.0, 1.0);

// Normalize and visualize the distance
gl_FragColor = vec4(vec3(distance), 1.0);
```

We also clamped it to make sure things that are further than 1 `uv` away are still just treated as 1.

But that's really it- and now we can dramatically improve the

runtime of our raymarching with our fancy jump distance lookup.

Erase / Drag around

Let's update our raymarching logic to use our new distance field!

Naive Global Illumination

Yes, this is still naive global illumination, but much better than what we started with!

I'm going to jump right in. Remember what our original raymarch approach was doing per-ray?

We're doing *almost* exactly the same thing, but `maxSteps` can be way smaller - like 32 now, and definitely goes to the edge of the screen. And this new `maxSteps` effectively controls quality and scales with the size of the canvas. Our previous method is a set distance (like 256 steps) so needs to be increased with larger canvases, and is clearly more expensive.

```
// We tested uv already (we know we aren't an
object), so skip step 0.
for (int step = 1; step < maxSteps; step++) {
    // How far away is the nearest object?
    float dist = texture(distanceTexture,
sampleUv).r;

    // Go the direction we're traveling (with
noise)
    sampleUv += rayDirection * dist;

    if (outOfBounds(sampleUv)) break;

    // We hit something! (EPS = small number, like
0.001)
```

```
    if (dist < EPS) {
      // Collect the radiance
      radDelta += texture(sceneTexture, sampleUv);
      break;
    }
}
```

And that's all we need to change to get our naive global illumination implementation!

Playground

Here are a bunch of controls to play with what we've built, and some extra features.

Enable Noise

Enable Sun - How ?

Enable Angular Noise (Sand Grain) - How ?

Enable Temporal Accumulation - How ?

Max Raymarch Steps

Sun Angle

Bonus Material

Oh, you're still here? Cool. Let's peel back the demo above then!

Each control has a "How?" link which jumps to the explanation, and there's a "Jump back to playground" at the end of each section. No need to read in order!

Make it look like we're outside

So if we want to make it look like we're outside, we need to record when we hit a surface, because if we didn't, we want to add extra radiance from the sun / sky.

```
if (dist < EPS) {
  radDelta += texture(sceneTexture, sampleUv);
  // Record that we hit something!
  hitSurface = true;
  break;
}
```

It's important we record that we hit something because if we *didn't*, then we want to add some sun/sky look and feel (radiance). And then we can just take a literal slice of the sky, and cast rays from that direction

```
if (!hitSurface) {
  radDelta += vec4(sunAndSky(angle), 1.0);
}
```

So, let's define the key radiance calculation.

First, choose some colors which represent what color and how much of an impact the "sky color" and "sun color" will have on the final render. The "sky color" will always be the same, while the "sun color" will be applied based on its angle. I chose to make the sun matter a lot more, be near white, and for the sky to be much dimmer with a blue hue. But these can be whatever you want!

```
const vec3 skyColor = vec3(0.02, 0.08, 0.2);
const vec3 sunColor = vec3(0.95, 0.95, 0.9);

vec3 sunAndSky(vec2 angle) {
```

First, we get the sun's relative angle to our ray, and `mod` with `TAU` to keep it in our unit circle.

```
float angleToSun = mod(rayAngle - sunAngle, TAU);
```

Then we calculate what the sun's intensity is, based on that angle, making use of `smoothstep` to add some smoothing.

```
float sunIntensity = smoothstep(1.0, 0.0,
angleToSun);
```

And finally, we apply the intensity to the sun color and add the sky color.

```
return sunColor * sunIntensity + skyColor;
```

That's it! Every time I've come across "sky radiance" in the wild, it's always some crazy set of equations I barely understand, and/or "copied from <so and so's> shadertoy". Maybe I'm way over simplifying things, but, this approach is what I landed on when thinking it through myself.

Jump back to playground.

Sand grain

So I was playing with noise, and happened on pixels "catching the light" by adding our same noise to how much we **increase** the angle by each iteration, adding some noisy overlap. So certain pixels get double the radiance.

```
float rayAngleStepSize = angleStepSize + offset *
TAU;
```

And in our core ray loop, where `i` is our ray index, I added the `sunAngle` to it, so the light would catch (overlap would happen) in the same direction as the sun.

```
float angle = rayAngle * (float(i) + offset) +
```

```
sunAngle;
```

Jump back to playground.

Temporal accumulation

This is pretty straightforward, but there's a lot you can do with it to customize it. I didn't mention it earlier because Radiance Cascades doesn't require temporal accumulation, which is basically a hack to deal with noise.

All we need to do to get temporal accumulation is, add a time component to our noise seed, so it varies over time, and mix with the last frame as an additional texture (ping-pong buffer).

Anywhere you use `rand` and what it to get smoothed, just add time.

```
rand(uv + time)
```

And when you return the final radiance, just mix it with the previous frame.

```
vec4 prevRadiance = texture(lastFrameTexture, vUv);
gl_FragColor = mix(finalRadiance, prevRadiance,
0.9);
```

On the CPU side, just update the loop to include / store the previous texture, and add a stopping condition. If the time reaches, say, 10 frames, stop mixing. `prev` is just `0` or `1 - ` initialized to `0` somewhere.

```
plane.material.uniforms.lastFrameTexture.value =
lastFrame ?? surface.texture;
renderer.setRenderTarget(giRenderTargets[prev]);
render();
prev = 1 - prev;
lastFrame = target.texture;
```

```
plane.material.uniforms.time.value += 1.0;
```

I also opted to pause temporal accumulation while drawing. It impacts performance as it needs to set a new render target each frame and adds a texture lookup.

Jump back to playground.

GI Falling Sand

Curious how to make this? Checkout Making a falling sand simulator.

Ready for part 2, Radiance Cascades? Check it out!