jason.today

Radiance Cascades: Building Real-Time Global Illumination

Jason McGhee

25–31 minutes

---

jason.today

Building Real-Time Global Illumination

An Interactive Walkthrough

This is the second post in a series. Checkout the first post, which walks through raymarching, the jump flood algorithm, distance fields, and a noise-based global illumination method. We use them all again here!

This is what we will build in this post. It's noiseless, real-time global illumination. The real deal.

Drag around inside!

Colors on the right - try toggling the sun!

▸ Additional Controls

▸ *Want a bigger canvas?*

▸ *Performance Issues?*

Why is the previous post's approach naive?

So last time we left off with a method that sent N rays in equidistant directions about a unit circle, with a bit of noise added whenever we pick a direction to cast a ray. Each ray marches until it hits something, leveraging sphere marching via a distance field to jump as far as possible with high certainty that nothing would be missed or jumped over. The stopping condition is hand-wavy, just some epsilon value - which we'll improve on.

In that post we used 32 rays - that's pretty low accuracy, which shows. We try to cover up with style and temporal accumulation. It should be noted that it could look much smoother using the same method. If we were to cast many more rays per pixel, say, 512 or 1024, use a more appropriate type of noise, and some additional smoothing... but it would no longer run in real-time in common cases - especially at large resolutions.

Take a look at the impact and look that noise has, and how smooth our GI can get at high ray counts.

Naive GI Noise

Ray Count32

An important question to ask is certainly, "do we care about noise?" If you like the aesthetic of noise or otherwise don't mind it, you can make the decision to use noise based methods. It inherently comes with inaccuracy alongside the visual artifacts. But, put the slider at 16 rays. This is roughly the cost of radiance cascades, yet the end result has no noise. Pretty crazy.

Let's take a moment to talk about how many rays we're casting to get a sense of the computation we're performing every frame. Let's say we have a 1024 x 1024 canvas - that's roughly 1M

pixels. If we're casting 512 (on the low side of nice and smooth) that's ~500M rays cast every frame. Because we added some great performance savings by creating distance fields, we only need to take (on the order of) 10 steps per ray - and really the most computationally expensive operation is a texture lookup which we do once to figure out if we need to perform a raymarch and once if we hit something - so let's call that roughly 2 texture lookups per cast ray totaling around 1B texture lookups. Even if we can perform a texture lookup in a nanosecond (likely slower), we're looking at a full second per frame. To run at just 60fps, we need to get that down to ~16ms. So roughly 100x faster.

How can we simplify the amount of work we need to do without sacrificing quality?

Penumbra hypothesis

It turns out, there's an idea called the "penumbra hypothesis" which provides critical insight that we'll leverage to dramatically reduce the amount of computational effort required.

The idea has two parts.

When a shadow is cast:

1. The necessary linear (spatial in pixel space) resolution to accurately capture some area of it is inversely proportional to that area's distance from the origin.
2. The necessary angular resolution to accurately capture some area of it is proportional to that area's distance from the origin.

Below is a scene that illustrates a penumbra.

Observe the shape and color of the shadow as it extends away

from the left edge of the drawn black opaque line. There's kind of two parts - the dark area on the right, and the area on the left that gets softer the further left you look. That softening center area is shaped like a cone (or triangle) and called the penumbra. (wikipedia has some nice labeled diagrams too).

Like all other canvases, if you want to get the original back, just hit the little refresh button in the bottom right.

So to put the penumbra hypothesis above into more familiar terms, given some pixel, to accurately represent its radiance (its rgba value) we need to collect light from all light sources in our scene. The further away a light source is, the more angular resolution and less linear (spatial pixel) resolution is required. In other words, the more rays we need to cast and the fewer individual pixels we need to look at.

If we observe that shadow in our illustration of the penumbra above, we can kind of see why that's the case in regard to the linear resolution. The shadow's left edge is quite sharp near the left edge of the black line and it's pretty hard to tell where exactly it becomes the same color as the background (or if it does).

