A STORY

A STORY

A STORY

A STORY

APPLY TO C++

THIS TALK IS NOT

- Expert advice on:
  - Path tracing
  - C++ code styles
- Style “X” is best!

BUT IT IS

- My interpretation
- Real code

BUT IT IS

- My interpretation
- Real code
- Interesting & fun (hopefully)

STYLES

STYLES

- Object Oriented

STYLES

- Object Oriented
- Functional Programming

STYLES

- Object Oriented
- Functional Programming
- Data-Oriented Design

PATH TRACING

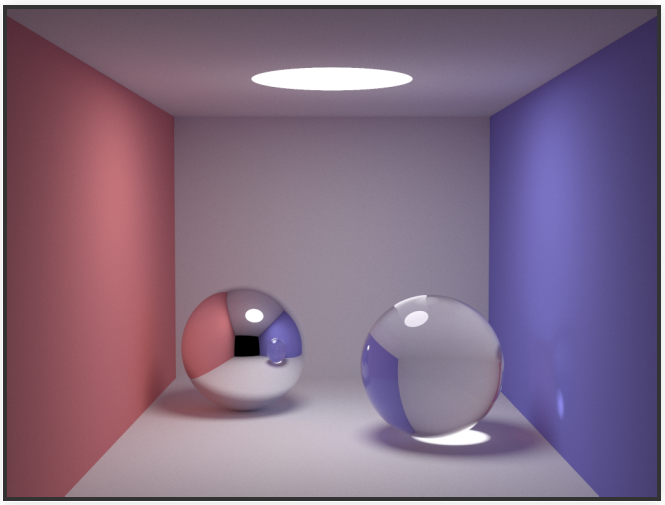


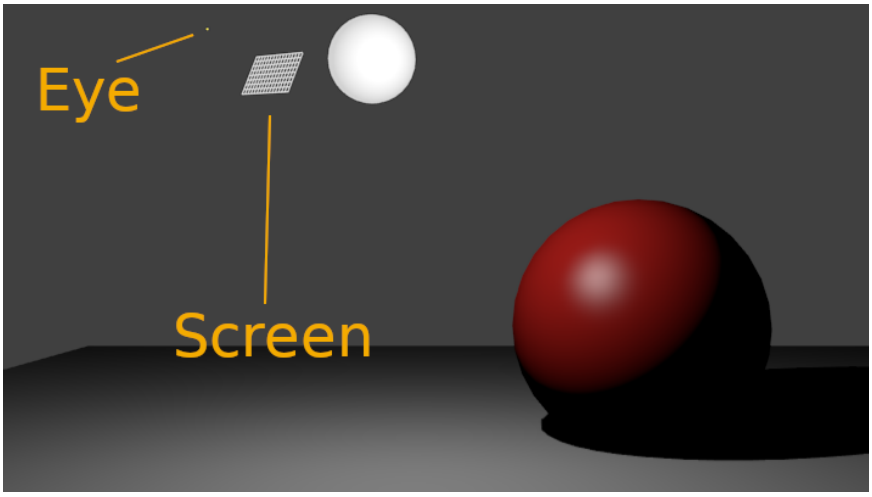
Image credit: Kevin Beason

PATH TRACING

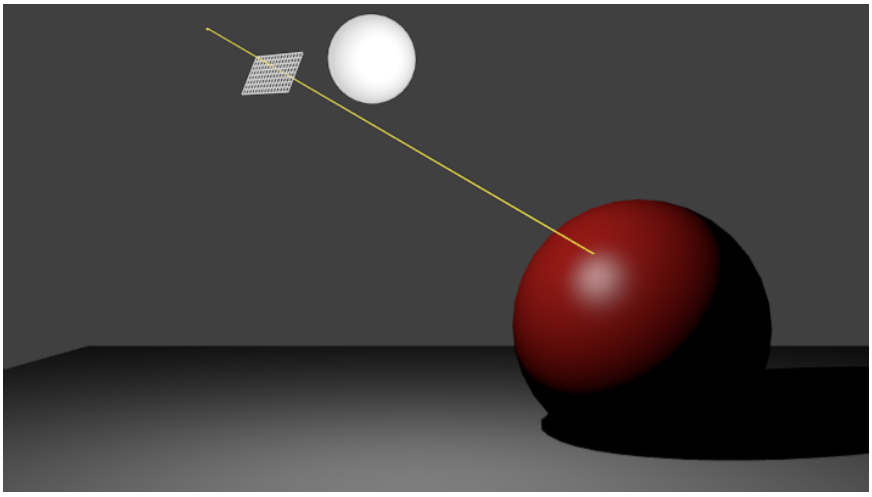
```
36.   Sphere(1e5, Vec(50,-1e5+81.6,81.6),Vec(),Vec(.75,.75,.75),DIFF),//Top
37.   Sphere(16.5,Vec(27,16.5,47),       Vec(),Vec(1,1,1)*.999, SPEC),//Mirr
38.   Sphere(16.5,Vec(73,16.5,78),       Vec(),Vec(1,1,1)*.999, REFR),//Glas
39.   Sphere(600, Vec(50,681.6-.27,81.6),Vec(12,12,12),  Vec(), DIFF) //Lite
40. };
41. inline double clamp(double x){ return x<0 ? 0 : x>1 ? 1 : x; }
42. inline int toInt(double x){ return int(pow(clamp(x),1/2.2)*255+.5); }
43. inline bool intersect(const Ray &r, double &t, int &id){
44.   double n=sizeof(spheres)/sizeof(Sphere), d, inf=t=1e20;
45.   for(int i=int(n);i--;) if((d=spheres[i].intersect(r))&&d<t){t=d;id=i;}
46.   return t<inf;
47. }
48. Vec radiance(const Ray &r, int depth, unsigned short *Xi){
49.   double t;                               // distance to intersection
50.   int id=0;                               // id of intersected object
51.   if (!intersect(r, t, id)) return Vec(); // if miss, return black
52.   const Sphere &obj = spheres[id];        // the hit object
53.   Vec x=r.o+r.d*t, n=(x-obj.p).norm(), nl=n.dot(r.d)<0?n:n*-1, f=obj.c;
54.   double p = f.x>f.y && f.x>f.z ? f.x : f.y>f.z ? f.y : f.z; // max refl
55.   if (++depth>5) if (erand48(Xi)<p) f=f*(1/p); else return obj.e; //R.R.
56.   if (obj.refl == DIFF){                  // Ideal DIFFUSE reflection
57.     double r1=2*M_PI*erand48(Xi), r2=erand48(Xi), r2s=sqrt(r2);
58.     Vec w=nl, u=((fabs(w.x)>.1?Vec(0,1):Vec(1))%w).norm(), v=w%u;
59.     Vec d = (u*cos(r1)*r2s + v*sin(r1)*r2s + w*sqrt(1-r2)).norm();
60.     return obj.e + f.mult(radiance(Ray(x,d),depth,Xi));
61.   } else if (obj.refl == SPEC)            // Ideal SPECULAR reflection
62.     return obj.e + f.mult(radiance(Ray(x,r.d-n*2*n.dot(r.d)),depth,Xi));
63.   Ray reflRay(x, r.d-n*2*n.dot(r.d));     // Ideal dielectric REFRACTION
64.   bool into = n.dot(nl)>0;                // Ray from outside going in?
65.   double nc=1, nt=1.5, nnt=into?nc/nt:nt/nc, ddn=r.d.dot(nl), cos2t;
66.   if ((cos2t=1-nnt*nnt*(1-ddn*ddn))<0)    // Total internal reflection
67.     return obj.e + f.mult(radiance(reflRay,depth,Xi));
```

