thrive SIGGRAPH2019
LOS ANGELES • 28 JULY – 1 AUGUST

Introduction to Real-time Ray Tracing Part 3

# PRODUCTION-SCALE REAL-TIME RAY TRACING

**Morgan McGuire**

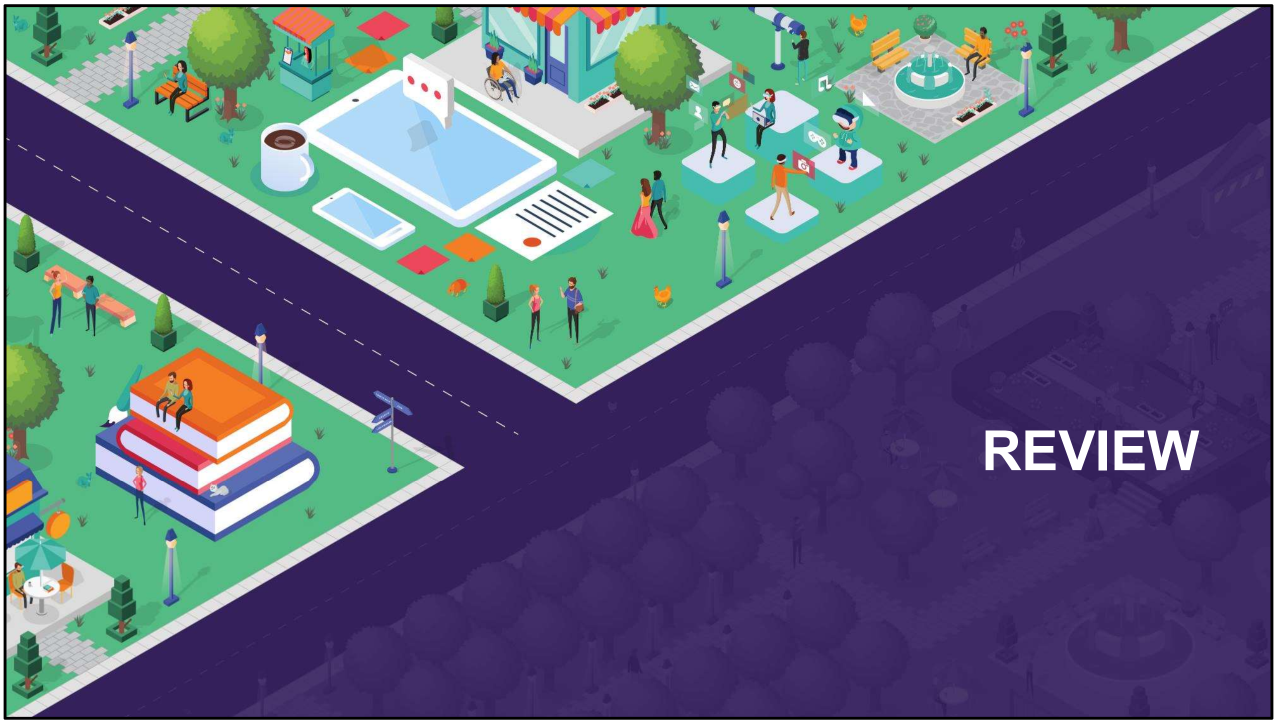
*Distinguished Research Scientist*
*NVIDIA*

We’ve shown you how to build a simple CPU path tracer, and how to accelerate it on a GPU. That’s real-time ray tracing, but it won’t maintain real-time performance for most production applications. I’m thinking of cases like

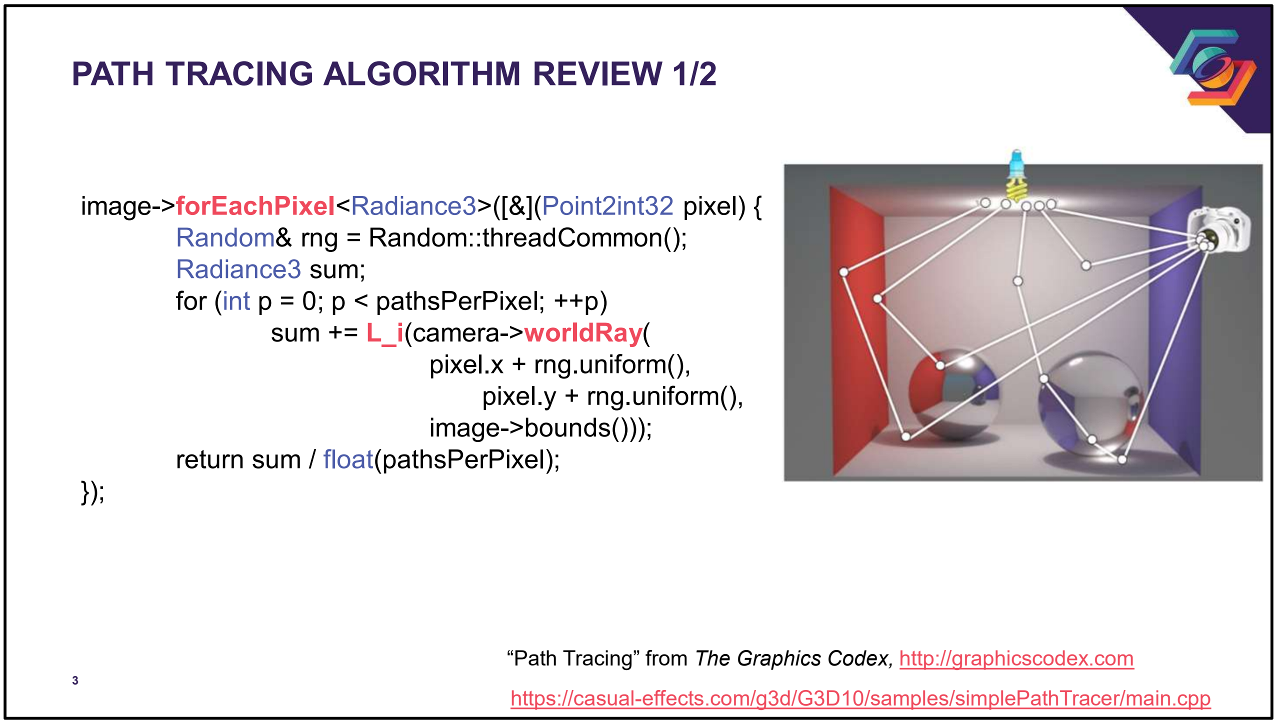
Film previsualization
DCC tools
Architectural walkthrough
Virtual reality
Games

Where there are millions of triangles with primary visibility, complex materials, and you need to render fast enough for interaction, regardless of whether your interactions are one or sixty times per second.

Let’s quickly review the course so far:



# REVIEW



## PATH TRACING ALGORITHM REVIEW 1/2

```
image->forEachPixel<Radiance3>([&](Point2int32 pixel) {
        Random& rng = Random::threadCommon();
        Radiance3 sum;
        for (int p = 0; p < pathsPerPixel; ++p)
                sum += L_i(camera->worldRay(
                                pixel.x + rng.uniform(),
                                    pixel.y + rng.uniform(),
                                image->bounds()));
        return sum / float(pathsPerPixel);
});
```

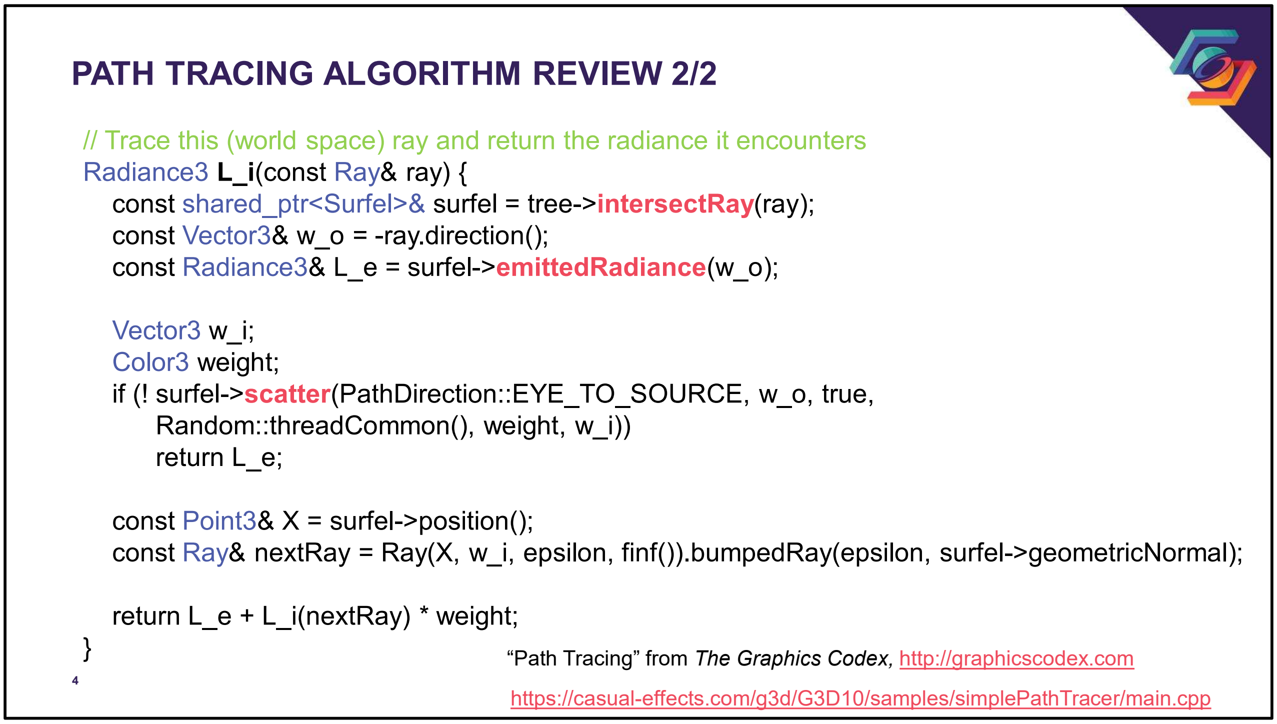
“Path Tracing” from *The Graphics Codex*, http://graphicscodex.com

https://casual-effects.com/g3d/G3D10/samples/simplePathTracer/main.cpp

The path tracing algorithm has two pieces.

The first is an iterator over pixels, which generates a bunch of primary rays for each pixel and averages the light along them.

The more interesting function is the one that evaluates the light coming *back* along a ray towards its origin.
I’ve labeled this function L_i, for “incoming light” following academic notation.



## PATH TRACING ALGORITHM REVIEW 2/2

```
// Trace this (world space) ray and return the radiance it encounters
Radiance3 L_i(const Ray& ray) {
    const shared_ptr<Surfel>& surfel = tree->intersectRay(ray);
    const Vector3& w_o = -ray.direction();
    const Radiance3& L_e = surfel->emittedRadiance(w_o);

    Vector3 w_i;
    Color3 weight;
    if (! surfel->scatter(PathDirection::EYE_TO_SOURCE, w_o, true,
        Random::threadCommon(), weight, w_i))
        return L_e;

    const Point3& X = surfel->position();
    const Ray& nextRay = Ray(X, w_i, epsilon, finf()).bumpedRay(epsilon, surfel->geometricNormal);

    return L_e + L_i(nextRay) * weight;
}
```

“Path Tracing” from *The Graphics Codex*, http://graphicscodex.com

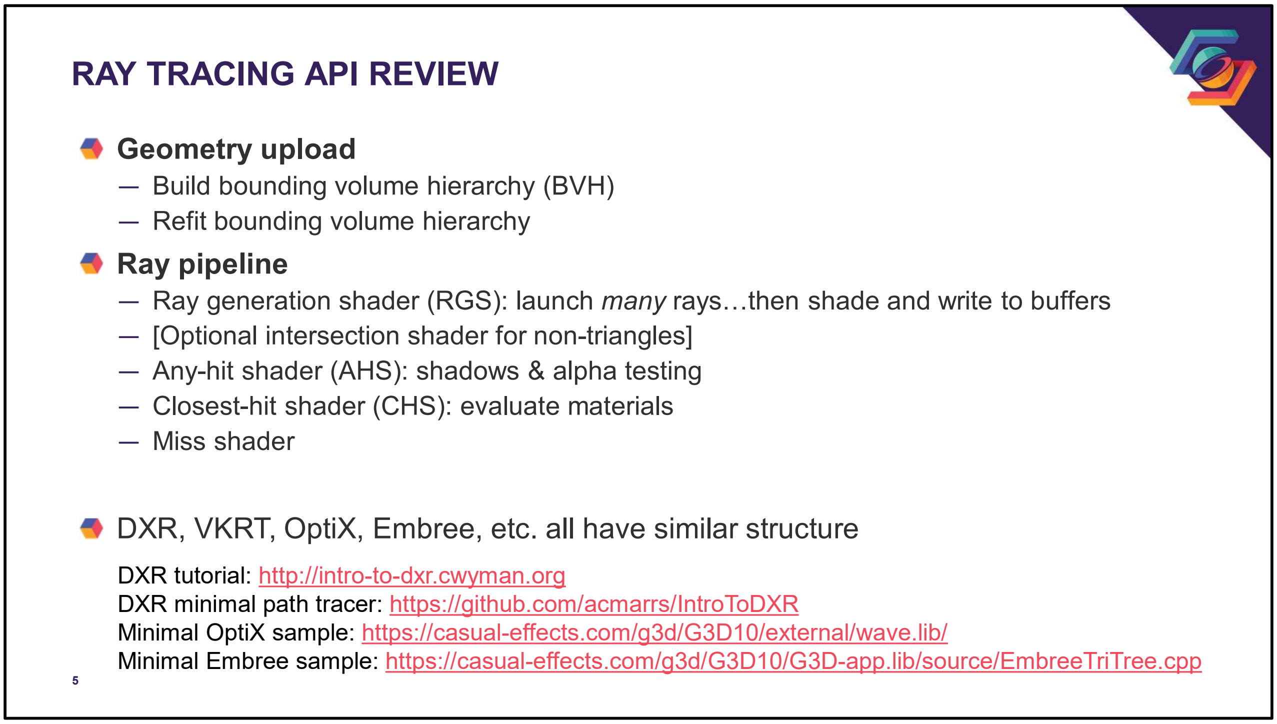
https://casual-effects.com/g3d/G3D10/samples/simplePathTracer/main.cpp

4

The incoming light function does three interesting things.

It intersects the ray with the scene
It computes the amount of light (maybe none) that is emitted at that intersection
And then it either terminates abruptly *or* recursively scatters and evaluates a new ray starting at the hit point.

Everything about the “material” is encoded in the emittedRadiance and scatter functions. You’ve seen simple ones. They’re about to get more complex.



## RAY TRACING API REVIEW

- **Geometry upload**
  - Build bounding volume hierarchy (BVH)
  - Refit bounding volume hierarchy
- **Ray pipeline**
  - Ray generation shader (RGS): launch *many* rays…then shade and write to buffers
  - [Optional intersection shader for non-triangles]
  - Any-hit shader (AHS): shadows & alpha testing
  - Closest-hit shader (CHS): evaluate materials
  - Miss shader
- DXR, VKRT, OptiX, Embree, etc. all have similar structure

DXR tutorial: http://intro-to-dxr.cwyman.org
DXR minimal path tracer: https://github.com/acmarrs/IntroToDXR
Minimal OptiX sample: https://casual-effects.com/g3d/G3D10/external/wave.lib/
Minimal Embree sample: https://casual-effects.com/g3d/G3D10/G3D-app.lib/source/EmbreeTriTree.cpp

That's the algorithm.

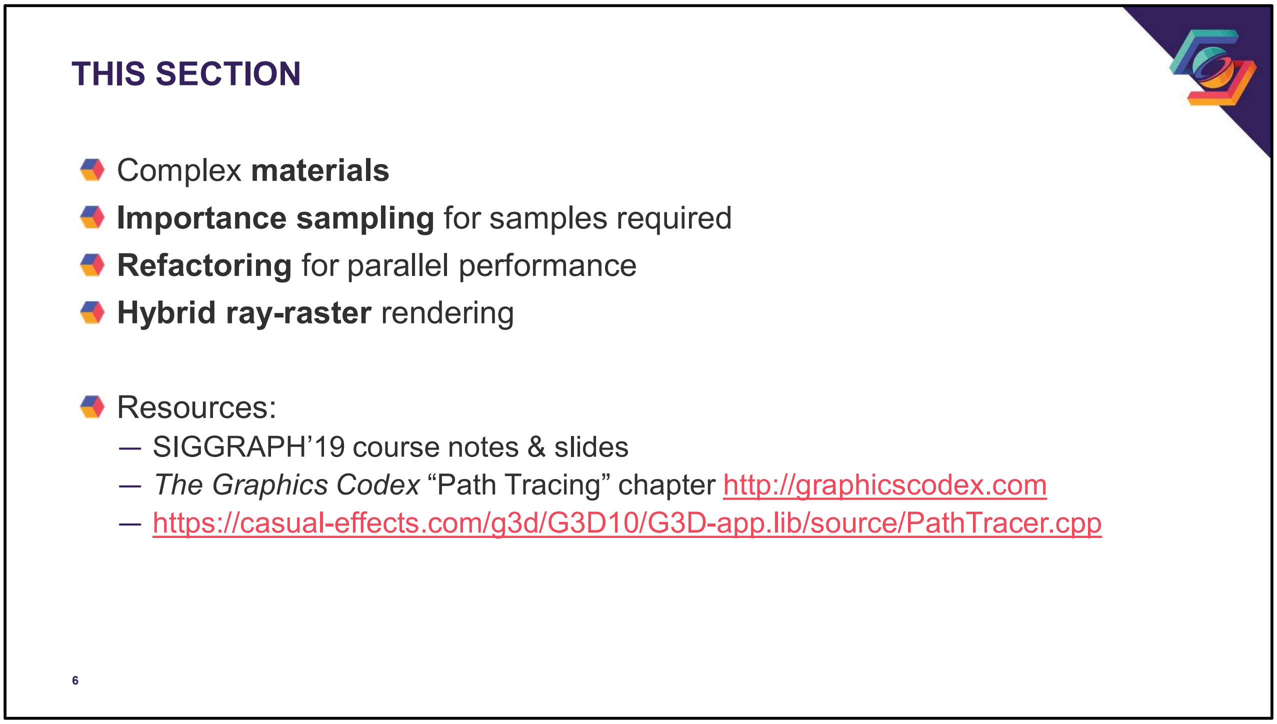
There are accelerated parallel ray tracing APIs for CPU and GPU.
They all work by analogy to the pipeline structure that you know from the rasterization graphics pipeline.

First you upload your geometry on scene creation, and may adjust it a little every frame.
Every frame, you launch a draw call.
That draw call has a number of stages. You're free to use them however you want, but usually the Closest-Hit shader is the interesting one that evaluates materials.

I've put some links here to sample code we wrote for communicating with the various APIs within a real-time graphics program.

You've already seen how to use the DXR API to do a pretty direct and elegant port of the simple CPU ray tracer in this API. And that runs fast for simple scenes, but we need to refactor it entirely to really take advantage of parallel tracing on either a CPU or GPU. We also need to make some algorithmic changes to handle better materials and gain a further speedup to convergence time.

So, let’s spend the rest of the hour doing that refactor.