As far as angular resolution, we can easily see the sense behind this condition by changing our ray count down to 4 in our previous canvas and placing a black dot in a dark area that *should* be illuminated, then increasing the ray count step by step. We can see how the dot can easily be lost or misrepresented if far enough away and the ray count isn't high enough.

So. More rays at lower resolution the further away a light source is...

Currently, our previous method casts the same number of rays

from every pixel, and operates only according to the scene resolution.

So, how do we dynamically increase the number of rays cast mid raymarch? Sounds like some sort of branching operation which doesn't sound particularly gpu friendly.

Also, reducing linear resolution sounds like taking bigger steps as we get further from our origin during our raymarch, but we swapped to using a distance field, not stepping pixel by pixel like we did in our very first implementation. So how would we reduce our linear resolution as we get further away? Isn't it optimized already?

Codifying the penumbra hypothesis

So instead of dynamically increasing rays mid cast / branching, let's break down our global illumination process into multiple passes. We'll do a high resolution pass with fewer rays, and a lower resolution pass with more rays. According to the penumbra hypothesis, that should more accurately model how lights / shadows behave.

So to start - we need our distance field texture and our scene texture (same as before). But this time we'll also need to keep around a `lastTexture` that we'll use to keep around the previous pass.

```
rcPass(distanceFieldTexture, drawPassTexture) {
  uniforms.distanceTexture.value =
distanceFieldTexture;
  uniforms.sceneTexture.value = drawPassTexture;
  uniforms.lastTexture.value = null;
```

```
  // ping-pong rendering
}
```

We need to be able to pass the previous raymarch pass into the next, so we'll setup a `for` loop using a ping-pong strategy and two render targets just like when we made our multi-pass JFA.

We'll start with the highest ray count, lowest resolution layer, (256 rays and 1/16th resolution). Then we'll do our low ray count, high resolution render.

```
// ping-pong rendering
for (let i = 2; i >= 1; i--) {
  uniforms.rayCount.value =
Math.pow(uniforms.baseRayCount.value, i);

  if (i > 1) {

renderer.setRenderTarget(rcRenderTargets[prev]);
    rcRender();
    uniforms.lastTexture.value =
rcRenderTargets[prev].texture;
    prev = 1 - prev;
  } else {
    uniforms.rayCount.value =
uniforms.baseRayCount.value;
    renderer.setRenderTarget(null);
    rcRender();
  }
}
```

In the shader, we're going to make 3 modifications to our original naive gi shader.

1. Add the ability to render at a lower resolution

2. Add the ability to merge with a previous pass
3. Add the ability to start and stop at a specified distance from the origin of the raymarch.

   That last modifications models the behavior in the penumbra hypothesis that describes how the required resolution drops and required ray count increases as you move further away. To model that behavior, we need to be able to tell our shader to march between a specified interval.

   So starting with rendering at a lower resolution - let's just try something out.

   Let's cut our resolution in half (floor each pixel to half).

```
  vec2 coord = uv * resolution;

  bool isLastLayer = rayCount == baseRayCount;
  vec2 effectiveUv = isLastLayer ? uv : floor(coord
/ 2.0) * 2.0 / resolution;
```

   Let's decide where to cast our ray from and how far it should travel. We'll arbitrarily decide on 1/8th of the UV space (screen) and the end will be the longest possible `sqrt(2.0)`. If it's our low-res pass, start at `partial` otherwise only go *until* `partial`.

```
float partial = 0.125;
float intervalStart = rayCount == baseRayCount ?
0.0 : partial;
float intervalEnd = rayCount == baseRayCount ?
partial : sqrt(2.0);
```

   And now our core raymarch loop (mostly reiterating) but no noise this time!

```
// Shoot rays in "rayCount" directions, equally
spaced, NO ADDED RANDOMNESS.
```

```
for (int i = 0; i < rayCount; i++) {
    float index = float(i);
    // Add 0.5 radians to avoid vertical angles
    float angleStep = (index + 0.5);
    float angle = angleStepSize * angleStep;
    vec2 rayDirection = vec2(cos(angle), -
sin(angle));

    // Start in our decided starting location
    vec2 sampleUv = effectiveUv + rayDirection *
intervalStart * scale;
    // Keep track of how far we've gone
    float traveled = intervalStart;
    vec4 radDelta = vec4(0.0);
```