Image credit: Kevin Beason

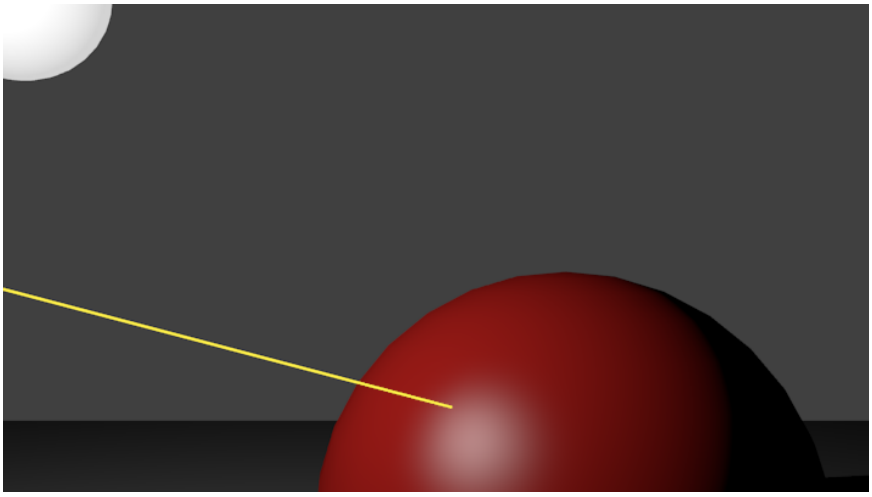
PATH TRACING



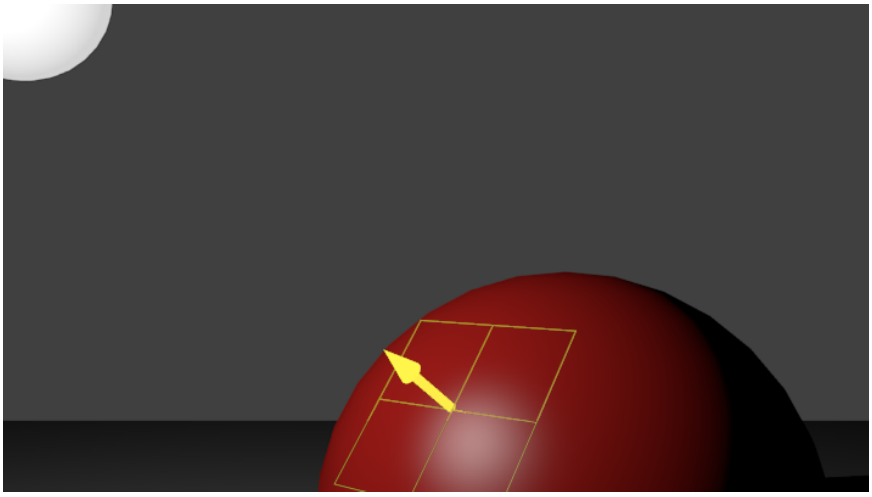
PATH TRACING



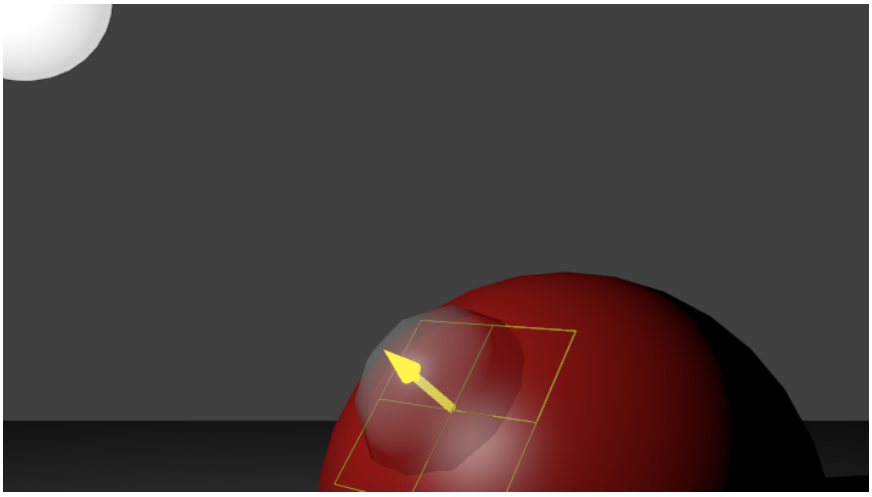
PATH TRACING



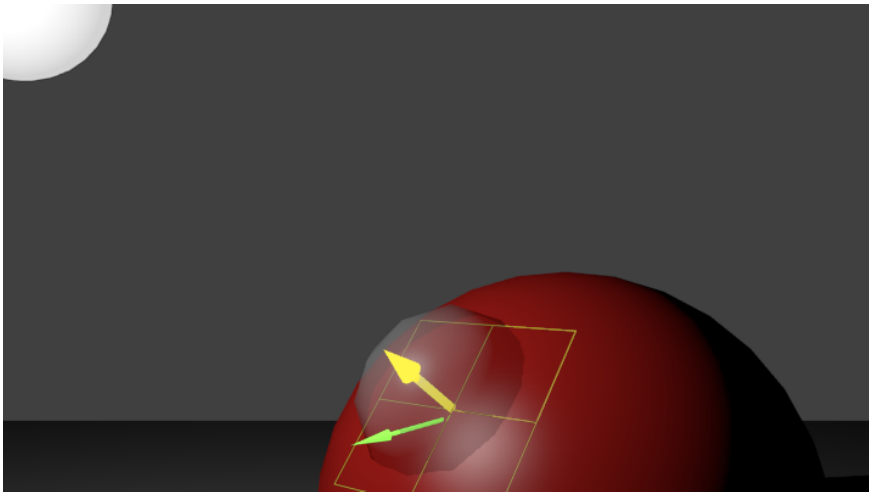
PATH TRACING



PATH TRACING



PATH TRACING



PATH TRACING

For each pixel:

PATH TRACING

For each pixel:

1. Fire a zillion rays into the scene

PATH TRACING

For each pixel:

1. Fire a zillion rays into the scene
2. Bounce randomly; accumulate intensity & colour

PATH TRACING

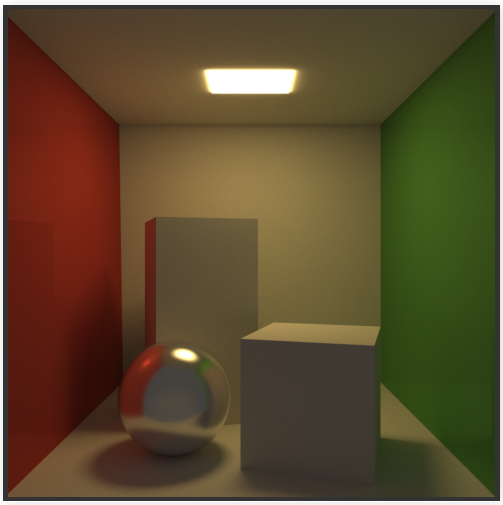
For each pixel:

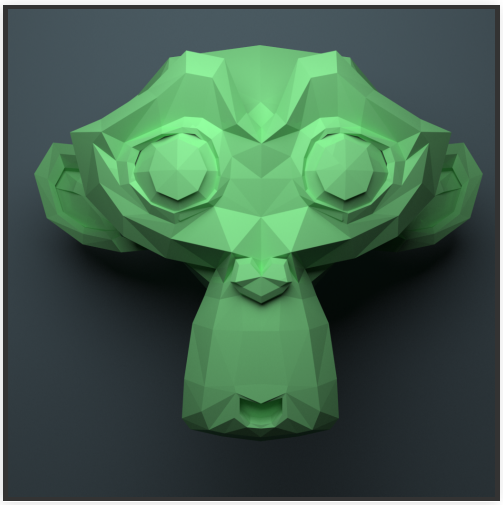
1. Fire a zillion rays into the scene
2. Bounce randomly; accumulate intensity & colour
3. Average them out

MY PATH TRACER

- Simple materials
- Spheres & triangles only
- Scenes in code
- Written 3 times!