## THIS SECTION

- Complex **materials**
- **Importance sampling** for samples required
- **Refactoring** for parallel performance
- **Hybrid ray-raster** rendering

- Resources:
  - — SIGGRAPH’19 course notes & slides
  - — *The Graphics Codex* “Path Tracing” chapter http://graphicscodex.com
  - — https://casual-effects.com/g3d/G3D10/G3D-app.lib/source/PathTracer.cpp

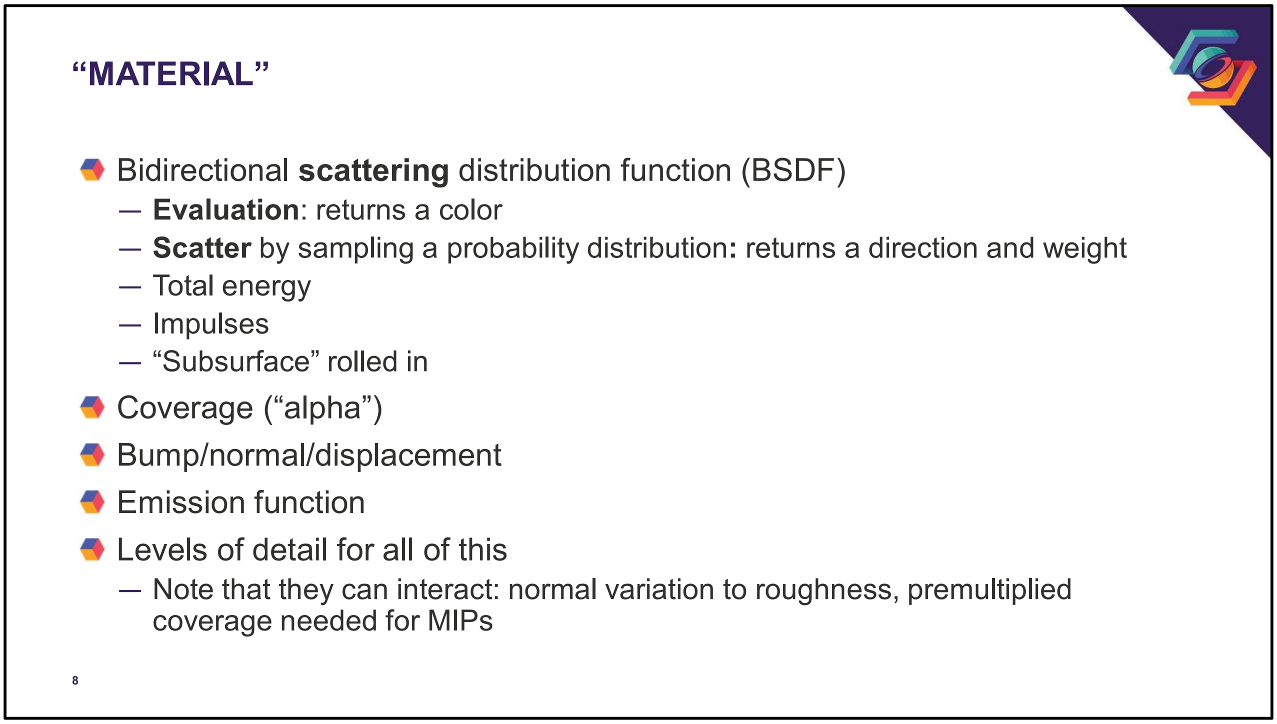
We’ll tackle in this order: materials, sampling, architecture-aware parallelism, and then briefly discuss hybrid rendering strategies suitable for running at high frame rates.

I’ve put full open source code for a very readable path tracer that uses these ideas online. It is a single file (with a lot of helper functions elsewhere).
This is based on one of my textbooks, which includes projects for guiding you through building that code yourself.

To show you that the improvements we’re making are very generalizable, that implementation it has Embree, OptiX, and native C++ BVH code and runs on Windows, Linux, and MacOS…although I admit my production code tends to be in DXR for Windows these days.

The course notes for this are The Graphics Codex, which is my $10 book at this URL.
The code is online right now as are all of the background chapters in the Graphics Codex, and after SIGGRAPH I’ll post the Path Tracing and Hardware Architecture chapters as well as these slides.

# REALISTIC MATERIALS



## “MATERIAL”

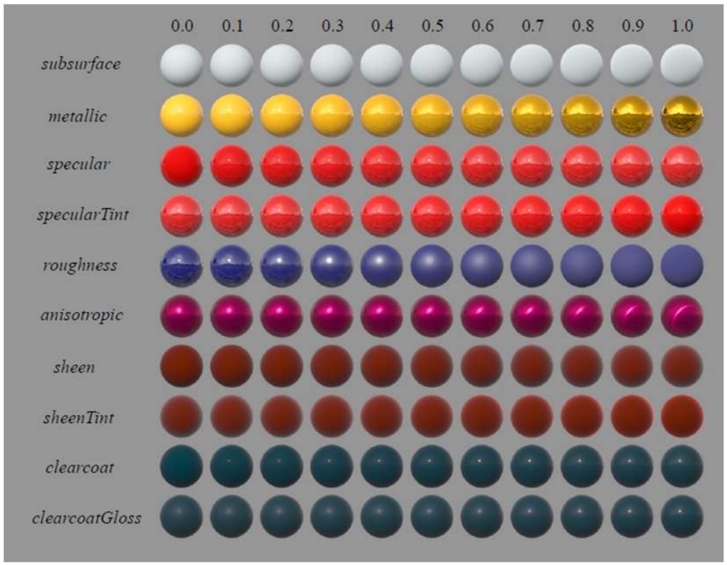
- Bidirectional **scattering** distribution function (BSDF)
  - **Evaluation**: returns a color
  - **Scatter** by sampling a probability distribution**:** returns a direction and weight
  - Total energy
  - Impulses
  - “Subsurface” rolled in
- Coverage (“alpha”)
- Bump/normal/displacement
- Emission function
- Levels of detail for all of this
  - Note that they can interact: normal variation to roughness, premultiplied coverage needed for MIPs

## DISNEY-STYLE MATERIALS

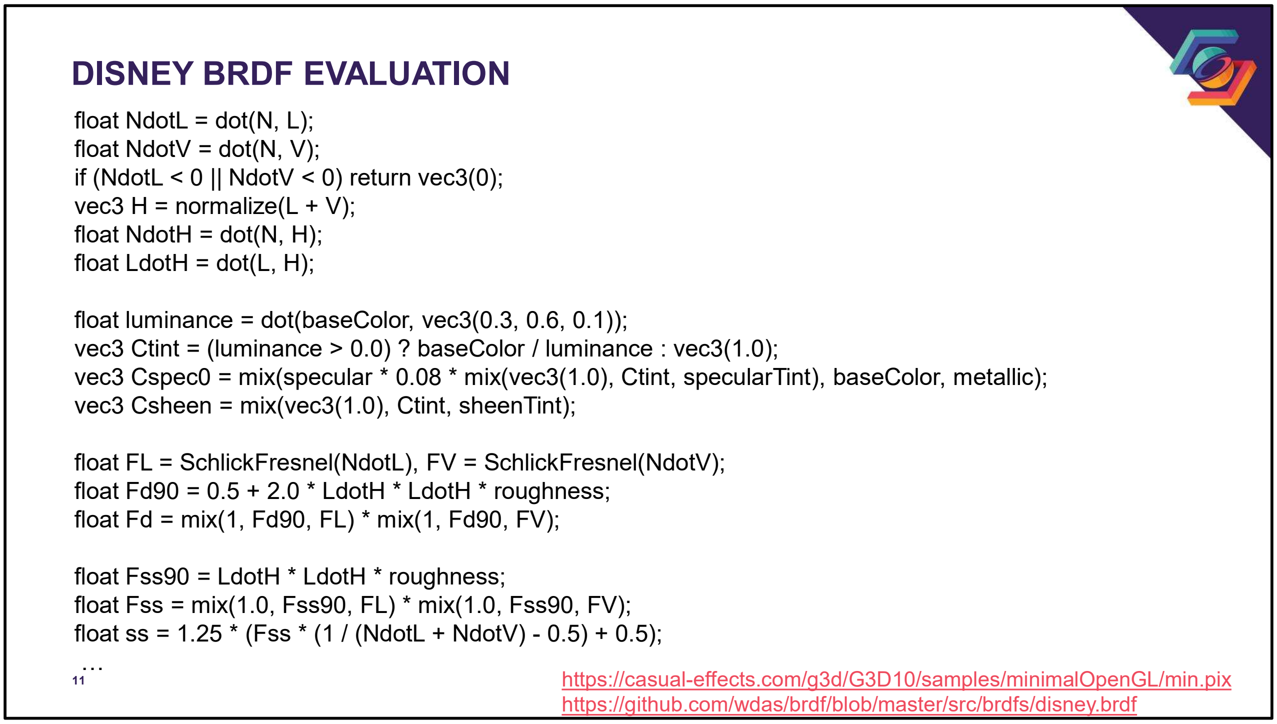
- GGX (GTR) microfacets
- Smith shadowing and masking
- Lambertian or similar
- Anisotropy
- “Layers”

- Similar to UE4, Unity PBR materials, various extensions to “Disney 2” with different remapping and subsurface; point is-- everyone’s converging on the same thing.

## PARAMETERS (FOR EACH TEXEL)



https://github.com/wdas/brdf/blob/master/src/brdfs/disney.brdf
http://blog.selfshadow.com/publications/s2012-shading-course/burley/s2012_pbs_disney_brdf_notes_v3.pdf



## DISNEY BRDF EVALUATION

```
float NdotL = dot(N, L);
float NdotV = dot(N, V);
if (NdotL < 0 || NdotV < 0) return vec3(0);
vec3 H = normalize(L + V);
float NdotH = dot(N, H);
float LdotH = dot(L, H);

float luminance = dot(baseColor, vec3(0.3, 0.6, 0.1));
vec3 Ctint = (luminance > 0.0) ? baseColor / luminance : vec3(1.0);
vec3 Cspec0 = mix(specular * 0.08 * mix(vec3(1.0), Ctint, specularTint), baseColor, metallic);
vec3 Csheen = mix(vec3(1.0), Ctint, sheenTint);

float FL = SchlickFresnel(NdotL), FV = SchlickFresnel(NdotV);
float Fd90 = 0.5 + 2.0 * LdotH * LdotH * roughness;
float Fd = mix(1, Fd90, FL) * mix(1, Fd90, FV);

float Fss90 = LdotH * LdotH * roughness;
float Fss = mix(1.0, Fss90, FL) * mix(1.0, Fss90, FV);
float ss = 1.25 * (Fss * (1 / (NdotL + NdotV) - 0.5) + 0.5);
…
```

https://casual-effects.com/g3d/G3D10/samples/minimalOpenGL/min.pix
https://github.com/wdas/brdf/blob/master/src/brdfs/disney.brdf

Evaluating this code isn't bad. There's a *lot* of operations, but it is straightforward, and there's plenty of reference code.

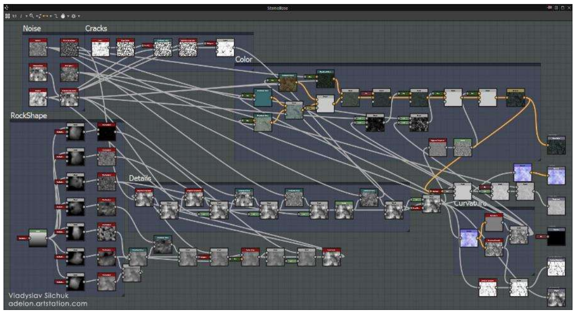
# DISNEY BRDF EVALUATION

```
…
float aspect = sqrt(1.0 - anisotropic * 0.9);
float ax = max(0.001, square(roughness) / aspect);
float ay = max(0.001, square(roughness) * aspect);
float Ds = GTR2_aniso(NdotH, dot(H, X), dot(H, Y), ax, ay);
float FH = SchlickFresnel(LdotH);
vec3 Fs = mix(Cspec0, vec3(1.0), FH);
float Gs  = SmithG_GGX_aniso(NdotL, dot(L, X), dot(L, Y), ax, ay) *
            SmithG_GGX_aniso(NdotV, dot(V, X), dot(V, Y), ax, ay);
vec3 Fsheen = FH * sheen * Csheen;
float Dr = GTR1(NdotH, mix(0.1, 0.001, clearcoatGloss));
float Fr = mix(0.04, 1.0, FH);
float Gr = SmithG_GGX(NdotL, 0.25) * SmithG_GGX(NdotV, 0.25);

return ((1.0 / PI) * mix(Fd, ss, subsurface) * baseColor + Fsheen) * (1.0 - metallic) +
    Gs * Fs * Ds + 0.25 * clearcoat * Gr * Fr * Dr;
```

https://casual-effects.com/g3d/G3D10/samples/minimalOpenGL/min.pix
https://github.com/wdas/brdf/blob/master/src/brdfs/disney.brdf

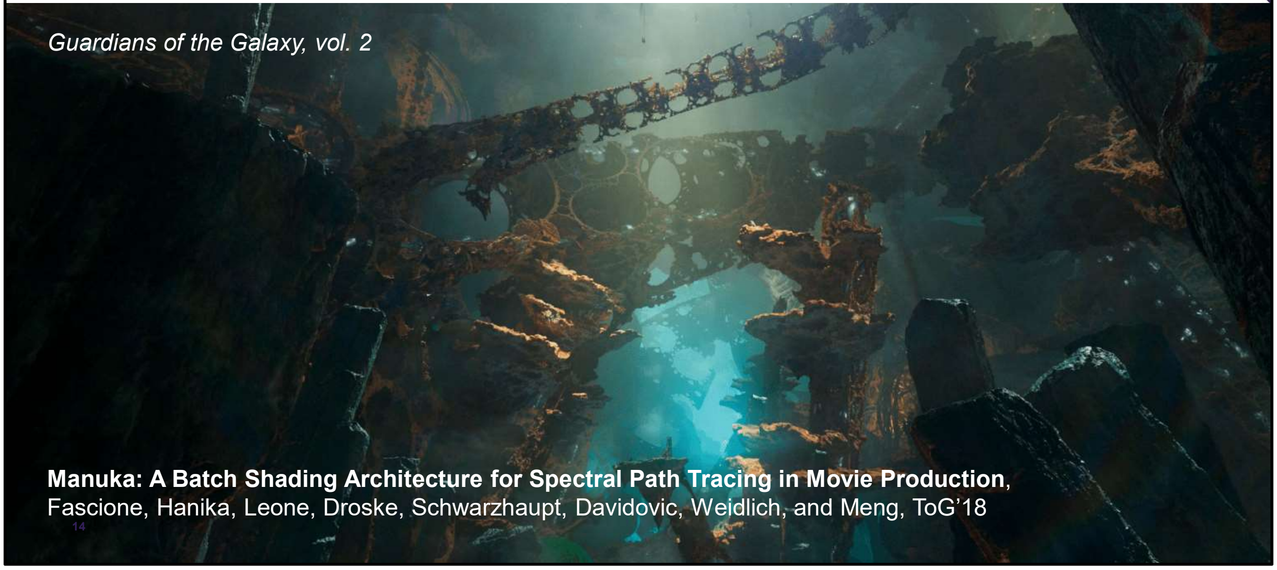
## SHADER GRAPHS

- This is where the parameters come from
- Note the artist meaning of “Layers”: hide tiling artifacts, not physical layers
- Great for artist workflow
- Bad for GPU efficiency
- Bake these down to simple textures if you can



Vladyslav Silchuk
adelon.artstation.com

# PROCEDURAL MATERIAL & GEOMETRY BAKING

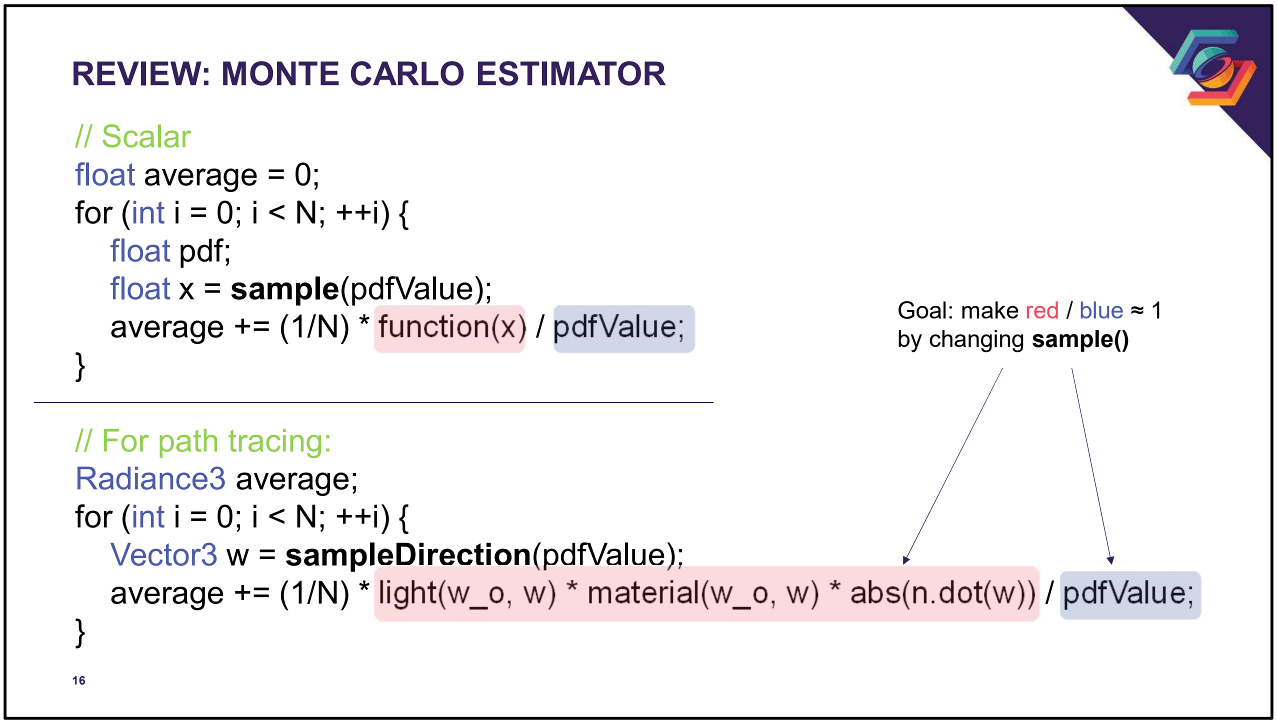


*Guardians of the Galaxy, vol. 2*

**Manuka: A Batch Shading Architecture for Spectral Path Tracing in Movie Production**,
Fascione, Hanika, Leone, Droske, Schwarzhaupt, Davidovic, Weidlich, and Meng, ToG'18

# IMPORTANCE SAMPLING

Recall the core monte carlo estimator:



## REVIEW: MONTE CARLO ESTIMATOR

```
// Scalar
float average = 0;
for (int i = 0; i < N; ++i) {
   float pdf;
   float x = sample(pdfValue);
   average += (1/N) * function(x) / pdfValue;
}
```

---

```
// For path tracing:
Radiance3 average;
for (int i = 0; i < N; ++i) {
   Vector3 w = sampleDirection(pdfValue);
   average += (1/N) * light(w_o, w) * material(w_o, w) * abs(n.dot(w)) / pdfValue;
}
```

Goal: make red / blue ≈ 1
by changing **sample()**

Sample has two return values: the sample which is a scattered light direction in the path tracer, and the differential probability with which it chose that value. You can implement the sampler any way that you want as long as you are honest about the probability with which you computed it.

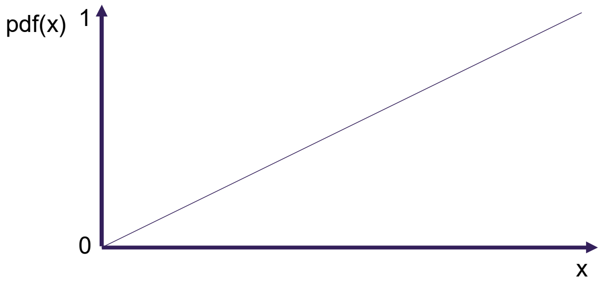
Now, there's noise in anything that takes random samples.