And when we actually take our steps along the ray...

```
    // (Existing loop, but to reiterate, we're
raymarching)
    for (int step = 1; step < maxSteps; step++) {
      // How far away is the nearest object?
      float dist = texture(distanceTexture,
effectiveUv).r;

      // Go the direction we're traveling
      sampleUv += rayDirection * dist * scale;

      if (outOfBounds(sampleUv)) break;

      // Read if our distance field tells us to!
      if (dist < minStepSize) {
          // Accumulate radiance or shadow!
          vec4 colorSample = texture(sceneTexture,
```

```
sampleUv);
          radDelta += vec4(pow(colorSample.rgb,
vec3(srgb)), 1.0);
          break;
      }

      // Stop if we've gone our interval length!
      traveled += dist;
      if (traveled >= intervalEnd) break;
    }

    // Accumulate total radiance
    radiance += radDelta;
}
```

And then same as before, we set the pixel to the final radiance... We also corrected sRGB here as well to make it easier to see all the rays (more on that in a bit)

```
vec3 final = radiance.rgb * oneOverRayCount;
vec3 correctSRGB = pow(final, vec3(1.0 / 2.2));

FragColor = vec4(correctSRGB, 1.0);
```

Here's what that looks like:

Base Ray Count16

Packing Direction in Pixels

We can first notice the two (radiance) cascades (or layers) we now have. One in the background and one in the foreground.

The one in the background has some crazy designs and a hole in the middle. The one in the foreground has clearly visible rays extending from the light source just over the circular hole in background.

We should be careful with language - as we know rays don't extend from the light source, but actually start from what could be perceived as the "end" of the ray "coming out" of the light source and are cast in different directions, one of which hit the light source and thus was illuminated.

Let's swap the base ray count between 4 and 16 and paint around a bit.

Note that our ray counts are 4 rays up close and 16 further away or 16 up close and 256 further away.

When we drag around light, that upper / background layer looks very reasonable- a bit pixelated, but better than the white noise from earlier.

When we draw shadows, they look offset proportional to the offset of the upper interval. But let's not worry about that for a moment...

So that's 256 rays at half resolution. And our core loop through ray angles did 256 rays for every pixel - but because we cut our resolution in half, 3 out of every 4 rays we marched were redundant. Woah!

And that's a key insight in radiance cascades. Specifically for our case - what if we split up the rays we need to cast? Instead of doing half resolution, let's do 1/4th resolution (which is every 16 total pixels) and instead of casting 256 rays per pixel, cast 16 rays for every pixel, offsetting by `TAU / 16` incrementally per pixel.

This group of pixels is called a "probe" in radiance cascades.

Let's make the changes.

Our `baseRayCount` is either `4` or `16` in our running example.

So let's define all our variables.

```
// A handy term we use in other calculations
float sqrtBase = sqrt(float(baseRayCount));
// The width / space between probes
// If our `baseRayCount` is 16, this is 4 on the
upper cascade or 1 on the lower.
float spacing = rayCount == baseRayCount ? 1.0 :
sqrtBase;
// Calculate the number of probes per x/y dimension
vec2 size = floor(resolution / spacing);
// Calculate which probe we're processing this pass
vec2 probeRelativePosition = mod(coord, size);
// Calculate which group of rays we're processing
this pass
vec2 rayPos = floor(coord / size);
// Calculate the index of the set of rays we're
processing
float baseIndex = float(baseRayCount) * (rayPos.x +
(spacing * rayPos.y));
// Calculate the size of our angle step
float angleStepSize = TAU / float(rayCount);
// Find the center of the probe we're processing
vec2 probeCenter = (probeRelativePosition + 0.5) *
spacing;
```

It's a fair amount of new complexity, but we're just encoding our base ray count (4 or 16) rays into each pixel for a downsampled version of our texture.