MY PATH TRACER





Model credit: "Suzanne" from Blender
With thanks to Michael Fogleman

GENERAL APPROACH

- Shared basic math library
  - `Vec3` / `Norm3`
  - `Ray`
  - `Hit`
- Some simple “util” classes
  - `MaterialSpec` struct
  - `Camera`

OBJECT ORIENTED

MAJOR COMPONENTS

- Scene
- Renderer
- Radiance calculation
- Materials
- Intersection

```
class Primitive {
public:
  virtual ~Primitive() = default;

  struct IntersectionRecord {
    Hit hit;
    const Material *material = nullptr;
  };

  [[nodiscard]] virtual bool intersect(
    const Ray &ray,
    IntersectionRecord &intersection) const = 0;
};
```

```
class Primitive {
public:
  virtual ~Primitive() = default;

  struct IntersectionRecord {
    Hit hit;
    const Material *material = nullptr;
  };

  [[nodiscard]] virtual bool intersect(
    const Ray &ray,
    IntersectionRecord &intersection) const = 0;
};
```

```
class Primitive {
public:
  virtual ~Primitive() = default;

  struct IntersectionRecord {
    Hit hit;
    const Material *material = nullptr;
  };

  [[nodiscard]] virtual bool intersect(
    const Ray &ray,
    IntersectionRecord &intersection) const = 0;
};
```

```
class Primitive {
public:
  virtual ~Primitive() = default;

  struct IntersectionRecord {
    Hit hit;
    const Material *material = nullptr;
  };

  [[nodiscard]] virtual bool intersect(
    const Ray &ray,
    IntersectionRecord &intersection) const = 0;
};
```

RENDER

```
for (int y = 0; y < height; ++y) {
  for (int x = 0; x < width; ++x) {
    Vec3 colour;
    for (int sample = 0; sample < numSamples; ++sample) {
      auto ray = camera_.randomRay(x, y, rng);
      colour += radiance(rng, ray, 0);
    }
    output.plot(x, y, colour / numSamples);
  }
}
```

RENDER

```
for (int y = 0; y < height; ++y) {
  for (int x = 0; x < width; ++x) {
    Vec3 colour;
    for (int sample = 0; sample < numSamples; ++sample) {
      auto ray = camera_.randomRay(x, y, rng);
      colour += radiance(rng, ray, 0);
    }
    output.plot(x, y, colour / numSamples);
  }
}
```

RENDER

```
for (int y = 0; y < height; ++y) {
  for (int x = 0; x < width; ++x) {
    Vec3 colour;
    for (int sample = 0; sample < numSamples; ++sample) {
      auto ray = camera_.randomRay(x, y, rng);
      colour += radiance(rng, ray, 0);
    }
    output.plot(x, y, colour / numSamples);
  }
}
```

RENDER

```
for (int y = 0; y < height; ++y) {
  for (int x = 0; x < width; ++x) {
    Vec3 colour;
    for (int sample = 0; sample < numSamples; ++sample) {
      auto ray = camera_.randomRay(x, y, rng);
      colour += radiance(rng, ray, 0);
    }
    output.plot(x, y, colour / numSamples);
  }
}
```

RENDER

```
for (int y = 0; y < height; ++y) {
  for (int x = 0; x < width; ++x) {
    Vec3 colour;
    for (int sample = 0; sample < numSamples; ++sample) {
      auto ray = camera_.randomRay(x, y, rng);
      colour += radiance(rng, ray, 0);
    }
    output.plot(x, y, colour / numSamples);
  }
}
```

RENDER

```
for (int y = 0; y < height; ++y) {
  for (int x = 0; x < width; ++x) {
    Vec3 colour;
    for (int sample = 0; sample < numSamples; ++sample) {
      auto ray = camera_.randomRay(x, y, rng);
      colour += radiance(rng, ray, 0);
    }
    output.plot(x, y, colour / numSamples);
  }
}
```

RADIANCE

```
Vec3 Renderer::radiance(
    std::mt19937 &rng, const Ray &ray, int depth) const {

  if (depth >= MaxDepth)
    return Vec3();

  Primitive::IntersectionRecord intersectionRecord;
  if (!scene_.intersect(ray, intersectionRecord))
    return Vec3();

  int numUSamples = depth == 0 ? renderParams_.firstBounceUSamples : 1;
  int numVSamples = depth == 0 ? renderParams_.firstBounceVSamples : 1;
```

RADIANCE

```
Vec3 Renderer::radiance(
    std::mt19937 &rng, const Ray &ray, int depth) const {

  if (depth >= MaxDepth)
    return Vec3();

  Primitive::IntersectionRecord intersectionRecord;
  if (!scene_.intersect(ray, intersectionRecord))
    return Vec3();

  int numUSamples = depth == 0 ? renderParams_.firstBounceUSamples : 1;
  int numVSamples = depth == 0 ? renderParams_.firstBounceVSamples : 1;
```

RADIANCE

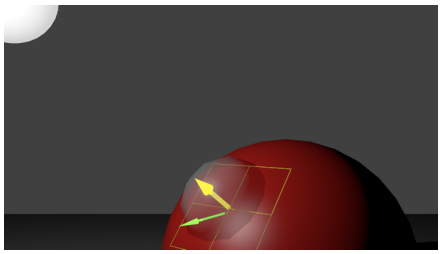
```
Vec3 Renderer::radiance(
    std::mt19937 &rng, const Ray &ray, int depth) const {

  if (depth >= MaxDepth)
    return Vec3();

  Primitive::IntersectionRecord intersectionRecord;
  if (!scene_.intersect(ray, intersectionRecord))
    return Vec3();

  int numUSamples = depth == 0 ? renderParams_.firstBounceUSamples : 1;
  int numVSamples = depth == 0 ? renderParams_.firstBounceVSamples : 1;
```

RADIANCE

```
Vec3 Renderer::radiance(
    std::mt19937 &rng, const Ray &ray, int depth) const {

  if (depth >= MaxDepth)
    return Vec3();

  Primitive::IntersectionRecord intersectionRecord;
  if (!scene_.intersect(ray, intersectionRecord))
    return Vec3();

  int numUSamples = depth == 0 ? renderParams_.firstBounceUSamples : 1;
  int numVSamples = depth == 0 ? renderParams_.firstBounceVSamples : 1;
```

RADIANCE

```
Vec3 Renderer::radiance(
    std::mt19937 &rng, const Ray &ray, int depth) const {

  if (depth >= MaxDepth)
    return Vec3();

  Primitive::IntersectionRecord intersectionRecord;
  if (!scene_.intersect(ray, intersectionRecord))
    return Vec3();

  int numUSamples = depth == 0 ? renderParams_.firstBounceUSamples : 1;
  int numVSamples = depth == 0 ? renderParams_.firstBounceVSamples : 1;
```



```
  Vec3 result;
  Sampler sampler(*this, rng, depth + 1);
  for (auto uSample = 0; uSample < numUSamples; ++uSample) {
    for (auto vSample = 0; vSample < numVSamples; ++vSample) {
      auto u = (uSample + unit(rng)) / numUSamples;
      auto v = (vSample + unit(rng)) / numVSamples;
      auto p = unit(rng);

      result += material.sample(hit, ray, sampler, u, v, p);
    }
  }
  return material.totalEmission(result / (numUSamples * numVSamples));
}
```

```
  Vec3 result;
  Sampler sampler(*this, rng, depth + 1);
  for (auto uSample = 0; uSample < numUSamples; ++uSample) {
    for (auto vSample = 0; vSample < numVSamples; ++vSample) {
      auto u = (uSample + unit(rng)) / numUSamples;
      auto v = (vSample + unit(rng)) / numVSamples;
      auto p = unit(rng);

      result += material.sample(hit, ray, sampler, u, v, p);
    }
  }
  return material.totalEmission(result / (numUSamples * numVSamples));
}
```

```
  Vec3 result;
  Sampler sampler(*this, rng, depth + 1);
  for (auto uSample = 0; uSample < numUSamples; ++uSample) {
    for (auto vSample = 0; vSample < numVSamples; ++vSample) {
      auto u = (uSample + unit(rng)) / numUSamples;
      auto v = (vSample + unit(rng)) / numVSamples;
      auto p = unit(rng);

      result += material.sample(hit, ray, sampler, u, v, p);
    }
  }
  return material.totalEmission(result / (numUSamples * numVSamples));
}
```

```
  Vec3 result;
  Sampler sampler(*this, rng, depth + 1);
  for (auto uSample = 0; uSample < numUSamples; ++uSample) {
    for (auto vSample = 0; vSample < numVSamples; ++vSample) {
      auto u = (uSample + unit(rng)) / numUSamples;
      auto v = (vSample + unit(rng)) / numVSamples;
      auto p = unit(rng);

      result += material.sample(hit, ray, sampler, u, v, p);
    }
  }
  return material.totalEmission(result / (numUSamples * numVSamples));
}
```

INTERSECTION

```
struct SpherePrimitive : Primitive {
  Sphere sphere;
  std::unique_ptr<Material> material;

  [[nodiscard]]
  bool intersect(const Ray &ray,
                 IntersectionRecord &rec) const override {
    Hit hit;
    if (!sphere.intersect(ray, hit))
      return false;
    rec = IntersectionRecord{hit, material.get()};
    return true;
  }
};
```

INTERSECTION

```
struct SpherePrimitive : Primitive {
  Sphere sphere;
  std::unique_ptr<Material> material;

  [[nodiscard]]
  bool intersect(const Ray &ray,
                 IntersectionRecord &rec) const override {
    Hit hit;
    if (!sphere.intersect(ray, hit))
      return false;
    rec = IntersectionRecord{hit, material.get()};
    return true;
  }
};
```