You can decrease noise by increasing N, since we average the results. But your run-time is proportional to N and noise decreases like sqrt(N), so that is a losing game after a while.

**Winning game**: If we can make sampler probability distribution proportional to the "integrand", then we get maximum information per sample and eliminate noise.

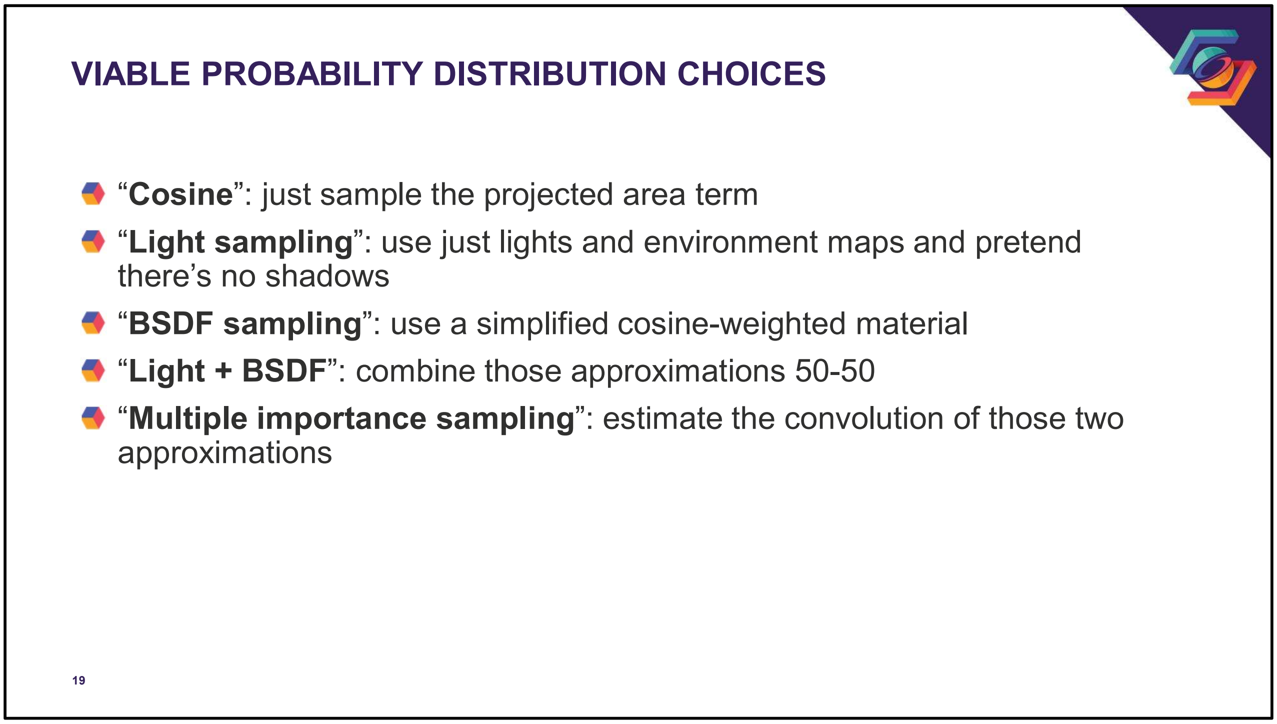
## GRADIENT EXAMPLE

- pdf(x) = x
- Uniform gives 0.1, 0.7, 0.8, 0.3, 0.5, … ~= 0.5
- *Perfect* importance sampling gives 0.5, 0.5, 0.5, 0.5,…! No noise.



## WHY WE DON’T DO PERFECT IMPORTANCE SAMPLING

- We don’t know what the light distribution is. Figuring that out is called “rendering”! Shadows and global illumination are what makes this hard.
- We don’t know the material ⊗ light convolution
- We don’t even have the material as an explicit value. We usually evaluate it on the fly.
- You can’t analytically sample proportional to a complicated function.
  - Analytic solutions for simple cases
  - Giant memory data structure CDFs for the general case
  - Lots of approximations needed
  - Look for !/$ in implementation, runtime cost, memory cost, sample efficiency, flexibility

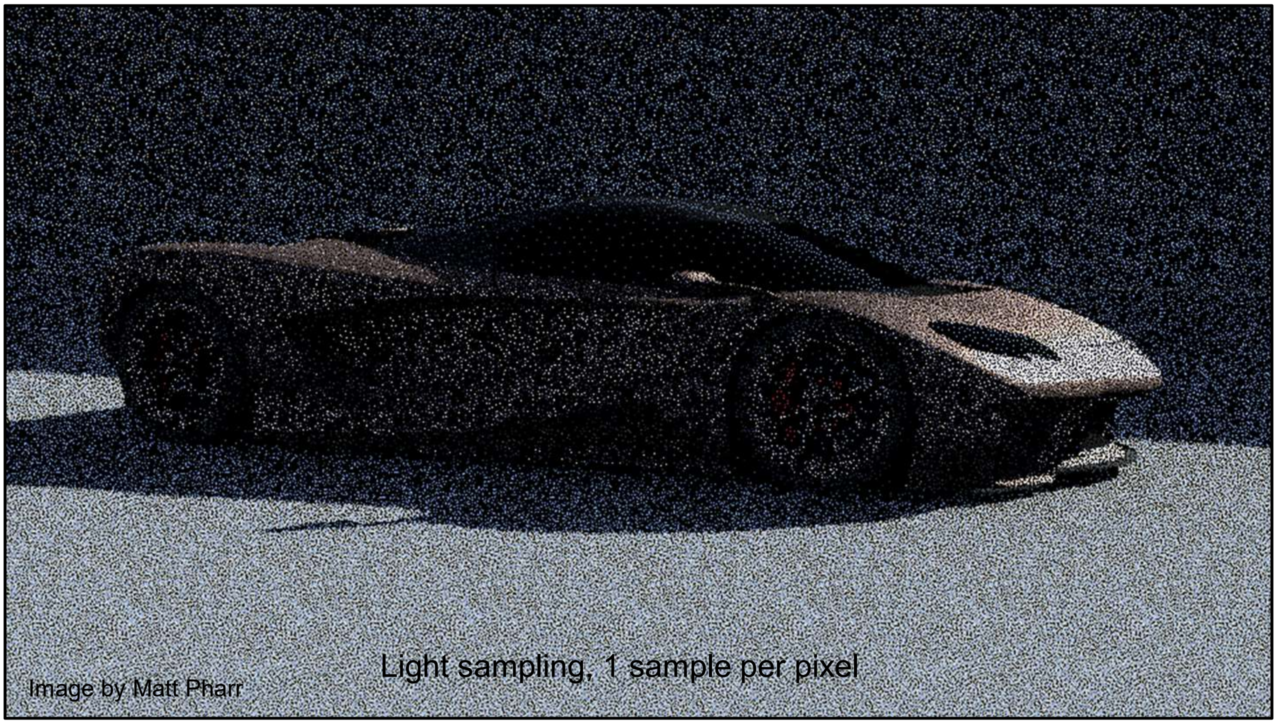


## VIABLE PROBABILITY DISTRIBUTION CHOICES

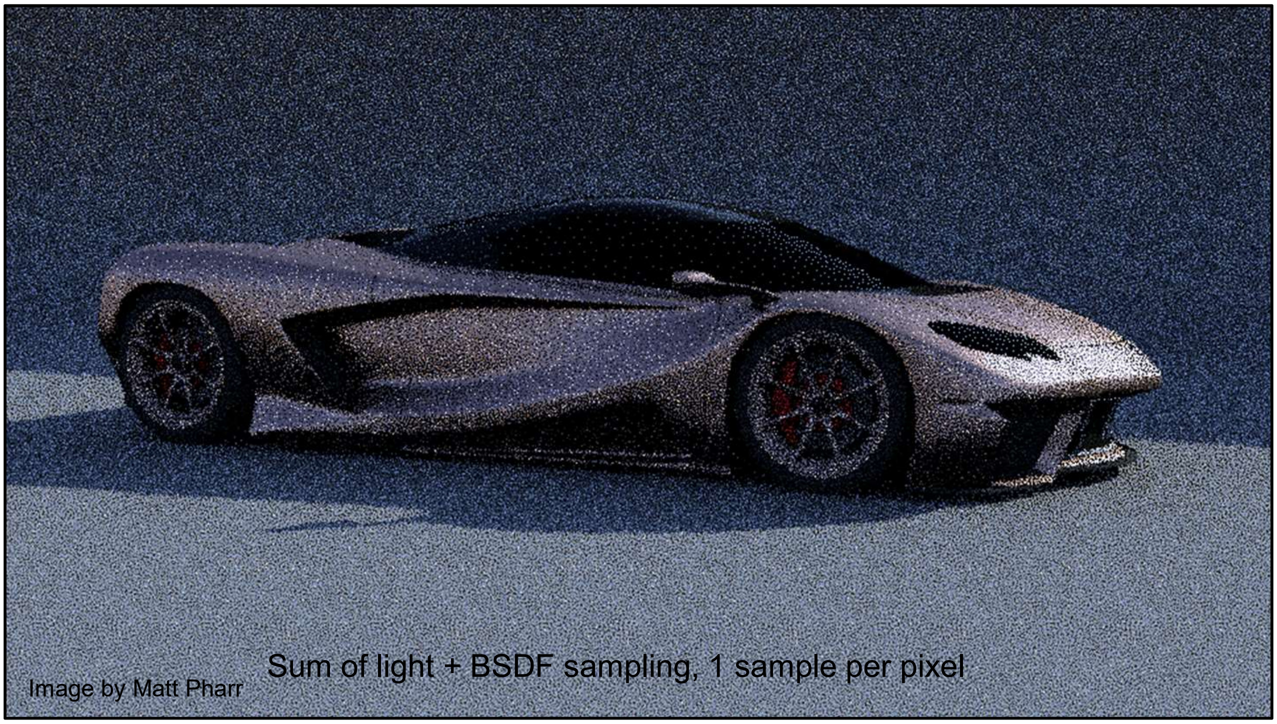
- “**Cosine**”: just sample the projected area term
- “**Light sampling**”: use just lights and environment maps and pretend there’s no shadows
- “**BSDF sampling**”: use a simplified cosine-weighted material
- “**Light + BSDF**”: combine those approximations 50-50
- “**Multiple importance sampling**”: estimate the convolution of those two approximations

Cosine is actually fine for really matte scenes with large lights, too noisy for everything else

These are sort of increasingly good, but complicated.



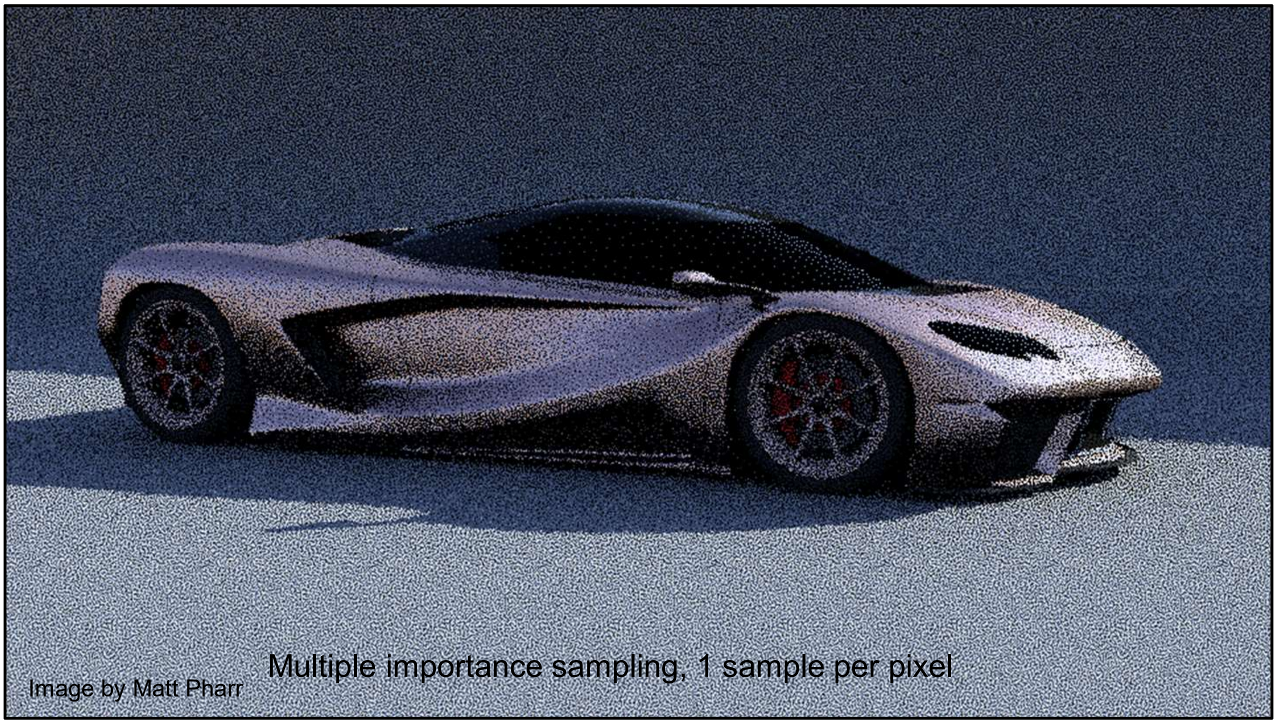
If you cast rays where you think the light is, then highlights can be sampled well, but you get a ton of noise everywhere



Sum of light + BSDF sampling, 1 sample per pixel

Image by Matt Pharr

If instead we cast half the rays towards the light and half according to the BSDF, then we get better use from the rays



Multiple importance sampling, 1 sample per pixel

Image by Matt Pharr

The best thing to do would be to estimate the convolution of the lighting and the material. If the lighting is just an environment map and single source like this, then you can do that well.

But it is a tricky to implement and only helps if you don’t have many shadows or complex global illumination.

… and the whole point of path tracing was that we want complex global illumination, so I usually take the easy way out…

## WHAT I ACTUALLY DO: APPROXIMATE BSDF + LIGHT

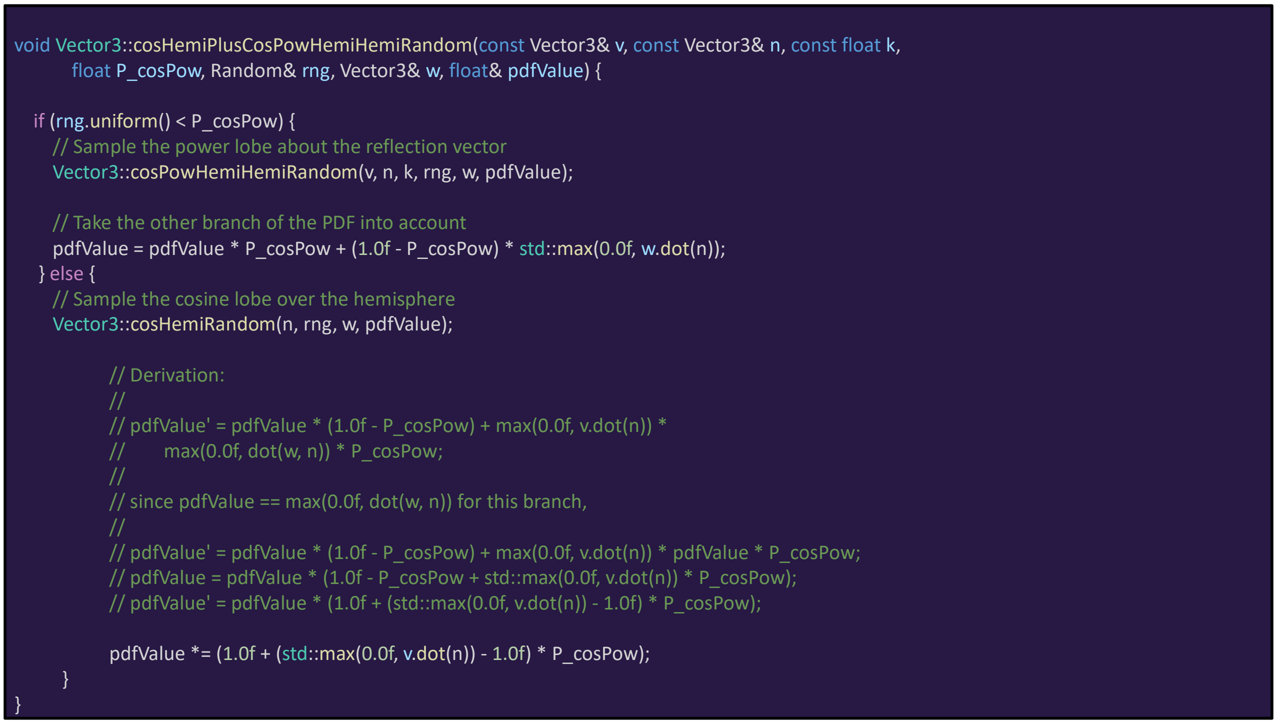
- Compute explicit direct illumination with a shadow ray
- Sample perfect impulses proportional to their total energy
- Sample the rest with pdf(w) ∝ abs(n.dot(w)) * (pow(max(n.dot(w_h), 0), k) + C)
  - Similar shape to the Disney cos-BSDF, but can be analytically sampled
  - Multiply by the ratio of the real BSDF to the pdf value so that we converge correctly
  - Compute based on average RGB and then scale by “color”

https://casual-effects.com/g3d/G3D10/G3D-base.lib/source/Vector3.cpp
https://casual-effects.com/g3d/G3D10/G3D-app.lib/source/Surfel.cpp
https://casual-effects.com/g3d/G3D10/G3D-app.lib/source/UniversalSurfel.cpp

Making the material importance sampler numerically stable is pretty tricky. It does a lot of multiplying and dividing by numbers with radically different magnitudes and then expects them to come out ok and cancel. If you aren’t careful, it can blow up to infinity or nan very easily. We spent a long time making this reference code numerically stable, which involves a lot of carrying around separate numerators and denominators until you expect their magnitudes to be similar as well as clamping and offsetting by epsilon.

Pete advocates casting a true recursive ray towards the light 50% of the time and a BSDF ray 50% of the time, instead of a shadow ray plus a recursive ray. His way, even if the light is not visible, the light ray cast gives you a real hit that you can shade. There are a few tradeoffs. This doesn’t work for point lights, and if the light isn’t visible you might be wasting that ray in a direction that doesn’t reflect any energy…and shadow rays are *really* cheap on GPUs now.