Then when we actually do our raymarching step, we only need to cast rays `baseRayCount` times per pixel.

```
// Shoot rays in "rayCount" directions, equally
spaced
for (int i = 0; i < baseRayCount; i++) {
  float index = baseIndex + float(i);
  float angleStep = index + 0.5;
  // Same as before from here out
}
```

In our first pass (upper cascade / background layer), we'll end up casting 256 rays total from each group of 16 pixels (probe), all from the same point - specifically the center of the probe.

In our second pass (lower cascade / foreground layer), we'll cast 16 rays total from each pixel (probe again), all from the same point - again, from the center.

That is a total cost of 2 x 16 ray raymarches - the same cost as our 32 ray raymarch, and the penumbra hypothesis says it will be more accurate (and hopefully look better - we're using an angular resolution of 256 rays instead of 32).

But at what point do we swap from the low ray, high spatial resolution to high ray low spatial resolution? We'll make "Interval Split" a slider and play with it.

```
// Calculate our intervals based on an input
`intervalSplit`
float intervalStart = rayCount == baseRayCount ?
0.0 : intervalSplit;
// End at the split or the max possible (in uv)
sqrt(2.0)
float intervalEnd = rayCount == baseRayCount ?
intervalSplit : sqrt(2.0);
```

No changes needed to how we leveraged them. But we do need to "merge" them, as in, when we get to the lower cascade, we need to read from the upper cascade, which is stored differently than a normal texture.

So first things first - we only want to read the upper cascade from the lower layer (there's no layer above the upper cascade) and we only want to do it if we're in an empty area. If we already have light, there's no reason to read from the upper cascade. We already hit something (which we know will be closer).

```
bool nonOpaque = radDelta.a == 0.0;

// Only merge on non-opaque areas
if (firstLevel && nonOpaque) {
```

Once we know we need to merge, we need to decode our encoded texture.

So we store 16 different directions (assuming a base of 16) in 16 different quadrants of our texture. That's a 4 x 4 grid. Once we calculate that we can find our current position by modding and dividing (and flooring) the index with `sqrtBase` (4) to find our x and y terms. Then multiply them by the size of a quadrant that we just calculated.

```
// The spacing between probes
vec2 upperSpacing = sqrtBase;
// Grid of probes
vec2 upperSize = floor(resolution / upperSpacing);
// Position of _this_ probe
vec2 upperPosition = vec2(
  mod(index, sqrtBase), floor(index / upperSpacing)
) * upperSize;
```

Next we offset where we sample from by the center of the

current layers probe relative to the upper probe.

```
vec2 offset = (probeRelativePosition + 0.5) /
sqrtBase;
vec2 upperUv = (upperPosition + offset) /
resolution
```

And finally we accumulate radiance from our previous texture at the calculated `upperUv`.

```
radDelta += texture(lastTexture, upperUv);
```

And here it is.

Use Linear Filter

Base Ray Count16

Cascade Index0

Interval Split0.125

Nailing the Details

Take a moment and swap between "Cascade Index" 0 and 1. When we set it to 1, we render the upper cascade texture directly. That is how it is stored. (starts from bottom left, then goes to the right and then up a row, then to the right etc.) Notice the clockwise rotation.

In general, our new canvas looks relatively similar to the previous canvas, but the upper cascade is clearly lower resolution. If we paint around though, it looks just as good. And if we lower the "Interval Split" a bunch, it looks even better. (shadows look reasonable now too)

There's a pretty clear issue though. Because we're sampling from the upper layer, we're getting a rather large pixelated grid pattern. And this makes sense as we're sampling from a single point for a group of pixels. We can smooth this out without extra work by setting the render target to use a linear filter. This means the gpu will upscale the upper cascade using bilinear interpolation giving us nearly free smoothing. There's a checkbox to compare with and without using a linear filter on the upper cascade. Once we enable it, things are really starting to look good.