INTERSECTION

```
struct SpherePrimitive : Primitive {
  Sphere sphere;
  std::unique_ptr<Material> material;

  [[nodiscard]]
  bool intersect(const Ray &ray,
                 IntersectionRecord &rec) const override {
    Hit hit;
    if (!sphere.intersect(ray, hit))
      return false;
    rec = IntersectionRecord{hit, material.get()};
    return true;
  }
};
```

```
bool Sphere::intersect(const Ray &ray, Hit &hit) const noexcept {
  auto op = centre_ - ray.origin();
  auto b = op.dot(ray.direction());
  auto determinant = b * b - op.lengthSquared() + radius_ * radius_;
  if (determinant < 0)
    return false;
  auto root = sqrt(determinant);

  // [...more maths elided for clarity...]

  hit = Hit{t, inside, hitPosition, normal};
  return true;
}
```

```
bool Sphere::intersect(const Ray &ray, Hit &hit) const noexcept {
  auto op = centre_ - ray.origin();
  auto b = op.dot(ray.direction());
  auto determinant = b * b - op.lengthSquared() + radius_ * radius_;
  if (determinant < 0)
    return false;
  auto root = sqrt(determinant);

  // [...more maths elided for clarity...]

  hit = Hit{t, inside, hitPosition, normal};
  return true;
}
```

```
bool Sphere::intersect(const Ray &ray, Hit &hit) const noexcept {
  auto op = centre_ - ray.origin();
  auto b = op.dot(ray.direction());
  auto determinant = b * b - op.lengthSquared() + radius_ * radius_;
  if (determinant < 0)
    return false;
  auto root = sqrt(determinant);

  // [...more maths elided for clarity...]

  hit = Hit{t, inside, hitPosition, normal};
  return true;
}
```

```
bool Triangle::intersect(const Ray &ray, Hit &hit) const noexcept {
  // similarly...
}
```

MATERIALS

```
class Material {
public:
  virtual ~Material() = default;

  class RadianceSampler {
  public:
    virtual ~RadianceSampler() = default;

    [[nodiscard]] virtual Vec3 sample(const Ray &ray) const = 0;
  };

  [[nodiscard]] virtual Vec3 sample(
        const Hit &hit, const Ray &incoming,
        const RadianceSampler &radianceSampler,
        double u, double v, double p) const = 0;
};
```

MATERIALS

```
class Material {
public:
  virtual ~Material() = default;

  class RadianceSampler {
  public:
    virtual ~RadianceSampler() = default;

    [[nodiscard]] virtual Vec3 sample(const Ray &ray) const = 0;
  };

  [[nodiscard]] virtual Vec3 sample(
        const Hit &hit, const Ray &incoming,
        const RadianceSampler &radianceSampler,
        double u, double v, double p) const = 0;
};
```

MATERIALS

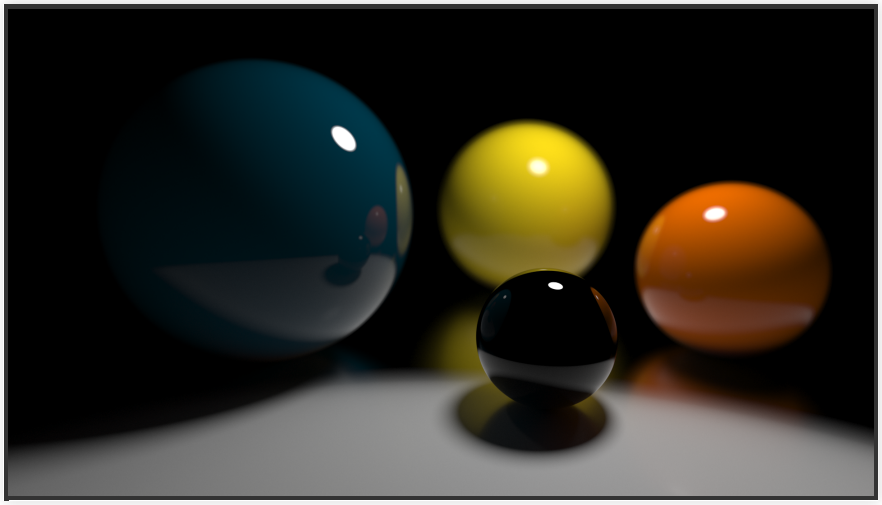
```
class Material {
public:
  virtual ~Material() = default;

  class RadianceSampler {
  public:
    virtual ~RadianceSampler() = default;

    [[nodiscard]] virtual Vec3 sample(const Ray &ray) const = 0;
  };

  [[nodiscard]] virtual Vec3 sample(
        const Hit &hit, const Ray &incoming,
        const RadianceSampler &radianceSampler,
        double u, double v, double p) const = 0;
};
```

```
class ShinyMaterial : public Material {
  MaterialSpec mat_;
public:
  Vec3 sample(
        const Hit &hit, const Ray &incoming,
        const RadianceSampler &radianceSampler, double u,
        double v, double p) const override {

    if (p < mat_.reflectivity) {
      return radianceSampler.sample(
          Ray(hit.position, coneSample(/*...reflection...*/, u, v)));
    }

    return mat_.diffuse
           * radianceSampler.sample(
               Ray(hit.position, hemisphereSample(hit.normal, u, v)));
  }
};
```

```
class ShinyMaterial : public Material {
  MaterialSpec mat_;
public:
  Vec3 sample(
        const Hit &hit, const Ray &incoming,
        const RadianceSampler &radianceSampler, double u,
        double v, double p) const override {

    if (p < mat_.reflectivity) {
      return radianceSampler.sample(
          Ray(hit.position, coneSample(/*...reflection...*/, u, v)));
    }

    return mat_.diffuse
           * radianceSampler.sample(
               Ray(hit.position, hemisphereSample(hit.normal, u, v)));
  }
};
```

```
class ShinyMaterial : public Material {
  MaterialSpec mat_;
public:
  Vec3 sample(
        const Hit &hit, const Ray &incoming,
        const RadianceSampler &radianceSampler, double u,
        double v, double p) const override {

    if (p < mat_.reflectivity) {
      return radianceSampler.sample(
          Ray(hit.position, coneSample(/*...reflection...*/, u, v)));
    }

    return mat_.diffuse
           * radianceSampler.sample(
               Ray(hit.position, hemisphereSample(hit.normal, u, v)));
  }
};
```

RESULTS



Scene credit: Michael Fogleman

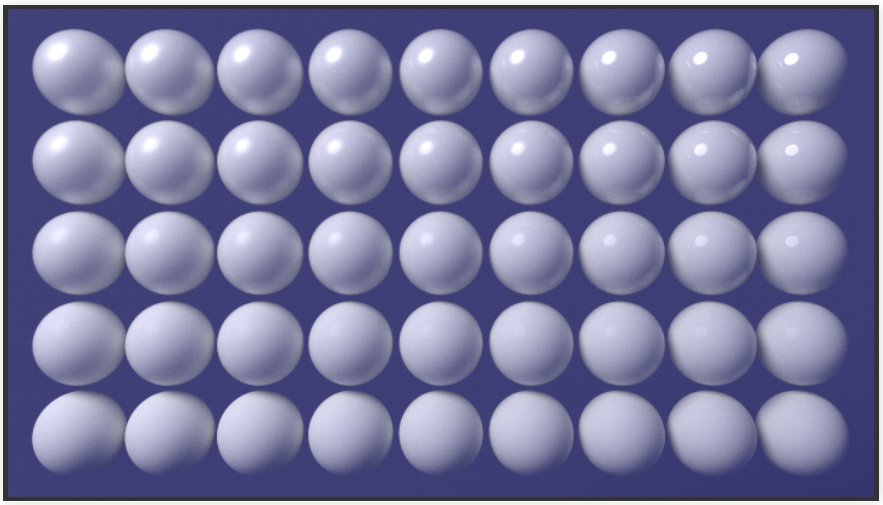
THINGS I LIKED

- Code layout
- Testability
- Performance

THINGS I DIDN'T LIKE

- Giving up `std::optional`
- (Arguable) overuse of `virtual`

FUNCTIONAL PROGRAMMING



FUNCTIONAL PROGRAMMING

- `ranges_v3`
- `tl::optional`

```
struct TrianglePrimitive {
  Triangle shape;
  Material material;
};

struct SpherePrimitive {
  Sphere shape;
  Material material;
};

using Primitive = std::variant<TrianglePrimitive, SpherePrimitive>;
```

```
struct TrianglePrimitive {
  Triangle shape;
  Material material;
};

struct SpherePrimitive {
  Sphere shape;
  Material material;
};

using Primitive = std::variant<TrianglePrimitive, SpherePrimitive>;
```