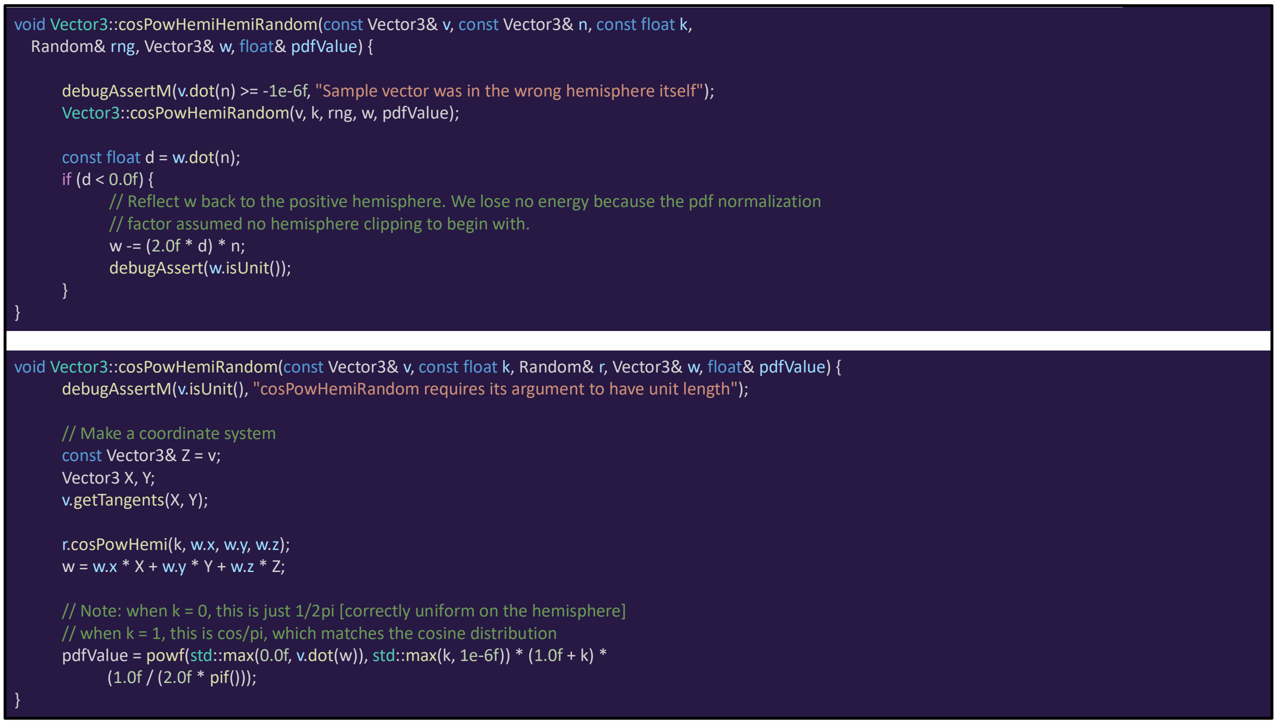
Cool kids do real multiple-importance sampling and use CDFs, spherical harmonics, etc..
Maybe I will some day, too.
But CDFs are hard to use on GPUs because they eat bandwidth. I lean towards using importon or photon mapping if you want to do MIS on a GPU for real-time tracing/



```
void Vector3::cosHemiPlusCosPowHemiHemiRandom(const Vector3& v, const Vector3& n, const float k,
        float P_cosPow, Random& rng, Vector3& w, float& pdfValue) {

  if (rng.uniform() < P_cosPow) {
    // Sample the power lobe about the reflection vector
    Vector3::cosPowHemiHemiRandom(v, n, k, rng, w, pdfValue);

    // Take the other branch of the PDF into account
    pdfValue = pdfValue * P_cosPow + (1.0f - P_cosPow) * std::max(0.0f, w.dot(n));
  } else {
    // Sample the cosine lobe over the hemisphere
    Vector3::cosHemiRandom(n, rng, w, pdfValue);

          // Derivation:
          //
          // pdfValue' = pdfValue * (1.0f - P_cosPow) + max(0.0f, v.dot(n)) *
          //       max(0.0f, dot(w, n)) * P_cosPow;
          //
          // since pdfValue == max(0.0f, dot(w, n)) for this branch,
          //
          // pdfValue' = pdfValue * (1.0f - P_cosPow) + max(0.0f, v.dot(n)) * pdfValue * P_cosPow;
          // pdfValue = pdfValue * (1.0f - P_cosPow + std::max(0.0f, v.dot(n)) * P_cosPow);
          // pdfValue' = pdfValue * (1.0f + (std::max(0.0f, v.dot(n)) - 1.0f) * P_cosPow);

          pdfValue *= (1.0f + (std::max(0.0f, v.dot(n)) - 1.0f) * P_cosPow);
      }
}
```

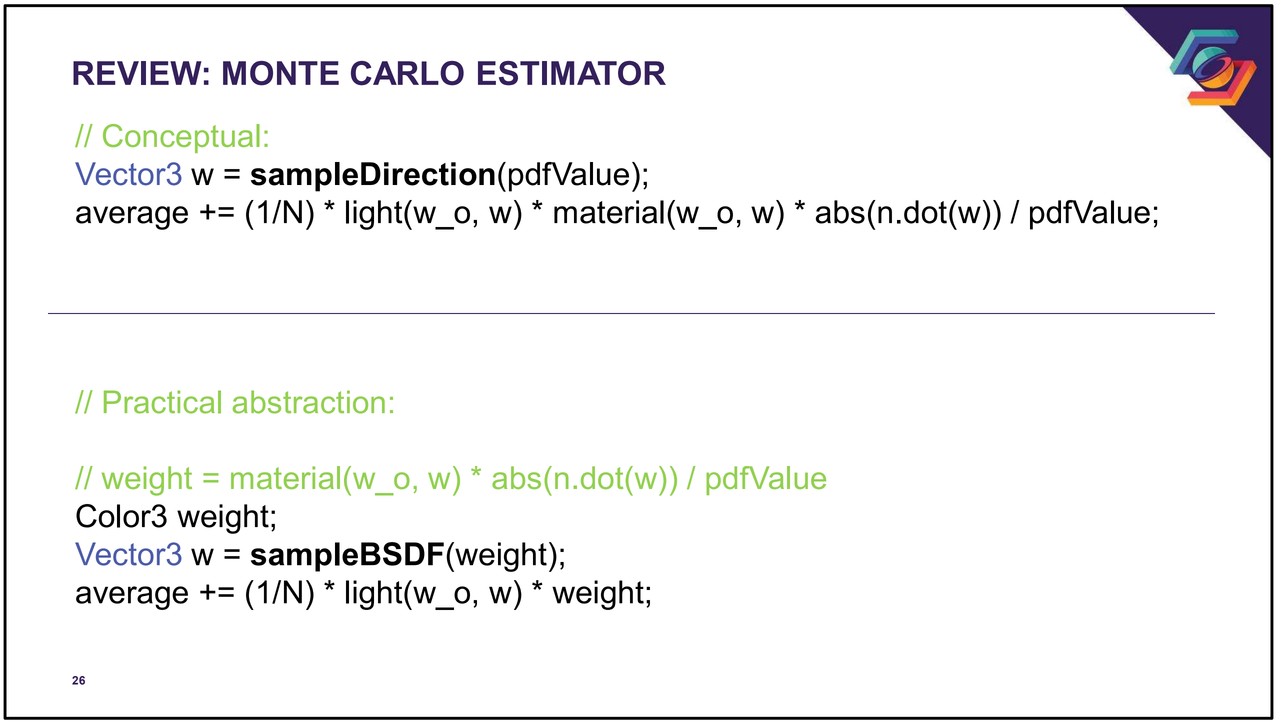
Here’s some code as a walkthrough. It is actually about 20 lines of code, but the abstractions and comments that make it real production code expand it.

I’m not going through this in detail right now because I linked to the online repo where you can just grab it directly, but I wanted to show you that there’s nothing slow or scary to implement hidden in there.



```
void Vector3::cosPowHemiHemiRandom(const Vector3& v, const Vector3& n, const float k,
   Random& rng, Vector3& w, float& pdfValue) {

        debugAssertM(v.dot(n) >= -1e-6f, "Sample vector was in the wrong hemisphere itself");
        Vector3::cosPowHemiRandom(v, k, rng, w, pdfValue);

        const float d = w.dot(n);
        if (d < 0.0f) {
                // Reflect w back to the positive hemisphere. We lose no energy because the pdf normalization
                // factor assumed no hemisphere clipping to begin with.
                w -= (2.0f * d) * n;
                debugAssert(w.isUnit());
        }
}
```

```
void Vector3::cosPowHemiRandom(const Vector3& v, const float k, Random& r, Vector3& w, float& pdfValue) {
        debugAssertM(v.isUnit(), "cosPowHemiRandom requires its argument to have unit length");

        // Make a coordinate system
        const Vector3& Z = v;
        Vector3 X, Y;
        v.getTangents(X, Y);

        r.cosPowHemi(k, w.x, w.y, w.z);
        w = w.x * X + w.y * Y + w.z * Z;

        // Note: when k = 0, this is just 1/2pi [correctly uniform on the hemisphere]
        // when k = 1, this is cos/pi, which matches the cosine distribution
        pdfValue = powf(std::max(0.0f, v.dot(w)), std::max(k, 1e-6f)) * (1.0f + k) *
                (1.0f / (2.0f * pif()));
}
```



## REVIEW: MONTE CARLO ESTIMATOR

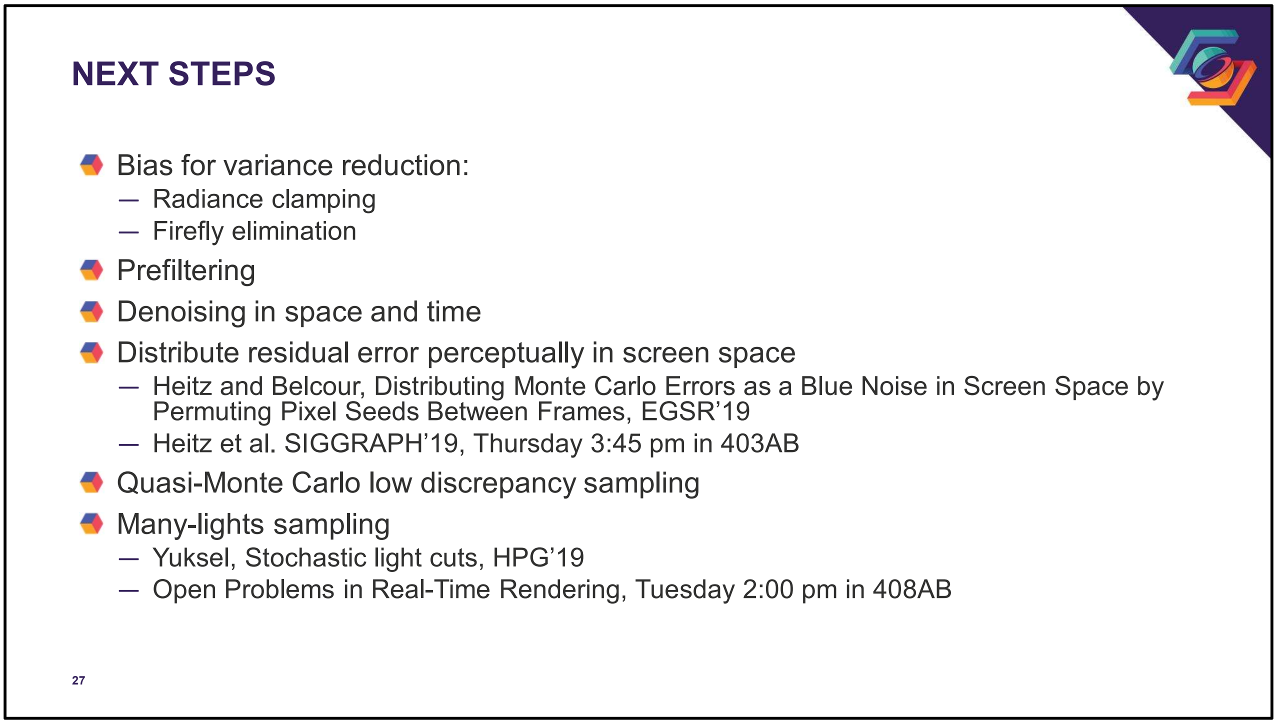
```
// Conceptual:
Vector3 w = sampleDirection(pdfValue);
average += (1/N) * light(w_o, w) * material(w_o, w) * abs(n.dot(w)) / pdfValue;
```

---

```
// Practical abstraction:

// weight = material(w_o, w) * abs(n.dot(w)) / pdfValue
Color3 weight;
Vector3 w = sampleBSDF(weight);
average += (1/N) * light(w_o, w) * weight;
```

The way this is usually abstracted is to have the sampler compute the whole ratio for you, for example, like this if we go back to the core code. So, that “WEIGHT” value is the function we wanted divided by the function that we used. You’ll see this a lot in the second half of this talk.

OK, so that’s how you get pretty fast sampling.
If you need more speed, I wouldn’t make the material or per sampler more complicated. Instead, I would look at other parts of the system:



## NEXT STEPS

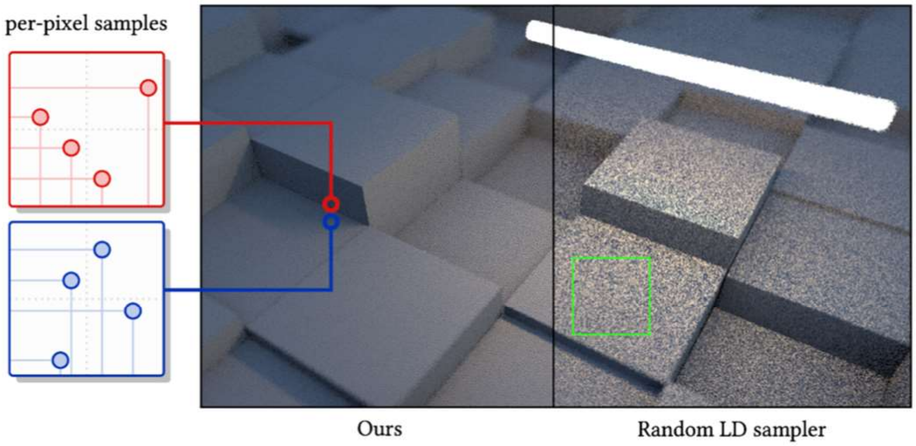
- Bias for variance reduction:
  - — Radiance clamping
  - — Firefly elimination
- Prefiltering
- Denoising in space and time
- Distribute residual error perceptually in screen space
  - — Heitz and Belcour, Distributing Monte Carlo Errors as a Blue Noise in Screen Space by Permuting Pixel Seeds Between Frames, EGSR'19
  - — Heitz et al. SIGGRAPH'19, Thursday 3:45 pm in 403AB
- Quasi-Monte Carlo low discrepancy sampling
- Many-lights sampling
  - — Yuksel, Stochastic light cuts, HPG'19
  - — Open Problems in Real-Time Rendering, Tuesday 2:00 pm in 408AB

I've ordered these from easiest to hardest in terms of implementation complexity.

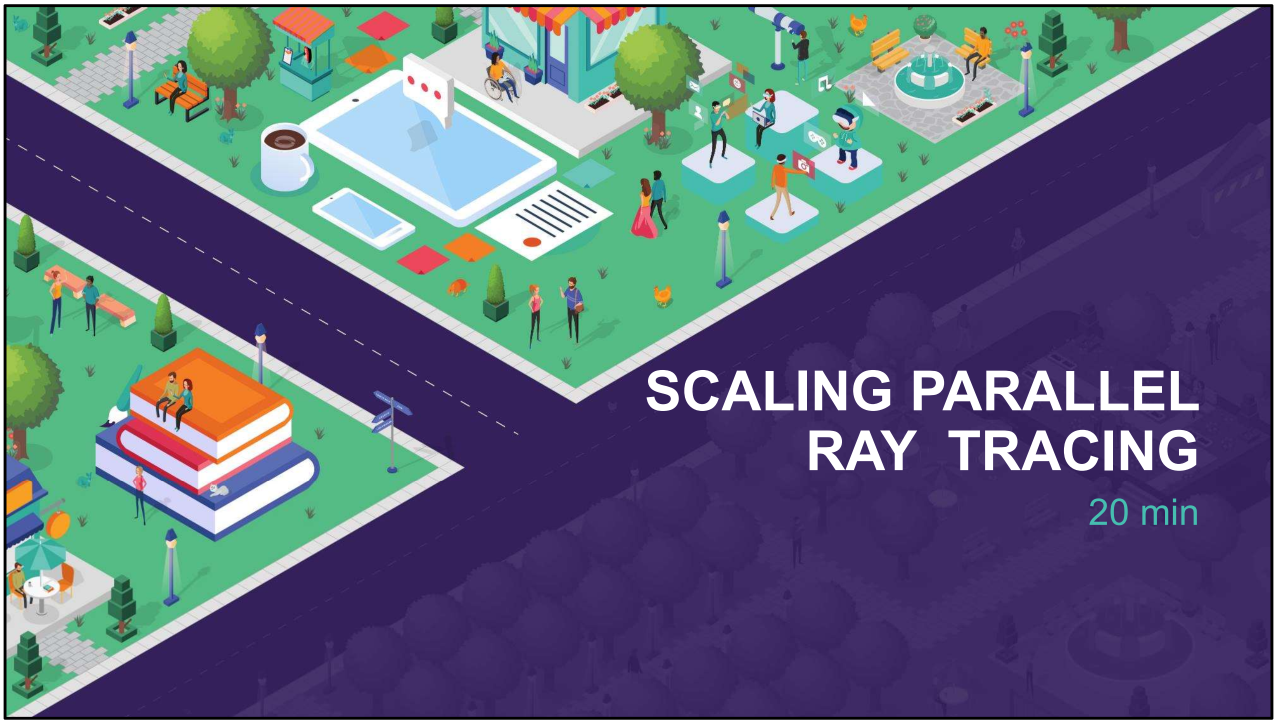
Most of our rays are just there to reduce noise. One source of noise is "fireflies"
A problem with Monte Carlo is that if you happen to sample a low probability part of the PDF and then that ray hits something bright in the scene, you'll end up dividing the bright value by a really small number and the result can explode. On average, this is indeed correct and that energy should be in the scene, but in practice it means that every now and then you get one super bright pixel for one frame, and it flickers and looks terrible. So, don't do that. Clamp the minimum PDF value, the maximum radiance, run an outlier detection at the end…and just throw away that energy to make the scene stable. Production renderers all do this.

The simplest prefiltering is MIP-maps on your textures in the materials. Trace rays, but keep track of how wide they would be if they were cones covering the pixel and then use that to find the texel footprint on each triangle.

The other things but they have large payoffs in many cases but get increasingly difficult because of how invasive they are to the structure of the ray tracer. They're also active research areas, so you generally want to read about them in papers instead of textbooks.