Playing a bit with the Interval Split, the ideal split seems to be pretty close to zero, but clearly not 0 (after drawing some light and shadow lines). ~0.02 (or like 8 pixels) looks a bit too long, but reasonable. Which is about half (or the radius) of 16 pixels. And the probe width is 16 pixels - and we're casting it from the center. So we'll need to keep experimenting but let's say that were our "rule" to determine interval length for a moment... So far we've been talking about this all as just a split.

Let's say we used our rule (which is probably *too long*), 16 x 8 pixels is 128 pixels. Our upper cascade is 256 rays, so it should use an interval that ends at 128 pixels. We'd need another layer - which would have pixel groups of 256 x 256 pixels - so (according to our rule) 256 x 128 pixels max, which is 32K, and beyond any canvas we'd use.

Let's codify that, and we'll also make sure it works for a base ray count of 4 - which will require more layers (we could calculate that too).

Generalizing to Many Cascades

We'll need new variables like `cascadeIndex` and `cascadeCount` which will pass in from the CPU...

But let's first figure out / codify our interval length, as this determines how many cascades we'll need.

Let's try basing everything off of our `baseRayCount` - we'll call it `base`.

Our lowest cascade (index of 0) starts at the center of our probe - that's easy. And for the length - let's have it just go 1 pixel - so 1 divided by shortest side of our resolution - and we'll scale steps by the ratio of the shortest side over resolution.

As we go up in cascade size, we need to scale by some amount. A mathematically simple way to scale is just to start at `pow(base, cascadeIndex - 2.0)` where `cascadeIndex`

```
float shortestSide = min(resolution.x,
resolution.y);
// Multiply steps by this to ensure unit circle
with non-square resolution
vec2 scale = shortestSide / resolution;

float intervalStart = cascadeIndex == 0.0 ? 0.0 : (
  pow(base, cascadeIndex - 1.0)
) / shortestSide;
float intervalLength = pow(base, cascadeIndex) /
shortestSide;
```

We also need to generalize a couple of other variables to be based on `cascadeIndex`.

Ray count is just `base ^ (cascadeIndex + 1)` - as the lowest layer is just base, then the next is `base * base` and so on. Similarly, the spacing for the current layer starts with `1`, then is a

total of `base` pixels which is `sqrtBase` on each dimension. And then `sqrtBase * sqrtBase` and so on.

```
float rayCount = pow(base, cascadeIndex + 1.0);
float spacing = pow(sqrtBase, cascadeIndex);
```

And finally we need to update our merging logic. It's actually really easy due to the effort we put in on the previous canvas - we just need to generalize when we perform it (as long as we aren't processing the upper-most cascade) and then generalize `upperSpacing`.

And, it's just the probe spacing of the upper cascade!

```
if (cascadeIndex < cascadeCount - 1.0 && nonOpaque)
{
  float upperSpacing = pow(sqrtBase, cascadeIndex +
1.0);
```

Alright - back to `cascadeCount` and `cascadeIndex` - we need to calculate and pass those is on the CPU side. Let's modify our logic to figure out how many cascades we need.

Well, say we have 300 x 400 - we know our longest possible ray length is 500 (it's a pythagorean triple after all - so the longest possible line is 500 pixels, which is our longest possible ray). Using a base of 4 - log base 4 of 500 is roughly 4.5, so we'll need a minimum of 5 cascades...