RENDER

```
auto renderOnePixel = [...](auto tuple) {
  auto [y, x] = tuple;
  std::mt19937 rng(...);
  return radiance(scene, rng,
        camera.randomRay(x, y, rng), 0, renderParams);
};

return ArrayOutput(width, height,
  view::cartesian_product(view::ints(0, height),
                          view::ints(0, width))
  | view::transform(renderOnePixel));
```

RENDER

```
auto renderOnePixel = [...](auto tuple) {
  auto [y, x] = tuple;
  std::mt19937 rng(...);
  return radiance(scene, rng,
        camera.randomRay(x, y, rng), 0, renderParams);
};

return ArrayOutput(width, height,
  view::cartesian_product(view::ints(0, height),
                          view::ints(0, width))
  | view::transform(renderOnePixel));
```

RENDER

```
auto renderOnePixel = [...](auto tuple) {
  auto [y, x] = tuple;
  std::mt19937 rng(...);
  return radiance(scene, rng,
        camera.randomRay(x, y, rng), 0, renderParams);
};

return ArrayOutput(width, height,
  view::cartesian_product(view::ints(0, height),
                          view::ints(0, width))
  | view::transform(renderOnePixel));
```

RENDER

```
auto renderOnePixel = [...](auto tuple) {
  auto [y, x] = tuple;
  std::mt19937 rng(...);
  return radiance(scene, rng,
        camera.randomRay(x, y, rng), 0, renderParams);
};

return ArrayOutput(width, height,
  view::cartesian_product(view::ints(0, height),
                          view::ints(0, width))
  | view::transform(renderOnePixel));
```

RADIANCE

```
auto radianceForRay = [&](const Ray &ray) {
  return radiance(scene, rng, ray, depth + 1, renderParams);
};

const auto incomingLight = accumulate(
    views::cartesian_product(views::ints(0, numVSamples),
                             views::ints(0, numUSamples))
        | views::transform(toUVSample)
        | views::transform([&](auto s) {
            return radianceAtIntersection(
                radianceForRay, *intersectionRecord,
                ray, s.first, s.second, unit(rng));
          }),
    Vec3());
```

RADIANCE

```
auto radianceForRay = [&](const Ray &ray) {
  return radiance(scene, rng, ray, depth + 1, renderParams);
};

const auto incomingLight = accumulate(
    views::cartesian_product(views::ints(0, numVSamples),
                             views::ints(0, numUSamples))
        | views::transform(toUVSample)
        | views::transform([&](auto s) {
            return radianceAtIntersection(
                radianceForRay, *intersectionRecord,
                ray, s.first, s.second, unit(rng));
          }),
    Vec3());
```

RADIANCE

```
auto radianceForRay = [&](const Ray &ray) {
  return radiance(scene, rng, ray, depth + 1, renderParams);
};

const auto incomingLight = accumulate(
    views::cartesian_product(views::ints(0, numVSamples),
                             views::ints(0, numUSamples))
        | views::transform(toUVSample)
        | views::transform([&](auto s) {
            return radianceAtIntersection(
                radianceForRay, *intersectionRecord,
                ray, s.first, s.second, unit(rng));
          }),
    Vec3());
```

RADIANCE

```
auto radianceForRay = [&](const Ray &ray) {
  return radiance(scene, rng, ray, depth + 1, renderParams);
};

const auto incomingLight = accumulate(
    views::cartesian_product(views::ints(0, numVSamples),
                             views::ints(0, numUSamples))
        | views::transform(toUVSample)
        | views::transform([&](auto s) {
            return radianceAtIntersection(
                radianceForRay, *intersectionRecord,
                ray, s.first, s.second, unit(rng));
          }),
    Vec3());
```

RADIANCE

```
auto radianceForRay = [&](const Ray &ray) {
  return radiance(scene, rng, ray, depth + 1, renderParams);
};

const auto incomingLight = accumulate(
    views::cartesian_product(views::ints(0, numVSamples),
                             views::ints(0, numUSamples))
        | views::transform(toUVSample)
        | views::transform([&](auto s) {
            return radianceAtIntersection(
                radianceForRay, *intersectionRecord,
                ray, s.first, s.second, unit(rng));
          }),
    Vec3());
```

RADIANCE

```
auto radianceForRay = [&](const Ray &ray) {
  return radiance(scene, rng, ray, depth + 1, renderParams);
};

const auto incomingLight = accumulate(
    views::cartesian_product(views::ints(0, numVSamples),
                             views::ints(0, numUSamples))
        | views::transform(toUVSample)
        | views::transform([&](auto s) {
            return radianceAtIntersection(
                radianceForRay, *intersectionRecord,
                ray, s.first, s.second, unit(rng));
          }),
    Vec3());
```

```
template <typename RadianceFunc>
Vec3 radianceAtIntersection(
        RadianceFunc &&radiance,
        const IntersectionRecord &intersectionRecord,
        const Ray &ray, double u, double v, double p) {
  // [...lots of material stuff...]
  if (p < reflectivity) {
    const auto newRay =
        Ray(hit.position, coneSample(/*...*/, u, v));
    return radiance(newRay);
  } else {
    const auto newRay = Ray(hit.position,
        hemisphereSample(/*...*/, u, v);
    return mat.diffuse * radiance(newRay);
  }
}
```

```
template <typename RadianceFunc>
Vec3 radianceAtIntersection(
        RadianceFunc &&radiance,
        const IntersectionRecord &intersectionRecord,
        const Ray &ray, double u, double v, double p) {
  // [...lots of material stuff...]
  if (p < reflectivity) {
    const auto newRay =
        Ray(hit.position, coneSample(/*...*/, u, v));
    return radiance(newRay);
  } else {
    const auto newRay = Ray(hit.position,
        hemisphereSample(/*...*/, u, v);
    return mat.diffuse * radiance(newRay);
  }
}
```

INTERSECTION

```
struct IntersectVisitor {
  const Ray &ray;

  template <typename PrimT>
  auto operator()(const PrimT &primitive) const {
    return primitive.shape.intersect(ray)
        .map([&primitive](auto hit) {
            return IntersectionRecord{hit, primitive.material};
        });
  }
};

tl::optional<IntersectionRecord> intersect(
    const Primitive &primitive, const Ray &ray) {
  return std::visit(IntersectVisitor{ray}, primitive);
}
```

INTERSECTION

```
struct IntersectVisitor {
  const Ray &ray;

  template <typename PrimT>
  auto operator()(const PrimT &primitive) const {
    return primitive.shape.intersect(ray)
        .map([&primitive](auto hit) {
            return IntersectionRecord{hit, primitive.material};
        });
  }
};

tl::optional<IntersectionRecord> intersect(
    const Primitive &primitive, const Ray &ray) {
  return std::visit(IntersectVisitor{ray}, primitive);
}
```

INTERSECTION

```
struct IntersectVisitor {
  const Ray &ray;

  template <typename PrimT>
  auto operator()(const PrimT &primitive) const {
    return primitive.shape.intersect(ray)
        .map([&primitive](auto hit) {
            return IntersectionRecord{hit, primitive.material};
        });
  }
};

tl::optional<IntersectionRecord> intersect(
    const Primitive &primitive, const Ray &ray) {
  return std::visit(IntersectVisitor{ray}, primitive);
}
```

INTERSECTION

```
struct IntersectVisitor {
  const Ray &ray;

  template <typename PrimT>
  auto operator()(const PrimT &primitive) const {
    return primitive.shape.intersect(ray)
        .map([&primitive](auto hit) {
            return IntersectionRecord{hit, primitive.material};
        });
  }
};

tl::optional<IntersectionRecord> intersect(
    const Primitive &primitive, const Ray &ray) {
  return std::visit(IntersectVisitor{ray}, primitive);
}
```