***A Low-Discrepancy Sampler that Distributes Monte Carlo Errors as a Blue Noise in Screen Space*, Heitz et al. 2019**

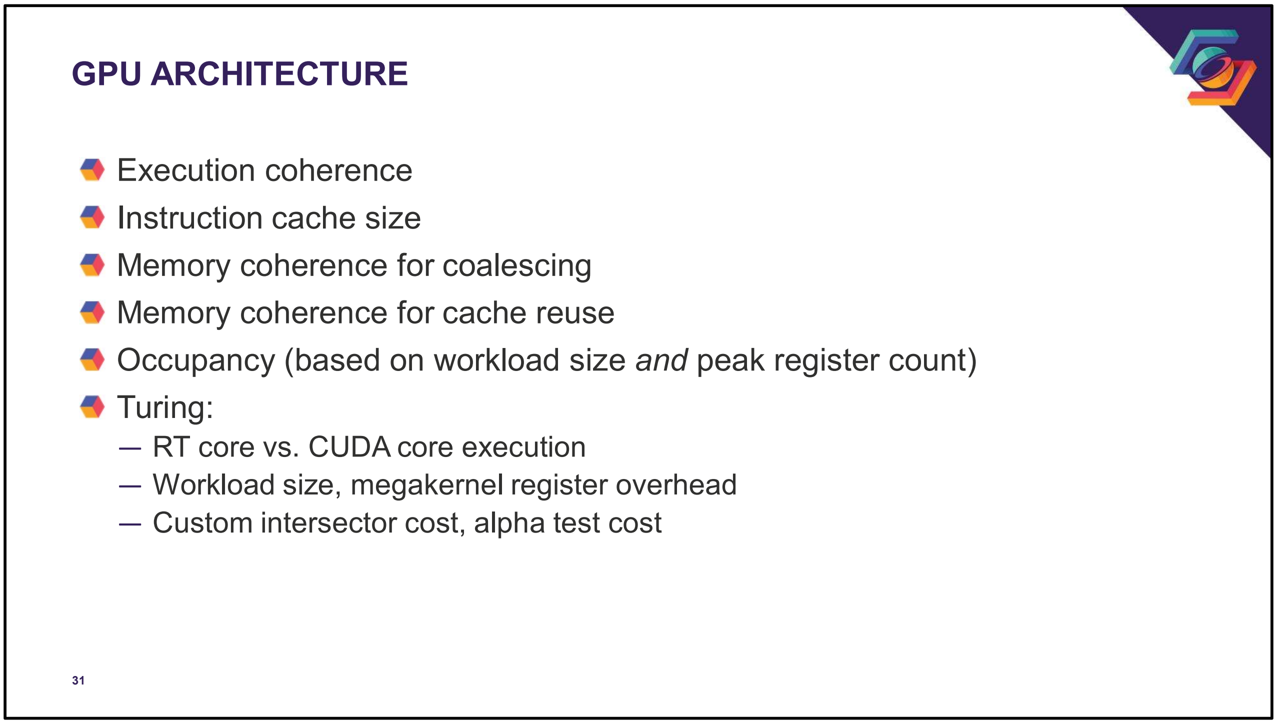


# SCALING PARALLEL RAY TRACING

20 min

## CPU ARCHITECTURE

- Tris vs. quads
- Packet tracing and medium SIMD size
- Custom intersector overhead…separate alpha testing trace/materials



## GPU ARCHITECTURE

- Execution coherence
- Instruction cache size
- Memory coherence for coalescing
- Memory coherence for cache reuse
- Occupancy (based on workload size *and* peak register count)
- Turing:
  - — RT core vs. CUDA core execution
  - — Workload size, megakernel register overhead
  - — Custom intersector cost, alpha test cost

## GENERAL ADVICE

- **Keep closest-hit shader short**
  - — Instruction cache
  - — Low register count
  - — Move material evaluation, shading, and framebuffer write to the ray generation shader
- **Keep computation coherent**
  - — Wavefront tracing to maximize coherence and minimize registers
  - — Ray direction and origin coherence for traversal memory efficiency
  - — Sort/bin rays based on material
- **Beware of near-misses**
  - — Triangle fans
  - — Large triangles
- **Short rays are fast**
  - — Disable backface culling – shortening rays is good for a BVH!
  - — Bound t_min and t_max as tight as possible (easy for shadow rays)

thrive
SIGGRAPH2019
LOS ANGELES • 28 JULY – 1 AUGUST

# SCALABLE REFACTOR

## Case Study

https://casual-effects.com/g3d/G3D10/G3D-app.lib/source/PathTracer.cpp

I’m going write this on the CPU for simplicity of syntax since I want the code to really compile.
I do the exact same thing on the GPU, where each threadpool iteration launch becomes a ray gen or compute shader call.

## ONE PIXEL → ONE PATH PER “THREAD”

Before

```
// Compute the average radiance into this pixel
image->forEachPixel<Radiance3>([&](Point2int32 pixel) {…
      for (int p = 0; p < pathsPerPixel; ++p) {
            const Point2 pixelPos(pixel.x + rng.uniform(), pixel.y + rng.uniform());
            sum += L_i(camera->worldRay(pixelPos.x, pixelPos.y, image->bounds()));
      }

➡    return sum / float(pathsPerPixel);
});
```

After

```
for (int p = 0; p < pathsPerPixel; ++p) {
      image->forEachPixel<Radiance3>([&](Point2int32 pixel) {…
            const Point2 pixelPos(pixel.x + rng.uniform(), pixel.y +rng.uniform())
            const Radiance3& L = L_i(camera->worldRay(pixelPos, image->bounds()));
      ➡    rawImage->bilinearAdd(pixelPos, L);
            weightImage->bilinearAdd(pixelPos, 1);
      }}); }
// Normalize by weights later…
```

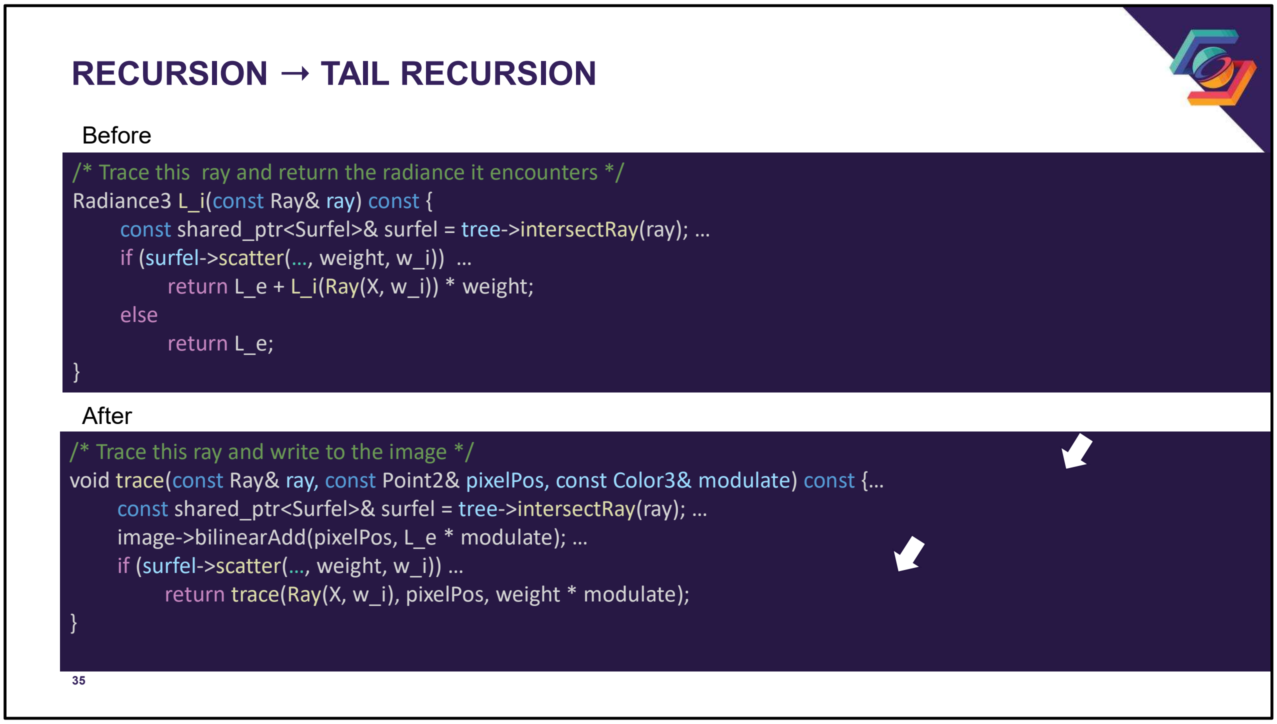
Instead of processing each PIXEL in a “thread”, we’re going to process each PATH on a thread.

(These aren’t real operating system threads, which have high overhead…they are CPU thread-pooled work or GPU SIMD warp lanes, but the APIs make them *look* like threads, so we’ll keep pretending that).

**Each path will remember its source location on the image plane and bilinearly interpolate into it.**
This gives us a lot more parallelism to work with, which will help to “fill the machine”. It also sets up the following steps because we can now deal with paths individually

In practice you don’t even need that outer FOR loop…just spawn a total number of threads equal to pixels times paths per pixel.

Bilinear add uses atomic addition, so you don’t have to worry about the race condition.



## RECURSION → TAIL RECURSION

Before

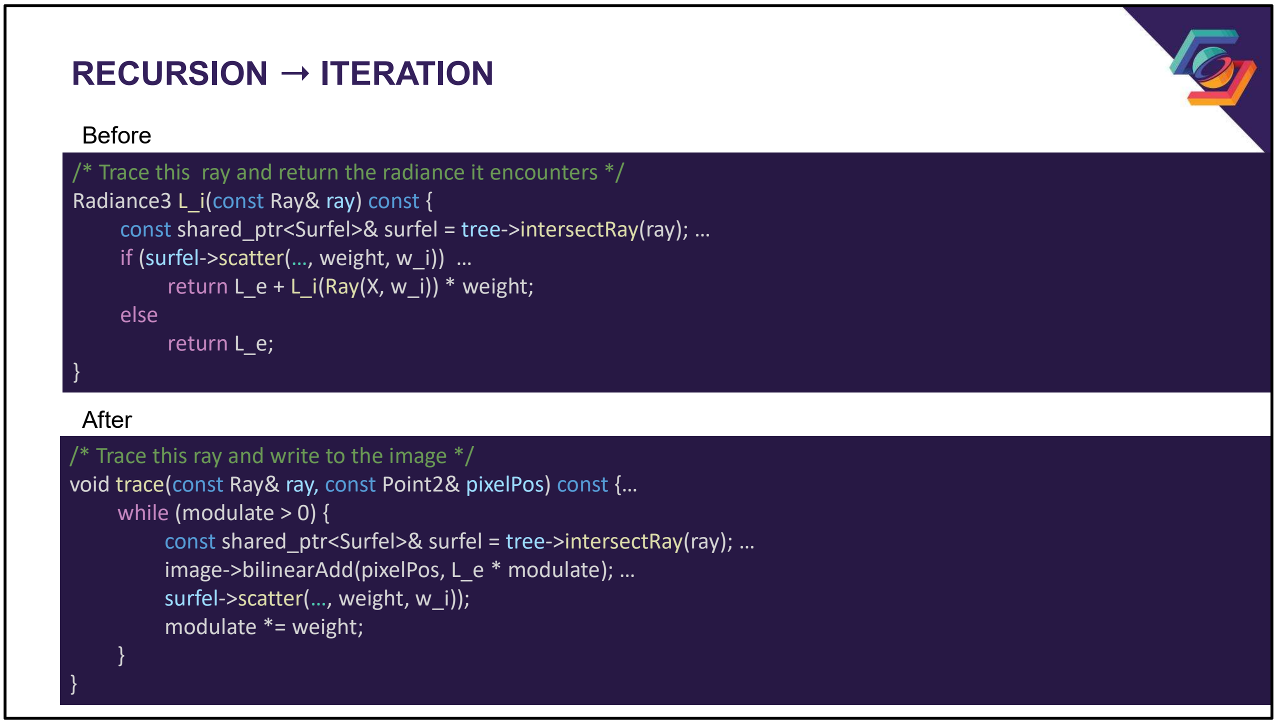
```
/* Trace this  ray and return the radiance it encounters */
Radiance3 L_i(const Ray& ray) const {
        const shared_ptr<Surfel>& surfel = tree->intersectRay(ray); …
        if (surfel->scatter(…, weight, w_i))  …
                return L_e + L_i(Ray(X, w_i)) * weight;
        else
                return L_e;
}
```

After

```
/* Trace this ray and write to the image */
void trace(const Ray& ray, const Point2& pixelPos, const Color3& modulate) const {…
        const shared_ptr<Surfel>& surfel = tree->intersectRay(ray); …
        image->bilinearAdd(pixelPos, L_e * modulate); …
        if (surfel->scatter(…, weight, w_i)) …
                return trace(Ray(X, w_i), pixelPos, weight * modulate);
}
```

Next, we're going to eliminate the recursion.

Step 1: To make this tail recursive, we have to carry through the value to modulate by when writing back to the original pixel.



## RECURSION → ITERATION

Before

```
/* Trace this  ray and return the radiance it encounters */
Radiance3 L_i(const Ray& ray) const {
      const shared_ptr<Surfel>& surfel = tree->intersectRay(ray); …
      if (surfel->scatter(…, weight, w_i))  …
            return L_e + L_i(Ray(X, w_i)) * weight;
      else
            return L_e;
}
```

After

```
/* Trace this ray and write to the image */
void trace(const Ray& ray, const Point2& pixelPos) const {…
      while (modulate > 0) {
            const shared_ptr<Surfel>& surfel = tree->intersectRay(ray); …
            image->bilinearAdd(pixelPos, L_e * modulate); …
            surfel->scatter(…, weight, w_i));
            modulate *= weight;
      }
}
```

Step 2: Convert the tail recursion to iteration

Fire and forget…each step deeper into the transport graph carries the information to write back to the pixel. Never “return”.

We’ve now done two important things:

1. We eliminated the stack and reduced the working state. Essential for a GPU and good for a CPU.
2. Most importantly, we now have each RAY of each PATH treated equally. They don’t know their transport graph depth. That means we can do the next step and process all depths in exactly the same way, in parallel.

## RECURSION → ITERATION

Before

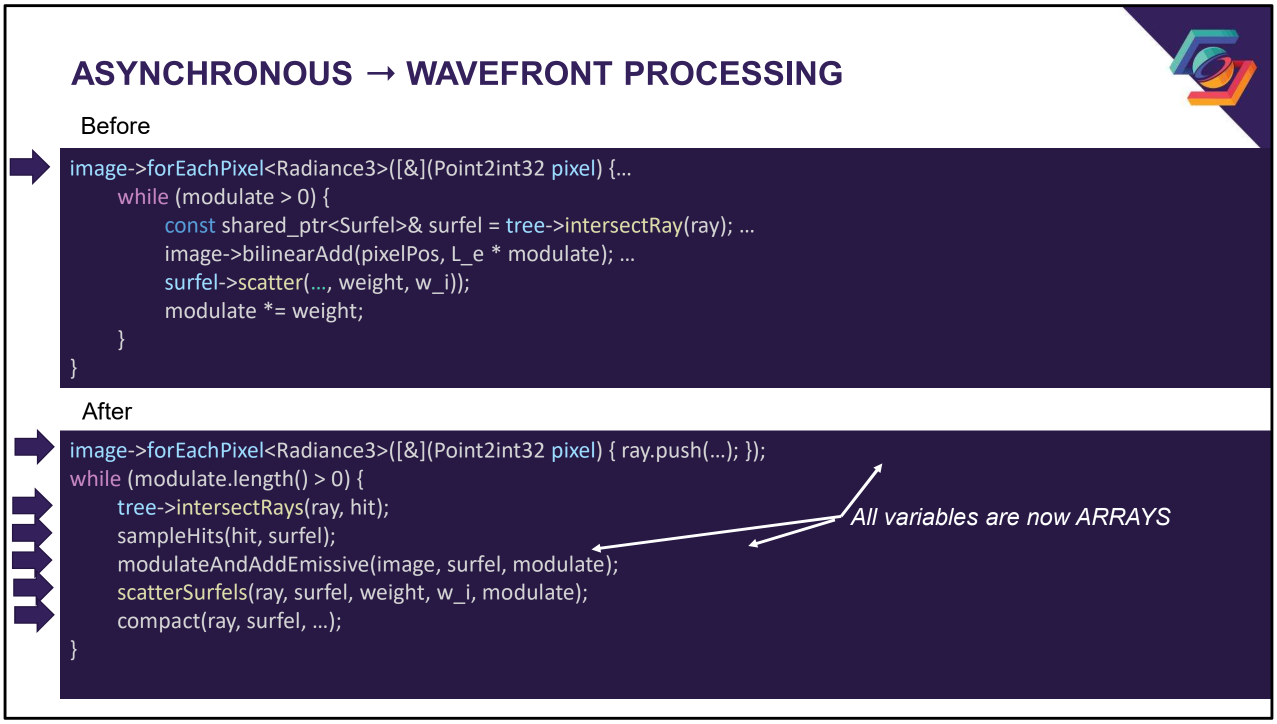
```
/* Trace this  ray and return the radiance it encounters */
Radiance3 L_i(const Ray& ray) const {
        const shared_ptr<Surfel>& surfel = tree->intersectRay(ray); …
        if (surfel->scatter(…, weight, w_i))  …
                return L_e + L_i(Ray(X, w_i)) * weight;
        else
                return L_e;
}
```

After

```
/* Trace all rays and write to the image */
image->forEachPixel<Radiance3>([&](Point2int32 pixel) {…
        while (modulate > 0) {
                const shared_ptr<Surfel>& surfel = tree->intersectRay(ray); …
                image->bilinearAdd(pixelPos, L_e * modulate); …
                surfel->scatter(…, weight, w_i));
                modulate *= weight;
        }
}
```

To make this clearer, I'll get rid of the trace function that we only call once per path and just inline the body into the thread launch.

OK, here's the big one. We're going to invert the structure and make EACH line of code process all of the rays and then hits in parallel. This eliminates the big thread launch and makes a bunch of little launches, one per line:



## ASYNCHRONOUS → WAVEFRONT PROCESSING

Before

```
image->forEachPixel<Radiance3>([&](Point2int32 pixel) {…
      while (modulate > 0) {
            const shared_ptr<Surfel>& surfel = tree->intersectRay(ray); …
            image->bilinearAdd(pixelPos, L_e * modulate); …
            surfel->scatter(…, weight, w_i));
            modulate *= weight;
      }
}
```

After

```
image->forEachPixel<Radiance3>([&](Point2int32 pixel) { ray.push(…); });
while (modulate.length() > 0) {
      tree->intersectRays(ray, hit);
      sampleHits(hit, surfel);
      modulateAndAddEmissive(image, surfel, modulate);
      scatterSurfels(ray, surfel, weight, w_i, modulate);
      compact(ray, surfel, …);
}
```

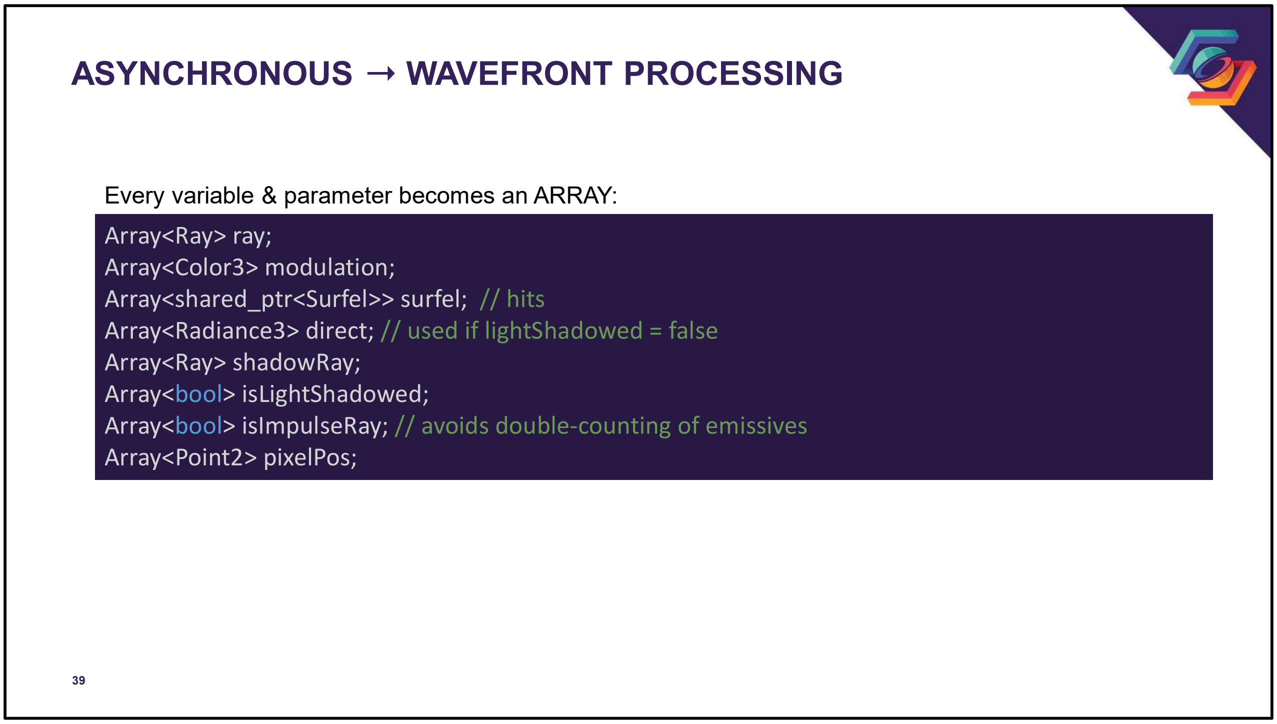
*All variables are now ARRAYS*

Now, you need to know that every variable on the top is a single number, or vector, or color, etc. but every variable on the bottom is an ARRAY of numbers, or vectors, or colors, etc.