But when we place a light point in the very top-left and limit to 5 cascades, we see that our longest interval doesn't reach the edge. So there's clearly an issue. In the short term, we're going to solve it in the most naive way possible. Add an extra cascade.

```
rcPass(distanceFieldTexture, drawPassTexture) {
  // initialize variables from before
```

```
  const diagonal = Math.sqrt(
    width * width + height * height
  );

  // Our calculation for number of cascades
  cascadeCount = Math.ceil(
    Math.log(diagonal) /
Math.log(uniforms.base.value)
  ) + 1;

  uniforms.cascadeCount.value = cascadeCount;

  for (let i = cascadeCount - 1; i >= 0; i--) {
    uniforms.cascadeIndex.value = i;

    // Same as before
  }
}
```

A fix we'll also discuss here, which we used in the last demo is - whenever we read a texture from the drawn scene, we take it from sRGB space and turn it into linear space, then apply our lighting, and then turn it back into sRGB. This produces a much brighter (accurate) result. It also illuminates a clear ringing artifact of vanilla radiance cascades. We'll do this with the approximate version of `vec3(2.2)` when reading and `vec3(1.0 / 2.2)` when writing. Ths reason behind this is WebGL uses sRGB and we want to work with colors in linear space. Here's a great shadertoy which demonstrates a simple example of why we should work with color in linear space instead of sRGB space.

In our raymarching:

```
vec4 colorSample = texture(sceneTexture, sampleUv);
radDelta += vec4(
  pow(colorSample.rgb, vec3(srgb)),
  colorSample.a
);
```

And our final output:

```
FragColor = vec4(
  !(cascadeIndex > firstCascadeIndex)
    ? totalRadiance.rgb
    : pow(totalRadiance.rgb, vec3(1.0 / srgb)),
  1.0
);
```

Another small fix - it's possible to leak light from one side to the other during the merge step, so let's clamp appropriately to not allow this. You can see this by turning the interval split to zero and drawing along an edge in the previous canvas.

```
// (From before)
vec2 offset = (probeRelativePosition + 0.5) /
sqrtBase;
// Clamp to ensure we don't go outside of any edge
vec2 clamped = clamp(offset, vec2(0.5), upperSize -
0.5);
// And add the clamped offset
vec2 upperUv = (upperPosition + clamped) /
resolution
```

Let's check it out!

Correct SRGB

Base Ray Count4

Cascade Index0

And at this point, this looks genuinely reasonable - if you paint around. But if you make a single dot, like it's setup by default, *especially* with "Correct SRGB" checked, there are some serious ringing artifacts!

And this is still an active area of research. There are a number of approaches to fixing this ringing, but many of them incur a fair amount of overhead (as in doubling the frame time) or cause other artifacts, etc.

And this isn't the only area of active research - pretty much the entire approach is still being actively worked on over in the discord. People are also working on various approaches to Radiance Cascades in 3D. Pretty exciting stuff!

Now that you get it, go checkout the final canvas at the top and see what all the different levers and knobs do in "Additional Controls".

Acknowledgements / Resources

In general, the folks in the Graphics Programming Discord, Radiance Cascades thread were incredibly helpful while I was learning. The creator of Radiance Cascades, Alexander Sannikov, gave great feedback on the issues my implementations had, along with Yaazarai, fad, tmpvar, Mytino, Goobley, Sam and many others, either directly or indirectly. I *really* liked how Yaazarai approached building Radiance Cascades and his work had the biggest direct influence on this work and post.

- Radiance Cascades Paper
- Yaazarai's Blog Post
- The official discord

Appendix: Penumbra with Radiance Cascades

Let's examine what our penumbra looks like with our new method.

It has subtly ringing artifacts in the shadow we saw above, but otherwise looks quite clean and is incredibly cheap to compute in comparison.

Appendix: Additional Hacks

I actually snuck in one more hack. I either have a bug somewhere or am missing something regarding bases other than 4. As you can see the artifacts are *much* worse at 16. So I added this hack which helped a lot. This further enforces that some of the hand-waving we did earlier can be improved. We just multiply the interval and interval lengths by a small amount for larger bases.

```
// Hand-wavy rule that improved smoothing of other base ray counts
float modifierHack = base < 16.0 ? 1.0 : sqrtBase;
```

If you'd like to see the full source for the demo at the top of the page, the source code is immediately after the canvas. Just inspect it there. You can also read the whole post as markdown

- all canvases swapped out for their underlying code.

License

All code contained in this page is under MIT license.