INTERSECTION

```
struct IntersectVisitor {
  const Ray &ray;

  template <typename PrimT>
  auto operator()(const PrimT &primitive) const {
    return primitive.shape.intersect(ray)
        .map([&primitive](auto hit) {
            return IntersectionRecord{hit, primitive.material};
        });
  }
};

tl::optional<IntersectionRecord> intersect(
    const Primitive &primitive, const Ray &ray) {
  return std::visit(IntersectVisitor{ray}, primitive);
}
```

```
tl::optional<Hit> Sphere::intersect(const Ray &ray) const noexcept {
  const auto determinant = /*...maths...*/;
  return safeSqrt(determinant)

      .and_then([&b](double root) -> tl::optional<double> {
        /* ...calc minusT, plusT.. */
        if (/* ...too close to zero...*/)
          return tl::nullopt;
        return minusT > Epsilon ? minusT : plusT;
      })

      .map([this, &ray](double t) {
        return Hit{t, /*...*/};
      });
}
```

```
tl::optional<Hit> Sphere::intersect(const Ray &ray) const noexcept {
  const auto determinant = /*...maths...*/;
  return safeSqrt(determinant)

      .and_then([&b](double root) -> tl::optional<double> {
        /* ...calc minusT, plusT.. */
        if (/* ...too close to zero...*/)
          return tl::nullopt;
        return minusT > Epsilon ? minusT : plusT;
      })

      .map([this, &ray](double t) {
        return Hit{t, /*...*/};
      });
}
```

```
tl::optional<Hit> Sphere::intersect(const Ray &ray) const noexcept {
  const auto determinant = /*...maths...*/;
  return safeSqrt(determinant)

      .and_then([&b](double root) -> tl::optional<double> {
        /* ...calc minusT, plusT.. */
        if (/* ...too close to zero...*/)
          return tl::nullopt;
        return minusT > Epsilon ? minusT : plusT;
      })

      .map([this, &ray](double t) {
        return Hit{t, /*...*/};
      });
}
```

```
tl::optional<Hit> Sphere::intersect(const Ray &ray) const noexcept {
  const auto determinant = /*...maths...*/;
  return safeSqrt(determinant)

      .and_then([&b](double root) -> tl::optional<double> {
        /* ...calc minusT, plusT.. */
        if (/* ...too close to zero...*/)
          return tl::nullopt;
        return minusT > Epsilon ? minusT : plusT;
      })

      .map([this, &ray](double t) {
        return Hit{t, /*...*/};
      });
}
```

```
tl::optional<Hit> Sphere::intersect(const Ray &ray) const noexcept {
  const auto determinant = /*...maths...*/;
  return safeSqrt(determinant)

      .and_then([&b](double root) -> tl::optional<double> {
        /* ...calc minusT, plusT.. */
        if (/* ...too close to zero...*/)
          return tl::nullopt;
        return minusT > Epsilon ? minusT : plusT;
      })

      .map([this, &ray](double t) {
        return Hit{t, /*...*/};
      });
}
```

```
tl::optional<Hit> Sphere::intersect(const Ray &ray) const noexcept {
  const auto determinant = /*...maths...*/;
  return safeSqrt(determinant)

      .and_then([&b](double root) -> tl::optional<double> {
        /* ...calc minusT, plusT.. */
        if (/* ...too close to zero...*/)
          return tl::nullopt;
        return minusT > Epsilon ? minusT : plusT;
      })

      .map([this, &ray](double t) {
        return Hit{t, /*...*/};
      });
}
```

```
tl::optional<Hit> Sphere::intersect(const Ray &ray) const noexcept {
  const auto determinant = /*...maths...*/;
  return safeSqrt(determinant)

      .and_then([&b](double root) -> tl::optional<double> {
        /* ...calc minusT, plusT.. */
        if (/* ...too close to zero...*/)
          return tl::nullopt;
        return minusT > Epsilon ? minusT : plusT;
      })

      .map([this, &ray](double t) {
        return Hit{t, /*...*/};
      });
}
```

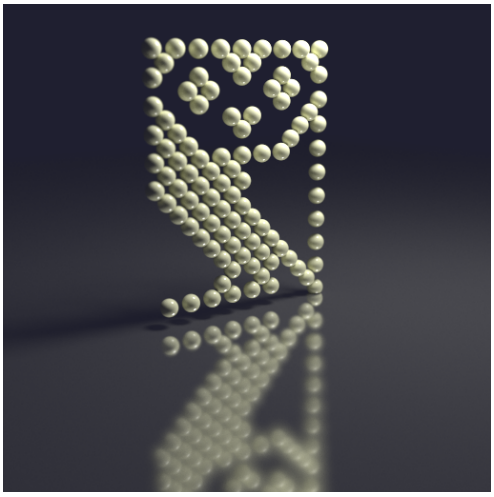
THINGS I LIKED

- `const :allthethings:`
- Code clearer…maybe?
- Testability

THINGS I DIDN'T LIKE

- Cryptic compiler error messages
- Concern I've broken FP rules (`rng`…)
- Performance
  - `std::mt19937 rng` per pixel?

DATA-ORIENTED DESIGN



- Design around most common operations
- Design data by access patterns
- Avoid branches
- Polymorphism by separation

CHANGES

- Intersection is most common operation
  - Only need nearest
- Two types of object
  - Separate into two lists

SCENE

```
class Scene {
  std::vector<TriangleVertices> triangleVerts_;
  std::vector<TriangleNormals> triangleNormals_;
  std::vector<Material> triangleMaterials_;

  std::vector<Sphere> spheres_;
  std::vector<Material> sphereMaterials_;
```

SCENE

```
class Scene {
  std::vector<TriangleVertices> triangleVerts_;
  std::vector<TriangleNormals> triangleNormals_;
  std::vector<Material> triangleMaterials_;

  std::vector<Sphere> spheres_;
  std::vector<Material> sphereMaterials_;
```

SCENE

```
class Scene {
  std::vector<TriangleVertices> triangleVerts_;
  std::vector<TriangleNormals> triangleNormals_;
  std::vector<Material> triangleMaterials_;

  std::vector<Sphere> spheres_;
  std::vector<Material> sphereMaterials_;
```

SCENE

```
class Scene {
  std::vector<TriangleVertices> triangleVerts_;
  std::vector<TriangleNormals> triangleNormals_;
  std::vector<Material> triangleMaterials_;

  std::vector<Sphere> spheres_;
  std::vector<Material> sphereMaterials_;
```

SCENE

```
class Scene {
  std::vector<TriangleVertices> triangleVerts_;
  std::vector<TriangleNormals> triangleNormals_;
  std::vector<Material> triangleMaterials_;

  std::vector<Sphere> spheres_;
  std::vector<Material> sphereMaterials_;
```

INTERSECTION - SPHERES

```
double nearestDist = nearerThan;
std::optional<size_t> nearestIndex;

for (size_t sphereIdx = 0; sphereIdx < spheres_.size(); ++sphereIdx) {
  // ...maths...
  if (!hit) continue;

  if (distance < currentNearestDist) {
    nearestIndex = sphereIdx; nearestDist = distance;
  }
}

if (!nearestIndex) return {}; // missed all spheres

// ...more maths to calc hit positions, normal...
return IntersectionRecord{Hit{...}, sphereMaterials_[*nearestIndex]};
```

INTERSECTION - SPHERES

```
double nearestDist = nearerThan;
std::optional<size_t> nearestIndex;

for (size_t sphereIdx = 0; sphereIdx < spheres_.size(); ++sphereIdx) {
  // ...maths...
  if (!hit) continue;

  if (distance < currentNearestDist) {
    nearestIndex = sphereIdx; nearestDist = distance;
  }
}

if (!nearestIndex) return {}; // missed all spheres

// ...more maths to calc hit positions, normal...
return IntersectionRecord{Hit{...}, sphereMaterials_[*nearestIndex]};
```

INTERSECTION - SPHERES

```
double nearestDist = nearerThan;
std::optional<size_t> nearestIndex;

for (size_t sphereIdx = 0; sphereIdx < spheres_.size(); ++sphereIdx) {
  // ...maths...
  if (!hit) continue;

  if (distance < currentNearestDist) {
    nearestIndex = sphereIdx; nearestDist = distance;
  }
}

if (!nearestIndex) return {}; // missed all spheres

// ...more maths to calc hit positions, normal...
return IntersectionRecord{Hit{...}, sphereMaterials_[*nearestIndex]};
```

INTERSECTION - SPHERES

```
double nearestDist = nearerThan;
std::optional<size_t> nearestIndex;

for (size_t sphereIdx = 0; sphereIdx < spheres_.size(); ++sphereIdx) {
  // ...maths...
  if (!hit) continue;

  if (distance < currentNearestDist) {
    nearestIndex = sphereIdx; nearestDist = distance;
  }
}

if (!nearestIndex) return {}; // missed all spheres

// ...more maths to calc hit positions, normal...
return IntersectionRecord{Hit{...}, sphereMaterials_[*nearestIndex]};
```