Allows us to separate the closest-hit intersector, material evaluation, and scatter code
    into separate kernels
    Fill the machine with coherent programs
    Megakernel is actually doing something similar, it is just a question of how much we schedule and how much the driver/OS/API schedules…I think eventually programmers will not have to do this for performance.

Fit in I$
Better data cache via locality
Better CPU branch prediction
Better GPU SIMD for the trace
Better GPU SIMD for everything
Gives us compaction and binning points

Drawback: LOT more memory ("deferred")

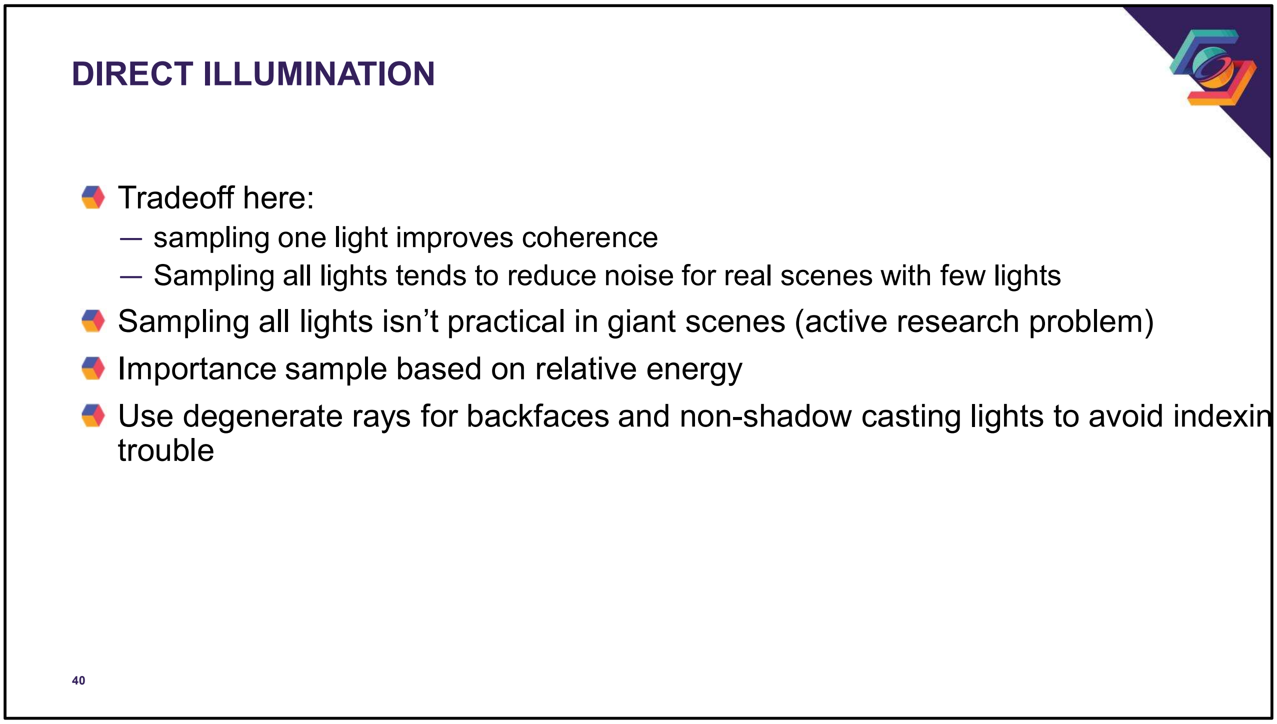


## ASYNCHRONOUS → WAVEFRONT PROCESSING

Every variable & parameter becomes an ARRAY:

```
Array<Ray> ray;
Array<Color3> modulation;
Array<shared_ptr<Surfel>> surfel;  // hits
Array<Radiance3> direct; // used if lightShadowed = false
Array<Ray> shadowRay;
Array<bool> isLightShadowed;
Array<bool> isImpulseRay; // avoids double-counting of emissives
Array<Point2> pixelPos;
```

My “isImpulseRay” is used for similar reasons as Pete’s “Count Only Reflected Light”



## DIRECT ILLUMINATION

- Tradeoff here:
  - — sampling one light improves coherence
  - — Sampling all lights tends to reduce noise for real scenes with few lights
- Sampling all lights isn’t practical in giant scenes (active research problem)
- Importance sample based on relative energy
- Use degenerate rays for backfaces and non-shadow casting lights to avoid indexin trouble

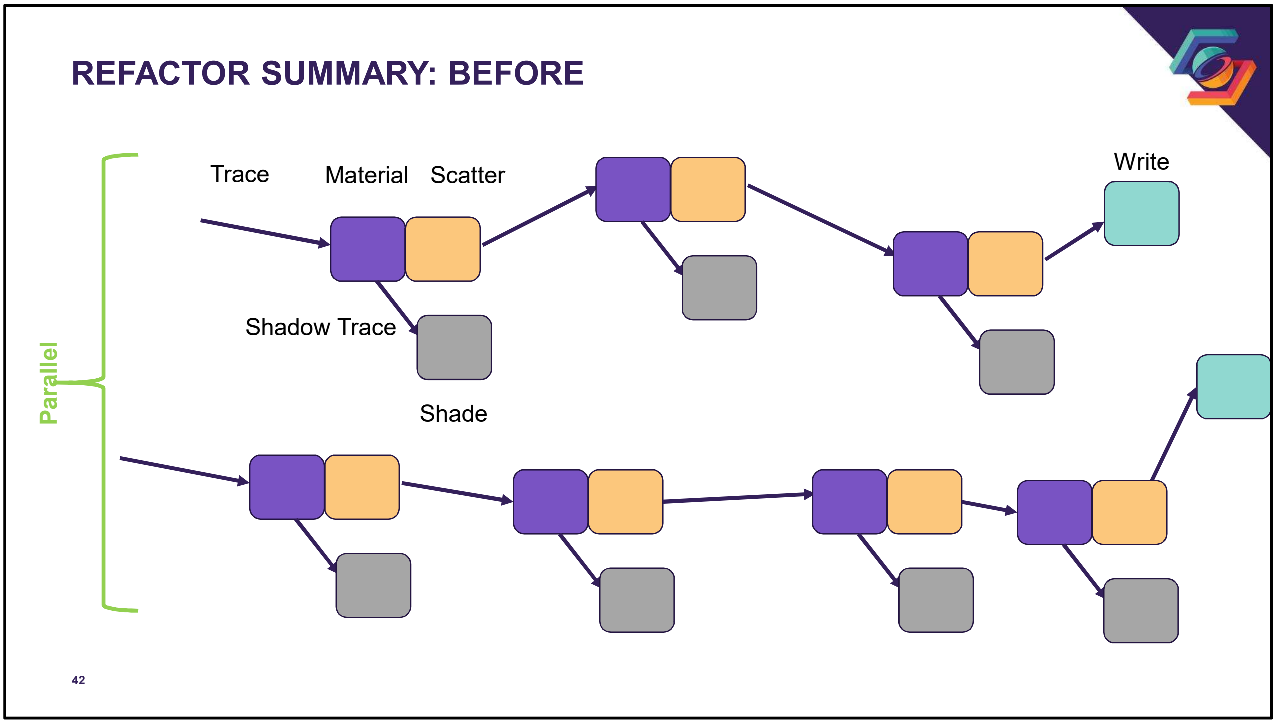
40

Tradeoff here:
	sampling one light improves coherence
	Sampling all lights tends to reduce noise for real scenes with few lights
Sampling all lights isn’t practical in giant scenes (active research problem)
Importance sample based on relative energy

If you’re abstracting your ray generation from the ray launch, as I’m doing here for CPU tracing, then you can reuse your ray buffer
Use degenerate rays for backfaces and non-shadow casting lights to avoid indexing trouble

## MATERIAL

- Bake all materials into flat textures
- Closest hit just stores the intersection; evaluate materials outside of the hit shader



## REFACTOR SUMMARY: BEFORE

Trace

Material

Scatter

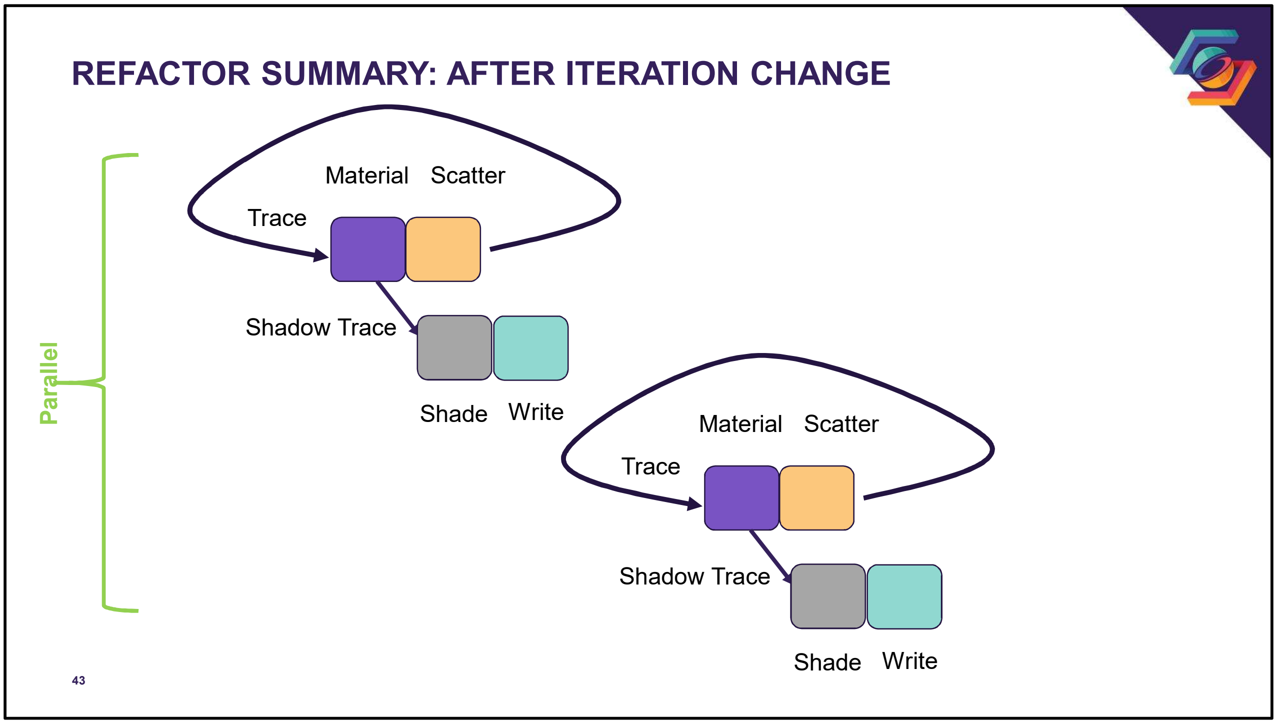
Write

Shadow Trace

Shade

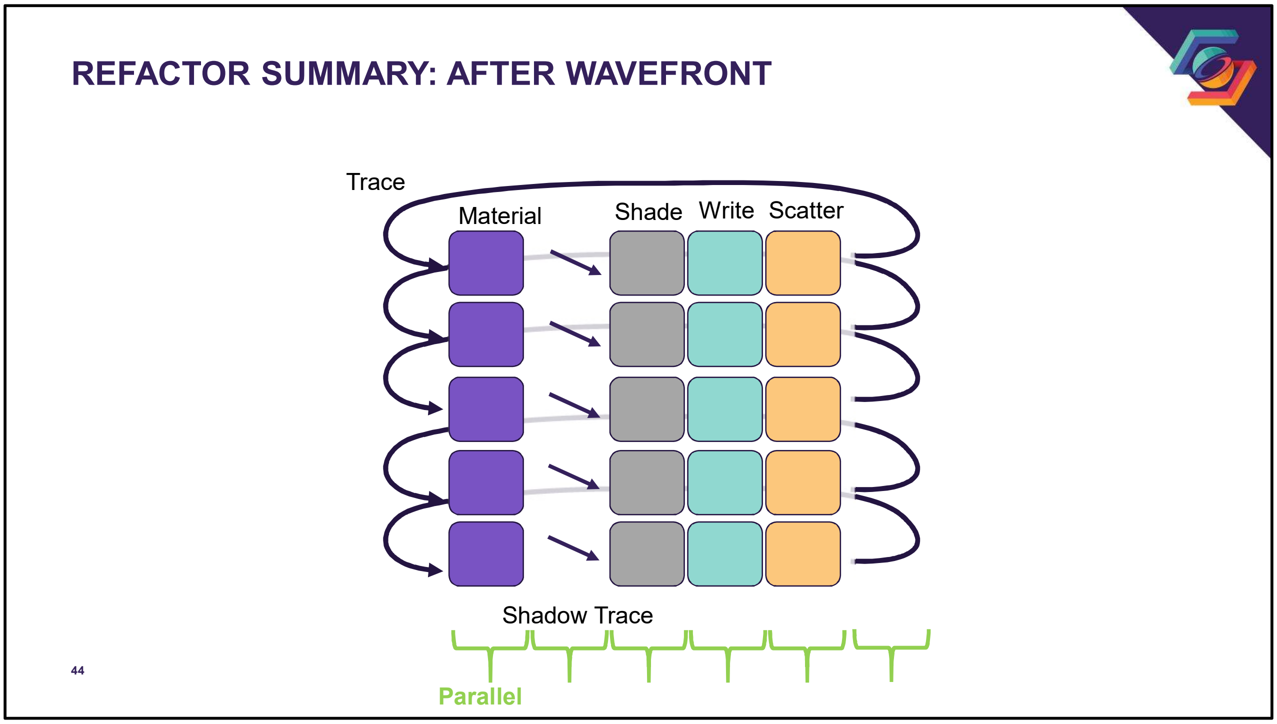
Parallel

We started with work that looks kind of like this



## REFACTOR SUMMARY: AFTER ITERATION CHANGE

Then we made it look like this by turning those long recursive paths into loops.



## REFACTOR SUMMARY: AFTER WAVEFRONT

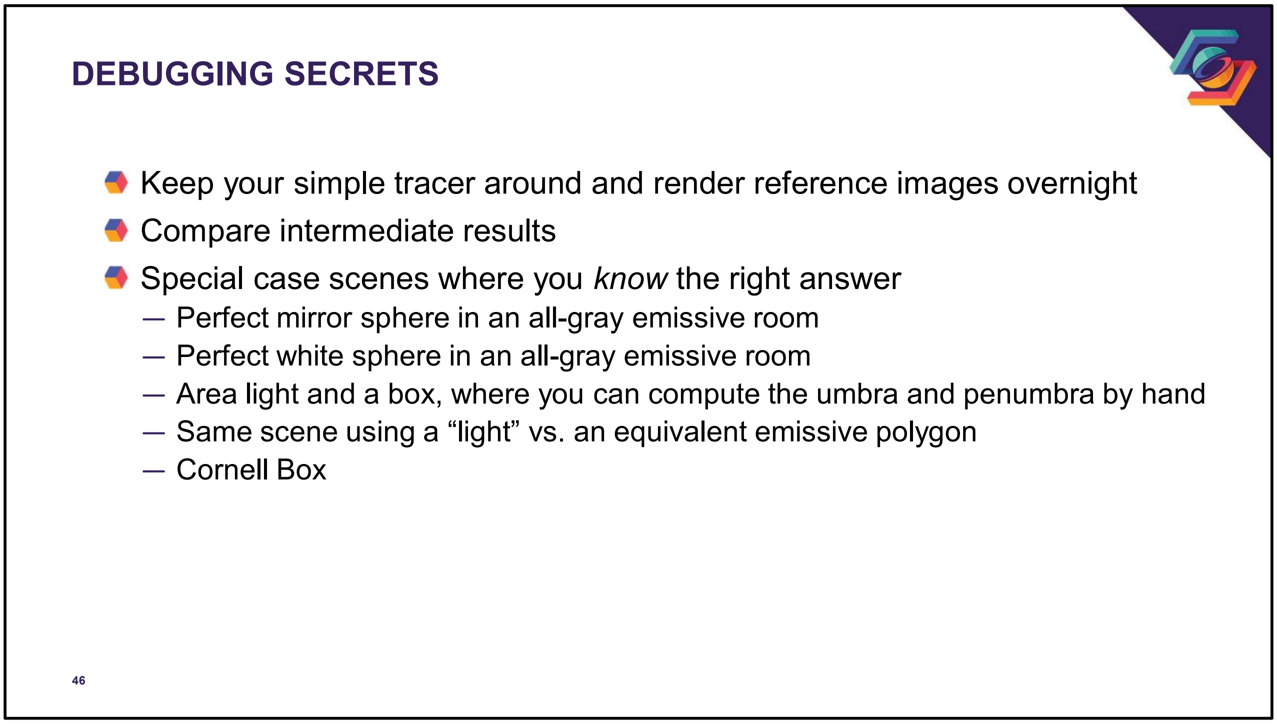
Finally, we inverted the parallelism so that we process EACH stage of the process in parallel, rather than making the overall process parallel. This gave us a really efficient workload.

This was rendered using the path tracer that I just described and gave you a code link for.

On a CPU it took a few minutes at 1920x1080 with 2048 paths per pixel of maximum depth six.
On a GPU with some denoising and a reduced path count it can render in a few milliseconds, for 10 fps interaction at about this quality.

If we drop to one path per pixel, then hits about 5 fps on CPU and 120 Hz on a GPU.

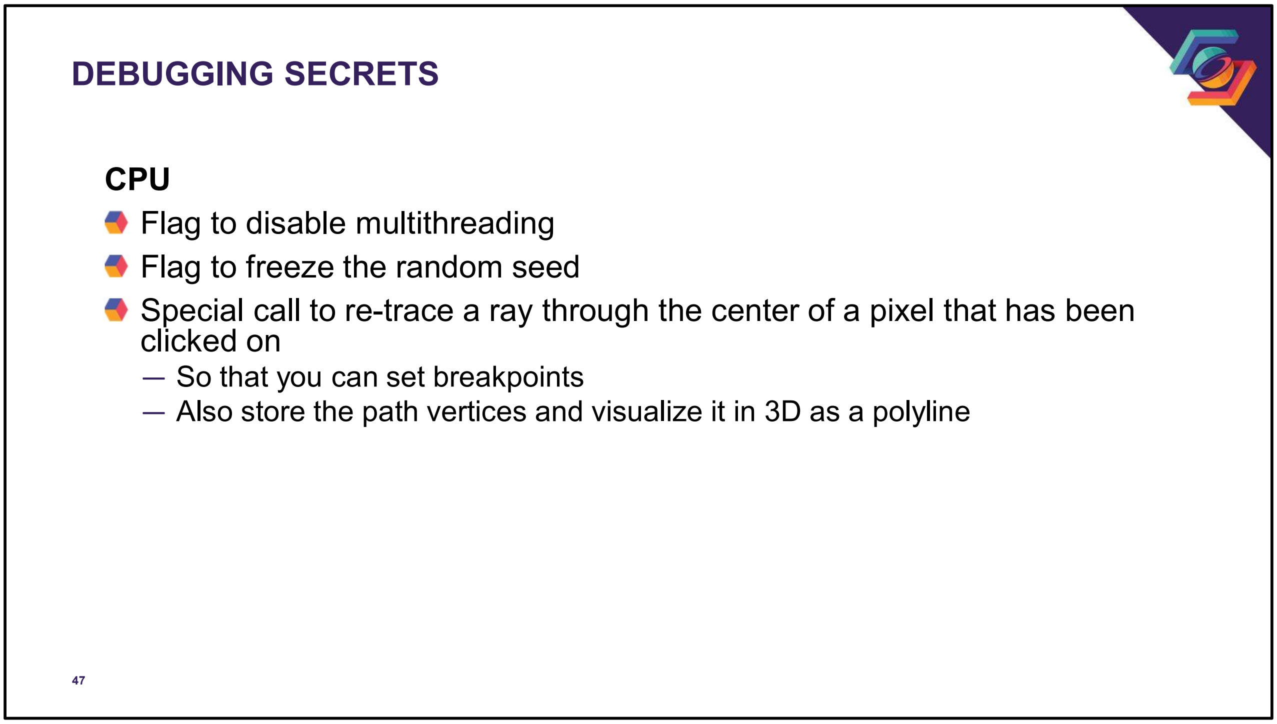
By the way, my code didn’t work right on the first try. Let me give you some debugging tips for parallel path tracers:



## DEBUGGING SECRETS

- Keep your simple tracer around and render reference images overnight
- Compare intermediate results
- Special case scenes where you *know* the right answer
  - Perfect mirror sphere in an all-gray emissive room
  - Perfect white sphere in an all-gray emissive room
  - Area light and a box, where you can compute the umbra and penumbra by hand
  - Same scene using a “light” vs. an equivalent emissive polygon
  - Cornell Box

Let’s now look at the last high level tricks you can use to hit or exceed that performance.