INTERSECTION - SPHERES

```
double nearestDist = nearerThan;
std::optional<size_t> nearestIndex;

for (size_t sphereIdx = 0; sphereIdx < spheres_.size(); ++sphereIdx) {
  // ...maths...
  if (!hit) continue;

  if (distance < currentNearestDist) {
    nearestIndex = sphereIdx; nearestDist = distance;
  }
}

if (!nearestIndex) return {}; // missed all spheres

// ...more maths to calc hit positions, normal...
return IntersectionRecord{Hit{...}, sphereMaterials_[*nearestIndex]};
```

INTERSECTION - SPHERES

```
double nearestDist = nearerThan;
std::optional<size_t> nearestIndex;

for (size_t sphereIdx = 0; sphereIdx < spheres_.size(); ++sphereIdx) {
  // ...maths...
  if (!hit) continue;

  if (distance < currentNearestDist) {
    nearestIndex = sphereIdx; nearestDist = distance;
  }
}

if (!nearestIndex) return {}; // missed all spheres

// ...more maths to calc hit positions, normal...
return IntersectionRecord{Hit{...}, sphereMaterials_[*nearestIndex]};
```

INTERSECTION - SPHERES

```
double nearestDist = nearerThan;
std::optional<size_t> nearestIndex;

for (size_t sphereIdx = 0; sphereIdx < spheres_.size(); ++sphereIdx) {
  // ...maths...
  if (!hit) continue;

  if (distance < currentNearestDist) {
    nearestIndex = sphereIdx; nearestDist = distance;
  }
}

if (!nearestIndex) return {}; // missed all spheres

// ...more maths to calc hit positions, normal...
return IntersectionRecord{Hit{...}, sphereMaterials_[*nearestIndex]};
```

INTERSECTION - TRIANGLES

```
for (size_t triIdx = 0; triIdx < triangleVerts_.size(); ++triIdx) {
  // ...calc u...maths, only needs triangle vertices...
  if (u < 0 || u > 1) continue;
  // ...calc v...
  if (v < 0 || u + v > 1) continue;
  // ...calc dist...
  if (dist < nearest) { /* note this as nearest */ }
}

// ...more maths to calculate actual normal...
return IntersectionRecord{Hit{...}, triangleMaterials_[*nearestIndex]};
```

INTERSECTION - TRIANGLES

```
for (size_t triIdx = 0; triIdx < triangleVerts_.size(); ++triIdx) {
  // ...calc u...maths, only needs triangle vertices...
  if (u < 0 || u > 1) continue;
  // ...calc v...
  if (v < 0 || u + v > 1) continue;
  // ...calc dist...
  if (dist < nearest) { /* note this as nearest */ }
}

// ...more maths to calculate actual normal...
return IntersectionRecord{Hit{...}, triangleMaterials_[*nearestIndex]};
```

INTERSECTION - TRIANGLES

```
for (size_t triIdx = 0; triIdx < triangleVerts_.size(); ++triIdx) {
  // ...calc u...maths, only needs triangle vertices...
  if (u < 0 || u > 1) continue;
  // ...calc v...
  if (v < 0 || u + v > 1) continue;
  // ...calc dist...
  if (dist < nearest) { /* note this as nearest */ }
}

// ...more maths to calculate actual normal...
return IntersectionRecord{Hit{...}, triangleMaterials_[*nearestIndex]};
```

INTERSECTION - TRIANGLES

```
for (size_t triIdx = 0; triIdx < triangleVerts_.size(); ++triIdx) {
  // ...calc u...maths, only needs triangle vertices...
  if (u < 0 || u > 1) continue;
  // ...calc v...
  if (v < 0 || u + v > 1) continue;
  // ...calc dist...
  if (dist < nearest) { /* note this as nearest */ }
}

// ...more maths to calculate actual normal...
return IntersectionRecord{Hit{...}, triangleMaterials_[*nearestIndex]};
```

INTERSECTION - TRIANGLES

```
for (size_t triIdx = 0; triIdx < triangleVerts_.size(); ++triIdx) {
  // ...calc u...maths, only needs triangle vertices...
  if (u < 0 || u > 1) continue;
  // ...calc v...
  if (v < 0 || u + v > 1) continue;
  // ...calc dist...
  if (dist < nearest) { /* note this as nearest */ }
}

// ...more maths to calculate actual normal...
return IntersectionRecord{Hit{...}, triangleMaterials_[*nearestIndex]};
```

INTERSECTION - TRIANGLES

```
for (size_t triIdx = 0; triIdx < triangleVerts_.size(); ++triIdx) {
  // ...calc u...maths, only needs triangle vertices...
  if (u < 0 || u > 1) continue;
  // ...calc v...
  if (v < 0 || u + v > 1) continue;
  // ...calc dist...
  if (dist < nearest) { /* note this as nearest */ }
}

// ...more maths to calculate actual normal...
return IntersectionRecord{Hit{...}, triangleMaterials_[*nearestIndex]};
```

REST BROADLY THE SAME

THINGS I LIKED

- Ability to optimise
- Performance

THINGS I LIKED

- Ability to optimise
- Performance
  - with caveat

THINGS I DIDN'T LIKE

- Testability
- Difficulty to change

CONCLUSIONS

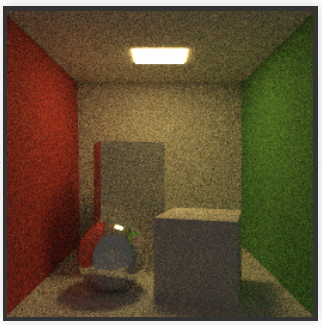
FAVOURITE?

- All of the above!!
- C++ is best when we can blend all its features

PERFORMANCE

- Intel(R) Core(TM) i9-9980XE CPU @ 3.00GHz
- `cpupower frequency-set --governor performance`
- Single threaded
- GCC 9.2

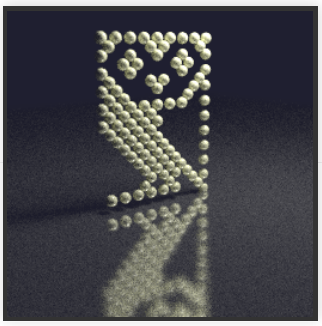
CORNELL BOX SCENE



256x256 32spp (1 sphere, 38 triangles)

Style	Render time (secs)
Object Oriented	72
Functional	82
Data-Oriented	72

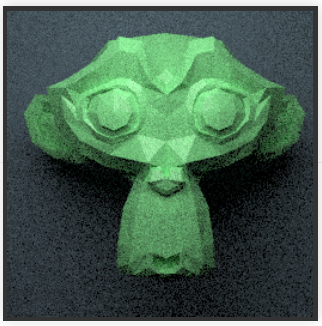
OWL SCENE



128spp (100 spheres, 12 triangles)

Style	Render time (secs)
Object Oriented	96
Functional	101
Data-Oriented	64

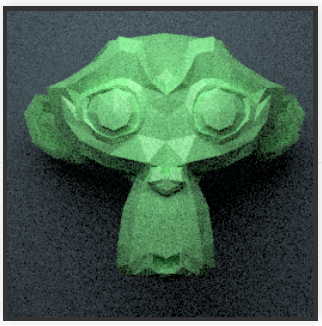
SUZANNE



256x256 8spp (2 spheres, 970 triangles)

Style	Render time (secs)
Object Oriented	80
Functional	119
Data-Oriented	106

SUZANNE



256x256 8spp (2 spheres, 970 triangles)

Style	Render time (secs)
Object Oriented	80
Functional	119
Data-Oriented	106

What on earth!?!

WHAT HAPPENED?

OBJECT ORIENTED

```
252,203,645,149   instructions    #    2.71  insn per cycle
 23,158,824,048   branches        #  845.438 M/sec
    139,741,218   branch-misses   #    0.60% of all branches
```

FUNCTIONAL

```
238,866,691,159   instructions    #    1.78  insn per cycle
 21,881,070,251   branches        #  560.513 M/sec
  1,105,066,725   branch-misses   #    5.05% of all branches
```

DATA-ORIENTED DESIGN

```
154,821,779,748   instructions    #    1.34  insn per cycle
 10,353,392,805   branches        #  305.213 M/sec
  1,242,670,094   branch-misses   #   12.00% of all branches
```

OBJECT ORIENTED

```
252,203,645,149   instructions    #    2.71  insn per cycle
 23,158,824,048   branches        #  845.438 M/sec
    139,741,218   branch-misses   #    0.60% of all branches
```

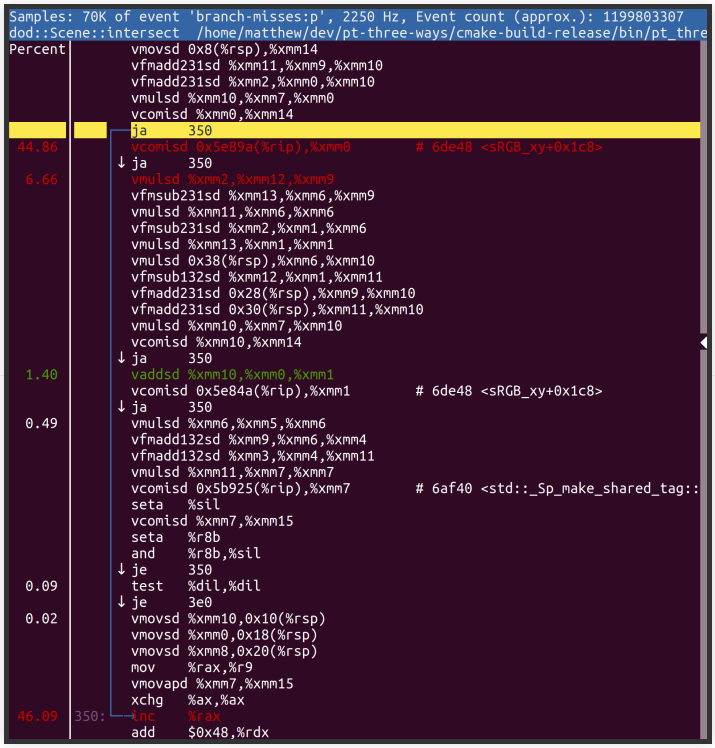
FUNCTIONAL

```
238,866,691,159   instructions    #    1.78  insn per cycle
 21,881,070,251   branches        #  560.513 M/sec
  1,105,066,725   branch-misses   #    5.05% of all branches
```

DATA-ORIENTED DESIGN

```
154,821,779,748   instructions    #    1.34  insn per cycle
 10,353,392,805   branches        #  305.213 M/sec
  1,242,670,094   branch-misses   #   12.00% of all branches
```

12% of all branches!?!


Samples: 70K of event 'branch-misses:p', 2250 Hz, Event count (approx.): 1199803307
dod::Scene::intersect /home/matthew/dev/pt-three-ways/cmake-build-release/bin/pt_thre
Percent
vmovsd 0x8(%rsp),%xmm14
vfmadd231sd %xmm11,%xmm9,%xmm10
vfmadd231sd %xmm2,%xmm0,%xmm10
vmulsd %xmm10,%xmm7,%xmm0
vcomisd %xmm0,%xmm14
ja 350
44.86 vcomisd 0x5e89a(%rip),%xmm0 # 6de48 <sRGB_xy+0x1c8>
ja 350
6.66 vmulsd %xmm2,%xmm12,%xmm9
vfmsub231sd %xmm13,%xmm6,%xmm9
vmulsd %xmm11,%xmm6,%xmm6
vfmsub231sd %xmm2,%xmm1,%xmm6
vmulsd %xmm13,%xmm1,%xmm1
vmulsd 0x38(%rsp),%xmm6,%xmm10
vfmsub132sd %xmm12,%xmm1,%xmm11
vfmadd231sd 0x28(%rsp),%xmm9,%xmm10
vfmadd231sd 0x30(%rsp),%xmm11,%xmm10
vmulsd %xmm10,%xmm7,%xmm10
vcomisd %xmm10,%xmm14
ja 350
1.40 vaddsd %xmm10,%xmm0,%xmm1
vcomisd 0x5e84a(%rip),%xmm1 # 6de48 <sRGB_xy+0x1c8>
ja 350
0.49 vmulsd %xmm6,%xmm5,%xmm6
vfmadd132sd %xmm9,%xmm6,%xmm4
vfmadd132sd %xmm3,%xmm4,%xmm11
vmulsd %xmm11,%xmm7,%xmm7
vcomisd 0x5b925(%rip),%xmm7 # 6af40 <std::_Sp_make_shared_tag::
seta %sil
vcomisd %xmm7,%xmm15
seta %r8b
and %r8b,%sil
je 350
0.09 test %dil,%dil
je 3e0
0.02 vmovsd %xmm10,0x10(%rsp)
vmovsd %xmm0,0x18(%rsp)
vmovsd %xmm8,0x20(%rsp)
mov %rax,%r9
vmovapd %xmm7,%xmm15
xchg %ax,%ax
46.09 350: inc %rax
add $0x48,%rdx

```
for (/* all triangles */) {
  auto u = calcU(/*...*/);
  if (u < 0 || u > 1) {
    continue;
  }
  auto v = calcV(/*...*/);
  if (v < 0 || u + v > 1) {
    continue;
  }
  auto dist = calcD(/*..*/);
  if (dist < nearest) {
    nearest = dist;
  }
}
```

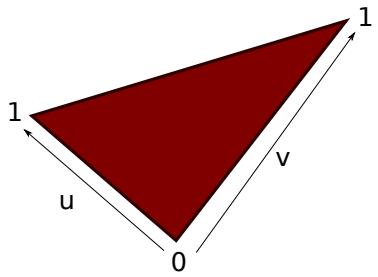
```
for (/* all triangles */) {
  auto u = calcU(/*...*/);
  if (u < 0 || u > 1) {
    continue;
  }
  auto v = calcV(/*...*/);
  if (v < 0 || u + v > 1) {
    continue;
  }
  auto dist = calcD(/*..*/);
  if (dist < nearest) {
    nearest = dist;
  }
}
```

```
vcomisd xmm14, xmm0 ; 0 >= u?
ja skip
vcomisd xmm0, [1.0] ; u >= 1.0?
ja skip
```

```
for (/* all triangles */) {
  auto u = calcU(/*...*/);
  if (u < 0 || u > 1) {
    continue;
  }
  auto v = calcV(/*...*/);
  if (v < 0 || u + v > 1) {
    continue;
  }
  auto dist = calcD(/*..*/);
  if (dist < nearest) {
    nearest = dist;
  }
}
```

```
vcomisd xmm14, xmm0 ; 0 >= u?
ja skip
vcomisd xmm0, [1.0] ; u >= 1.0?
ja skip
```

ray/plane intersection point



(not to scale)

BRANCH PREDICTION

- `u < 0` unpredictable
- `u > 1` unpredictable

BRANCH PREDICTION

- `u < 0` unpredictable
- `u > 1` unpredictable
- `(u < 0 || u > 1)` *should be* predictable*

BRANCH PREDICTION

- `u < 0` unpredictable
- `u > 1` unpredictable
- `(u < 0 || u > 1)` *should be* predictable*
- `(u < 0 || u > 1 || v < 0 || u + v > 1)` more so

BRANCH PREDICTION

- `u < 0` unpredictable
- `u > 1` unpredictable
- `(u < 0 || u > 1)` *should be* predictable*
- `(u < 0 || u > 1 || v < 0 || u + v > 1)` more so

* Provided compiler combines conditions…

BRANCH PREDICTION

- `u < 0` unpredictable
- `u > 1` unpredictable
- `(u < 0 || u > 1)` *should be* predictable*
- `(u < 0 || u > 1 || v < 0 || u + v > 1)` more so

* Provided compiler combines conditions…

`(u < 0) | (u > 1) | ...`

```
auto u = calcU(/*...*/);
if (u < 0 || u > 1) {
  continue;
}
auto v = calcV(/*...*/);
if (v <  0 || u + v > 1) {
  continue;
}
```

```
auto u = calcU(/*...*/);
auto v = calcV(/*...*/);
if ((u < 0) | (u > 1)
    | (v < 0) | (u + v > 1)) {
  continue;
}
```

FINAL STATS

Suzanne DoD: 106s → 68s (36% faster)

FINAL STATS

Scene	OO	FP	DoD
Cornell	72 → 64	82 → 64	72 → 56
Owl	96 → 96	101 → 100	64 → 64
Suzanne	80 → 104 (!)	119 → 82	106 → 68

IF I HAD MORE TIME...

- Code on GitHub
- Threading
- Devirtualisation
- Future directions...
- DoD improvements
- Thanks

GO WRITE SOMETHING COOL!