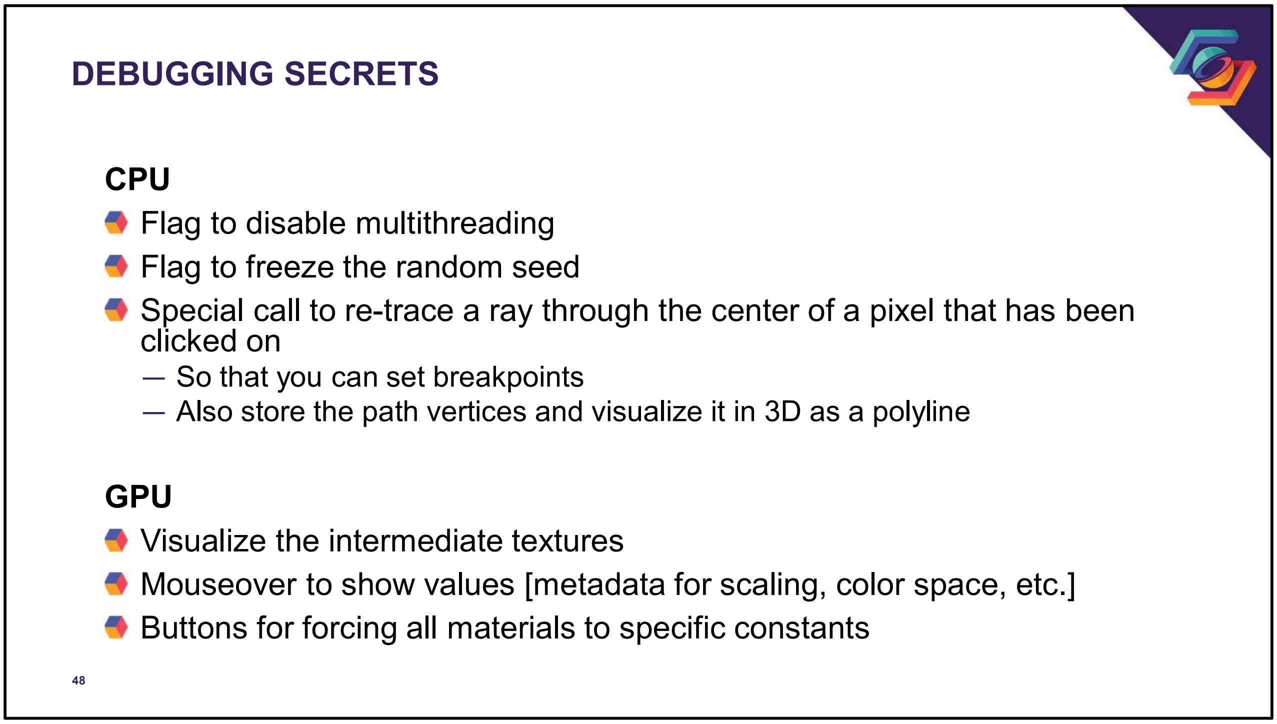


## DEBUGGING SECRETS

**CPU**

- Flag to disable multithreading
- Flag to freeze the random seed
- Special call to re-trace a ray through the center of a pixel that has been clicked on
  - So that you can set breakpoints
  - Also store the path vertices and visualize it in 3D as a polyline

Let’s now look at the last high level tricks you can use to hit or exceed that performance.



## DEBUGGING SECRETS

**CPU**

- Flag to disable multithreading
- Flag to freeze the random seed
- Special call to re-trace a ray through the center of a pixel that has been clicked on
  - So that you can set breakpoints
  - Also store the path vertices and visualize it in 3D as a polyline

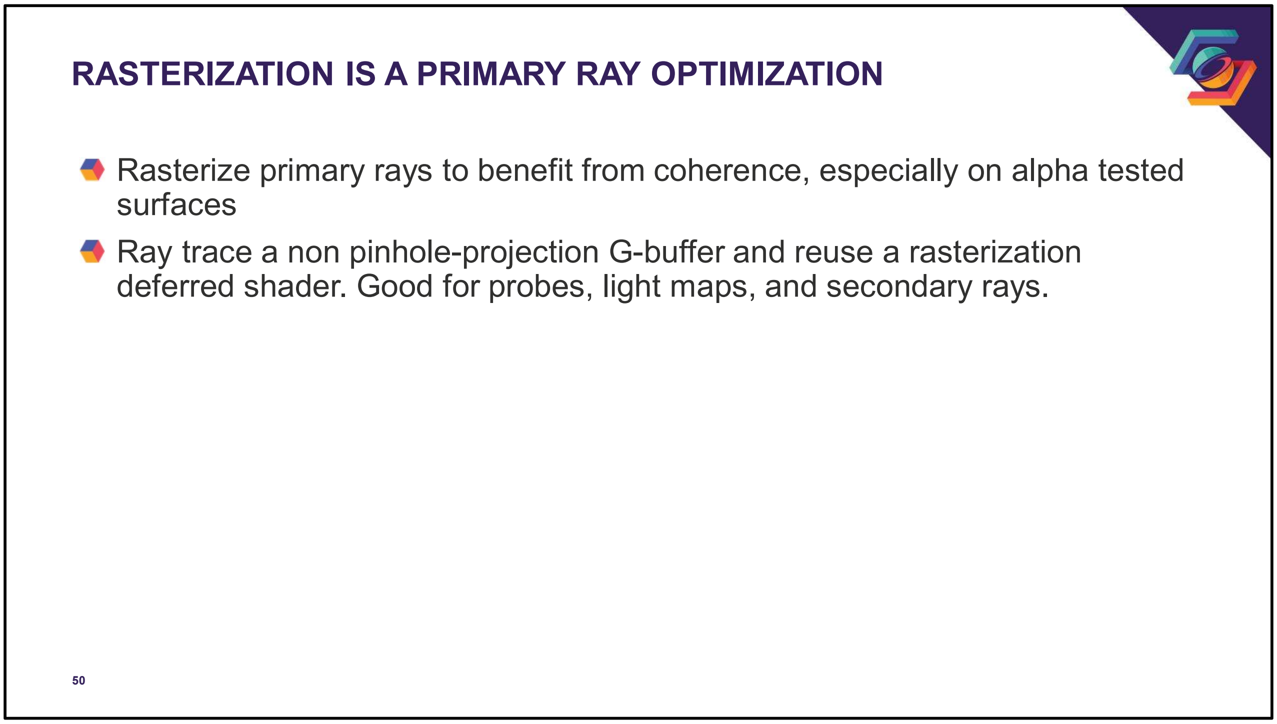
**GPU**

- Visualize the intermediate textures
- Mouseover to show values [metadata for scaling, color space, etc.]
- Buttons for forcing all materials to specific constants

Let’s now look at the last high level tricks you can use to hit or exceed that performance.

# HYBRID RAY-RASTER

5 min



## RASTERIZATION IS A PRIMARY RAY OPTIMIZATION

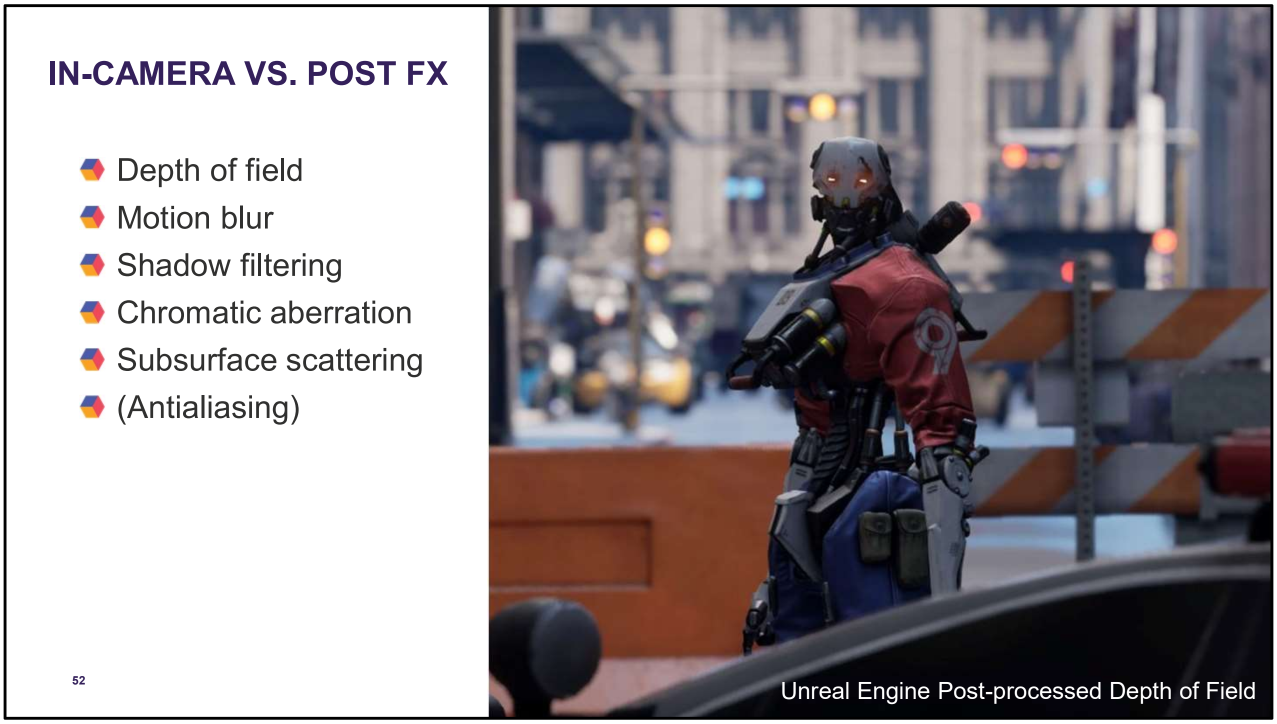
- Rasterize primary rays to benefit from coherence, especially on alpha tested surfaces
- Ray trace a non pinhole-projection G-buffer and reuse a rasterization deferred shader. Good for probes, light maps, and secondary rays.

Note that if you something like glass, everything it is indirect illumination! That means that when the player looks through a window, you’re back to full ray tracing.

## SHADOWS

- Ray-traced shadows are easier to control, but harder to filter efficiently than shadow maps
  - No maximum distance
  - Perfectly sharp (when you want that)
  - Omnidirectional
  - Perfectly directional rays work fine
  - Easier to handle [non-refractive] transparency
- Both have self-shadow issues and need biasing
- Rasterized shadow maps are a useful ray tracing optimization for spot lights

Shadow maps work pretty well for small spot lights, which are often a lot of the local lights in a game. Shadow maps are horrible for omnidirectional lights because you need a spherical projection instead of a planar one, they don’t work well for the sun because of resolution, and they can’t help you at all with area lights. So, ray trace the cases where ray tracing is best, but use shadow maps when you need to speed up spot lights.

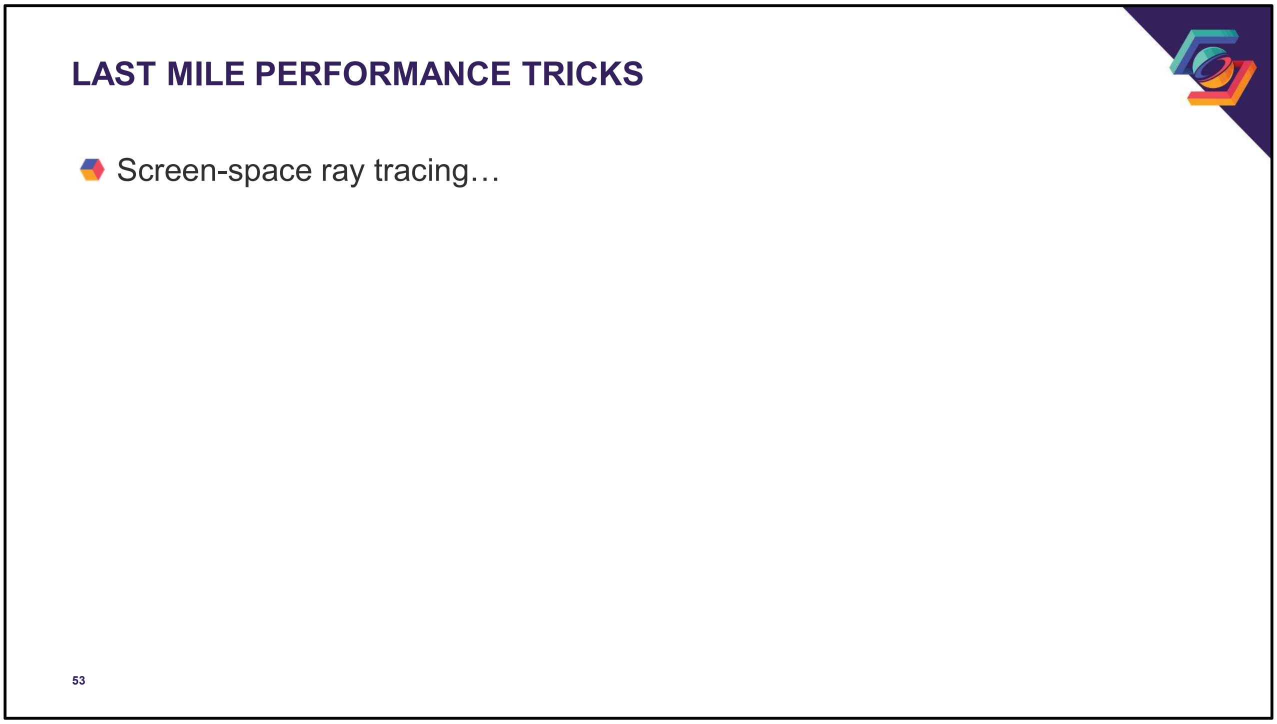


## IN-CAMERA VS. POST FX

- Depth of field
- Motion blur
- Shadow filtering
- Chromatic aberration
- Subsurface scattering
- (Antialiasing)

Unreal Engine Post-processed Depth of Field

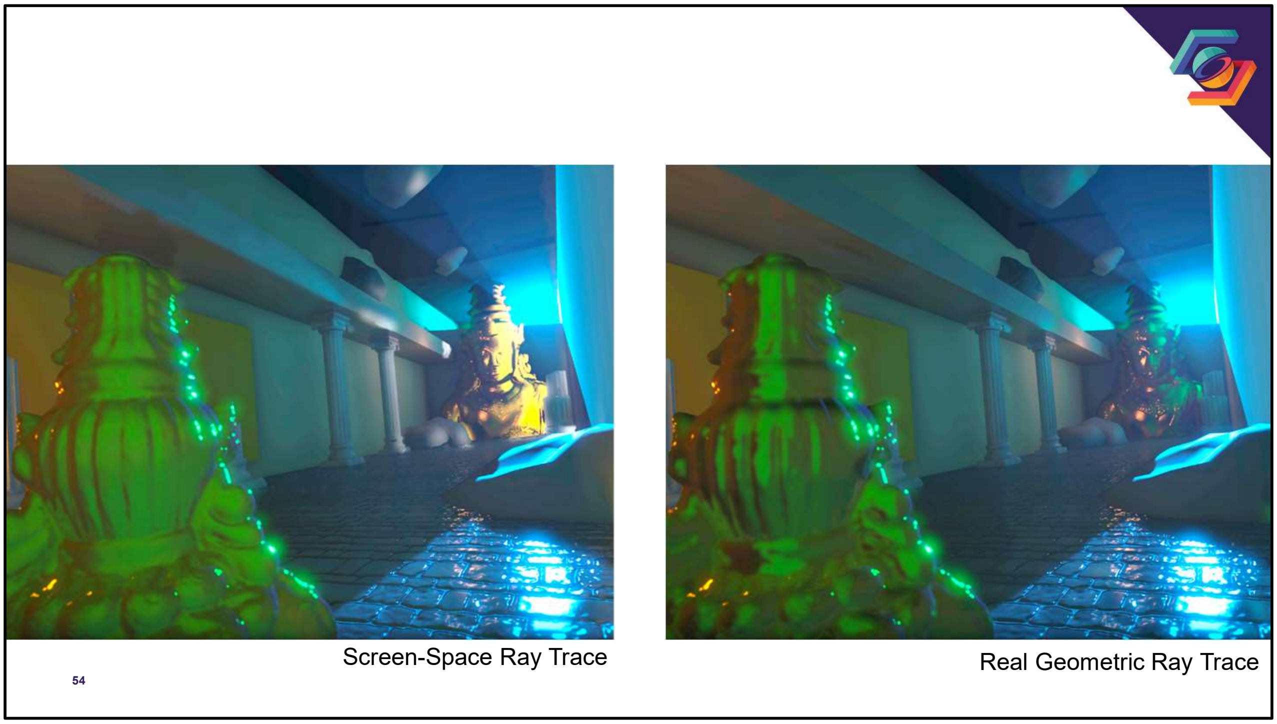
There are a number of phenomena which can be modeled elegantly with ray tracing



## LAST MILE PERFORMANCE TRICKS

- Screen-space ray tracing…

SSRT cheaper than true rays especially with alpha testing, and often indistinguishable for rough surfaces. But where it fails, it fails hard. Battlefield V mixes SSRT for instanced rubble with true rays for major objects

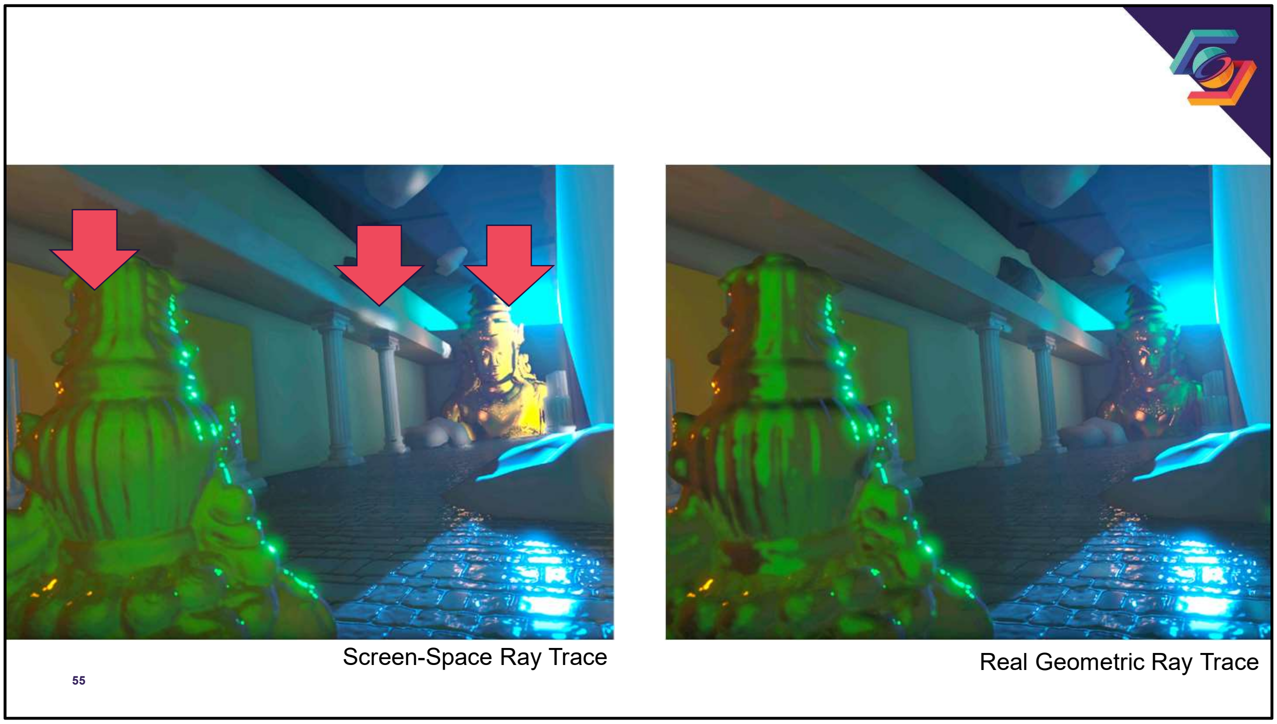


Screen-Space Ray Trace

Real Geometric Ray Trace

Here is a scene with a mixture of really shiny and diffuse surfaces. I screen-space ray traced it on the left and geometric traced it on the right.

Overall, the screen-space ray trace is not bad. So you *can* use it as an accelerator in some cases!
But when it is wrong, it can be really wrong…



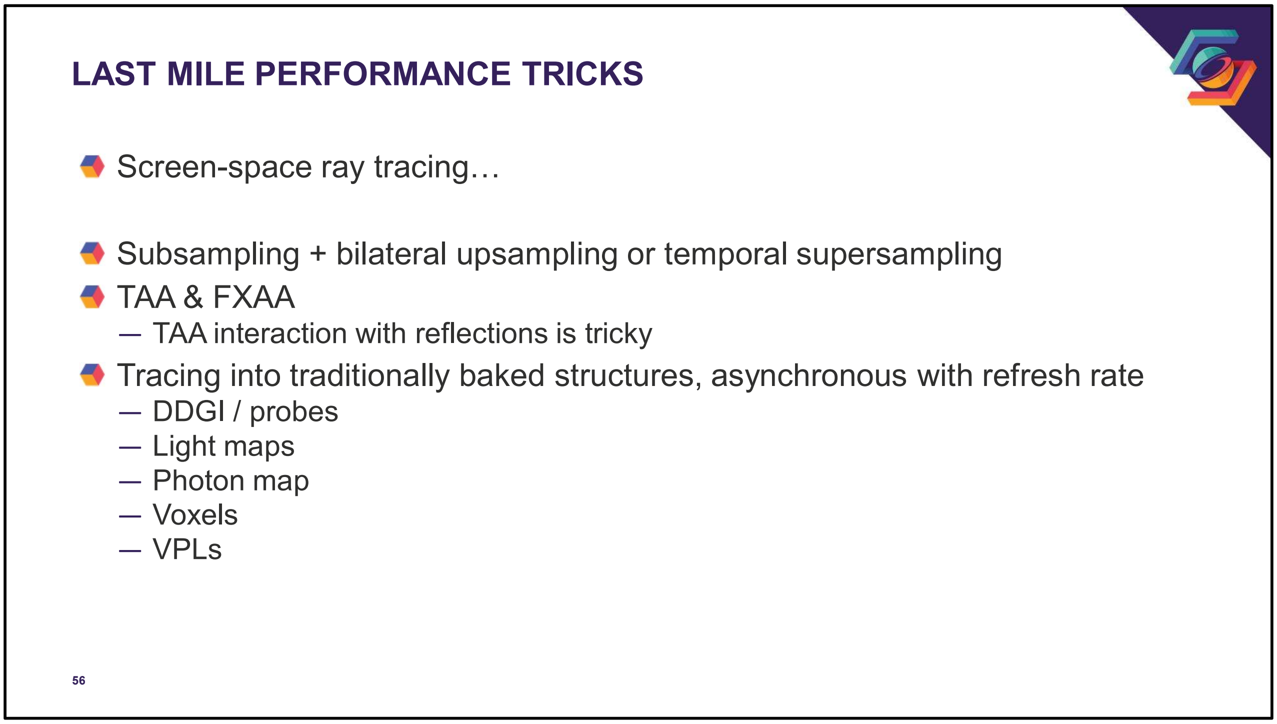
Screen-Space Ray Trace

Real Geometric Ray Trace

These areas just look very different between the images.

The screen-space ray tracer just gets the wrong answer in those areas. And it gets a DIFFERENT wrong answer as the camera moves. So, this isn’t exactly my first choice approximation.

What I do like…



## LAST MILE PERFORMANCE TRICKS

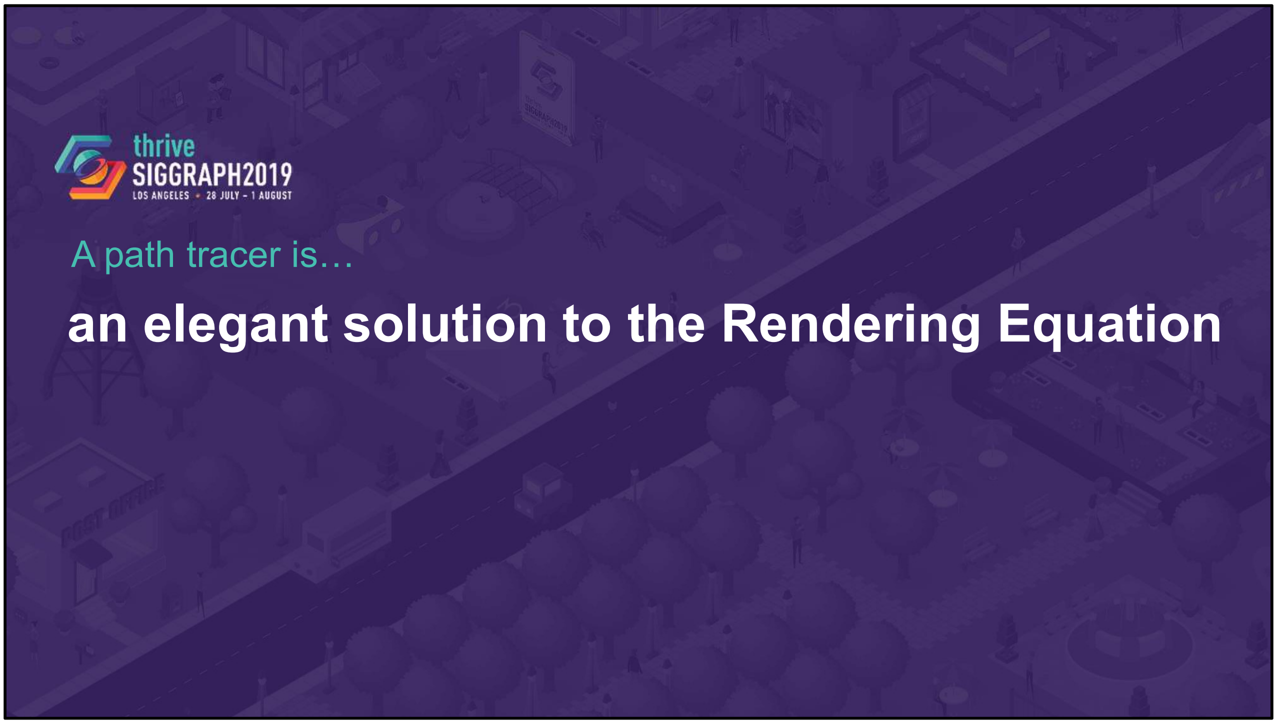
- Screen-space ray tracing…

- Subsampling + bilateral upsampling or temporal supersampling
- TAA & FXAA
  - — TAA interaction with reflections is tricky
- Tracing into traditionally baked structures, asynchronous with refresh rate
  - — DDGI / probes
  - — Light maps
  - — Photon map
  - — Voxels
  - — VPLs

is reducing the number of rays per frame by amortizing the cost in space or time.

You can do this using some intermediate data structure, which might be the screen, or a light map, or a probe or whatever.

Let me wrap my section with one slide of philosophy, and then I'll conclude the course:



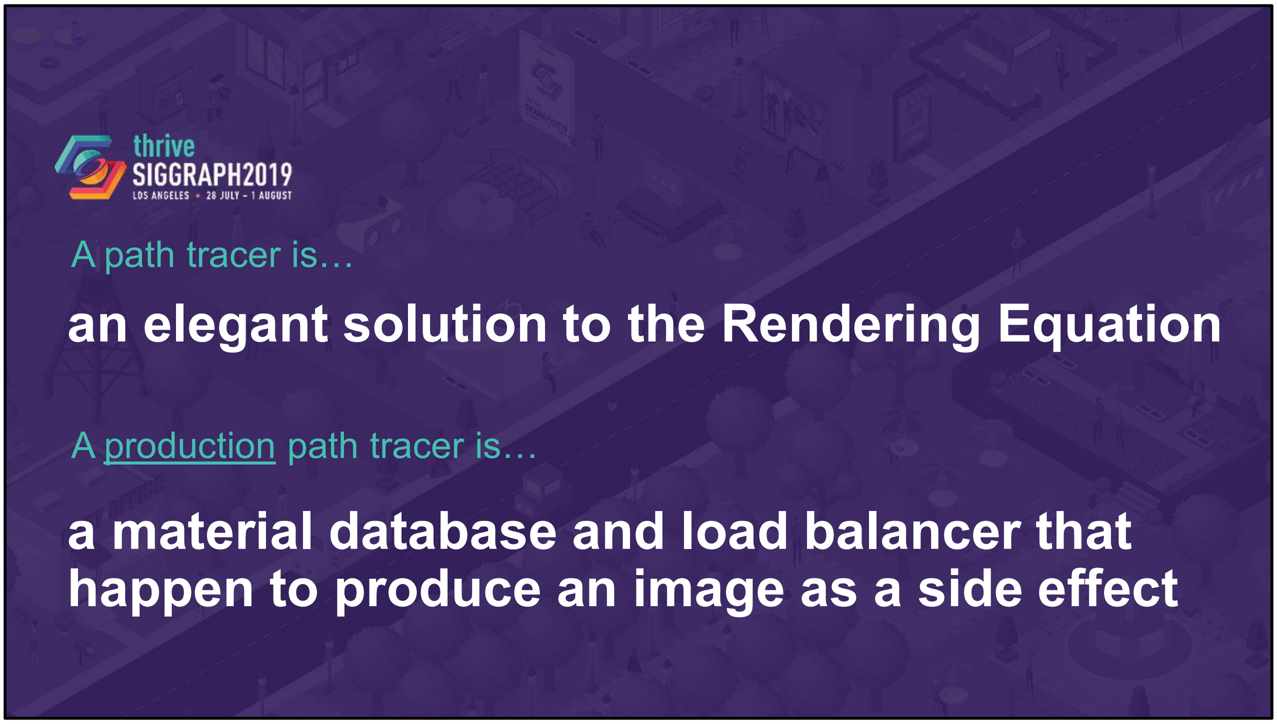
thrive
SIGGRAPH2019
LOS ANGELES • 28 JULY – 1 AUGUST

A path tracer is…

## an elegant solution to the Rendering Equation

The path tracer is pretty unrecognizable after all of this optimization. Peter Shirley showed you the beauty of Path Tracing. I destroyed it.

To paraphrase a quote from Larry Gritz,

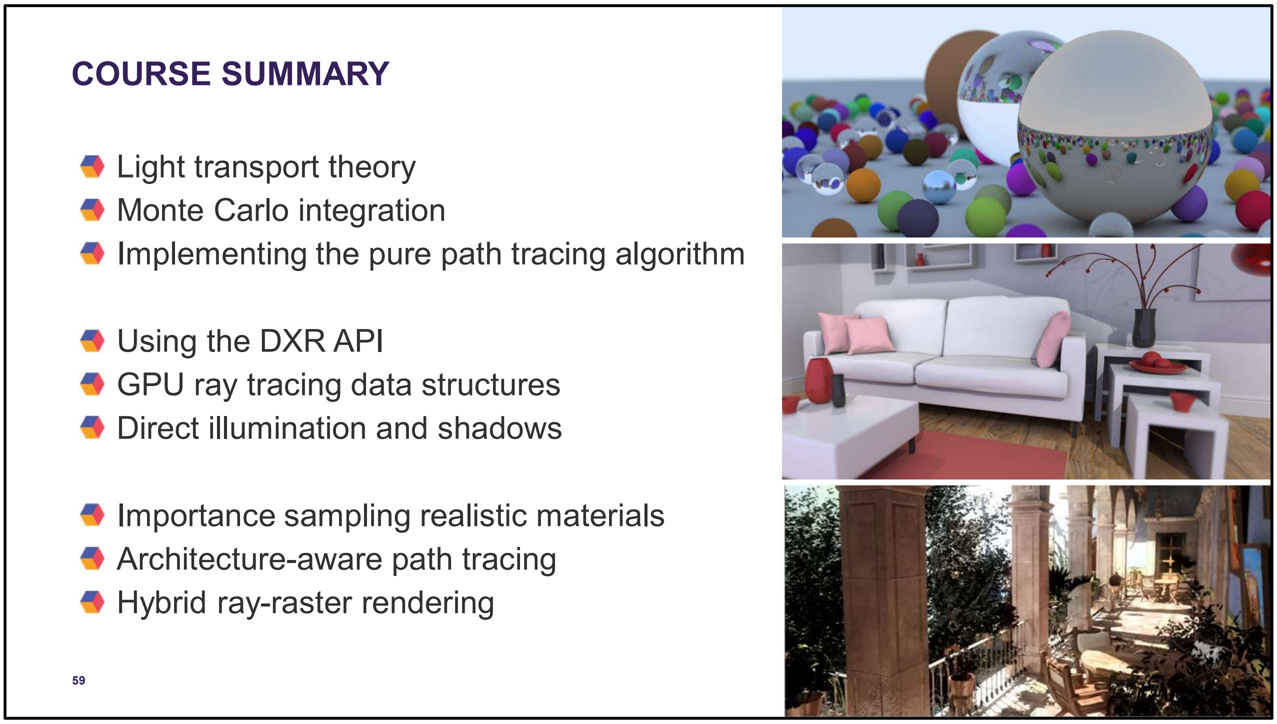


A path tracer is…

**an elegant solution to the Rendering Equation**

A <u>production</u> path tracer is…

**a material database and load balancer that happen to produce an image as a side effect**



## COURSE SUMMARY

- Light transport theory
- Monte Carlo integration
- Implementing the pure path tracing algorithm

- Using the DXR API
- GPU ray tracing data structures
- Direct illumination and shadows

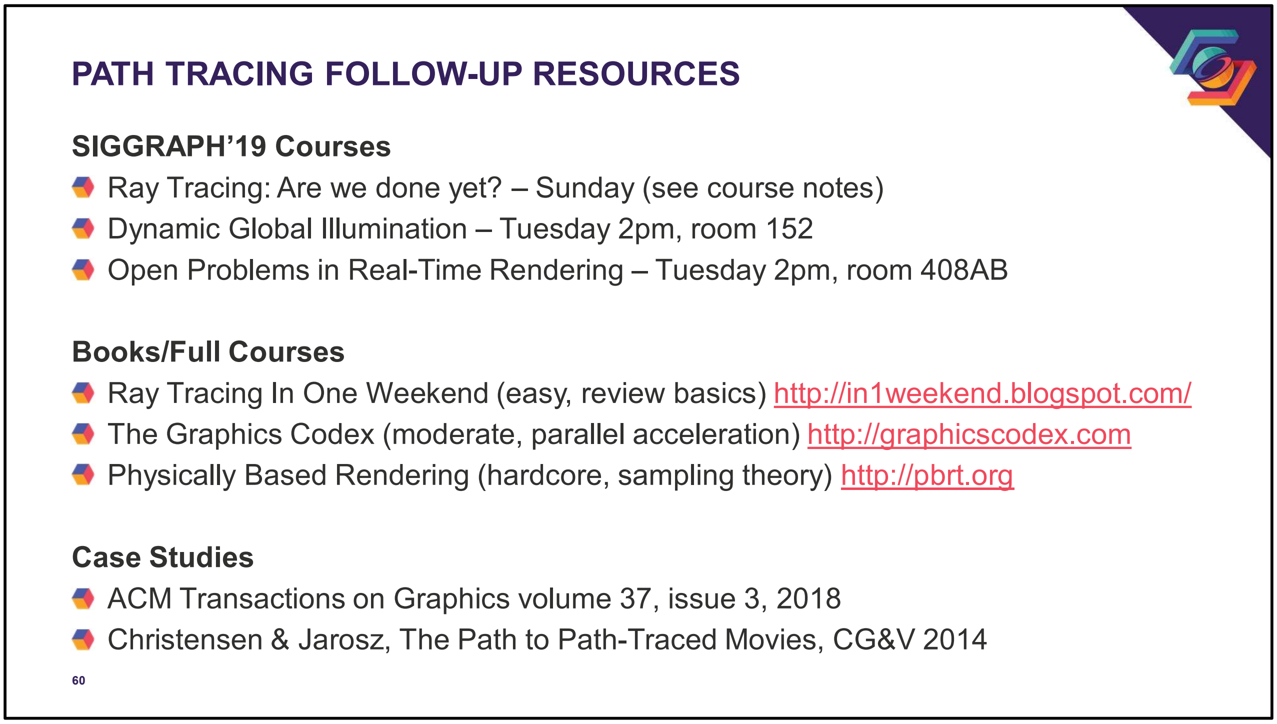
- Importance sampling realistic materials
- Architecture-aware path tracing
- Hybrid ray-raster rendering

In this course we've covered all of these topics and taken you on a whirlwind tour from a blank C++ file all of the way to real-time rendering.

The course notes and slides will be online after SIGGRAPH.
Extended versions of this material are also available from the three speakers on our websites via online books and tutorials that have been linked from the slides.

I'll ask my fellow instructors to join me back up on the stage so that we can take questions on all three sections.

On behalf of all of us, thank you very much for attending the course today.



## PATH TRACING FOLLOW-UP RESOURCES

**SIGGRAPH'19 Courses**

- Ray Tracing: Are we done yet? – Sunday (see course notes)
- Dynamic Global Illumination – Tuesday 2pm, room 152
- Open Problems in Real-Time Rendering – Tuesday 2pm, room 408AB

**Books/Full Courses**

- Ray Tracing In One Weekend (easy, review basics) http://in1weekend.blogspot.com/
- The Graphics Codex (moderate, parallel acceleration) http://graphicscodex.com
- Physically Based Rendering (hardcore, sampling theory) http://pbrt.org

**Case Studies**

- ACM Transactions on Graphics volume 37, issue 3, 2018
- Christensen & Jarosz, The Path to Path-Traced Movies, CG&V 2014

My Graphics Codex book goes deep on the parallel acceleration aspects

And Pharr and Humphreys' book, which is the masterwork on path tracing sampling

Both come with full source code and are inexpensive or free if you use the online versions.

# QUESTIONS