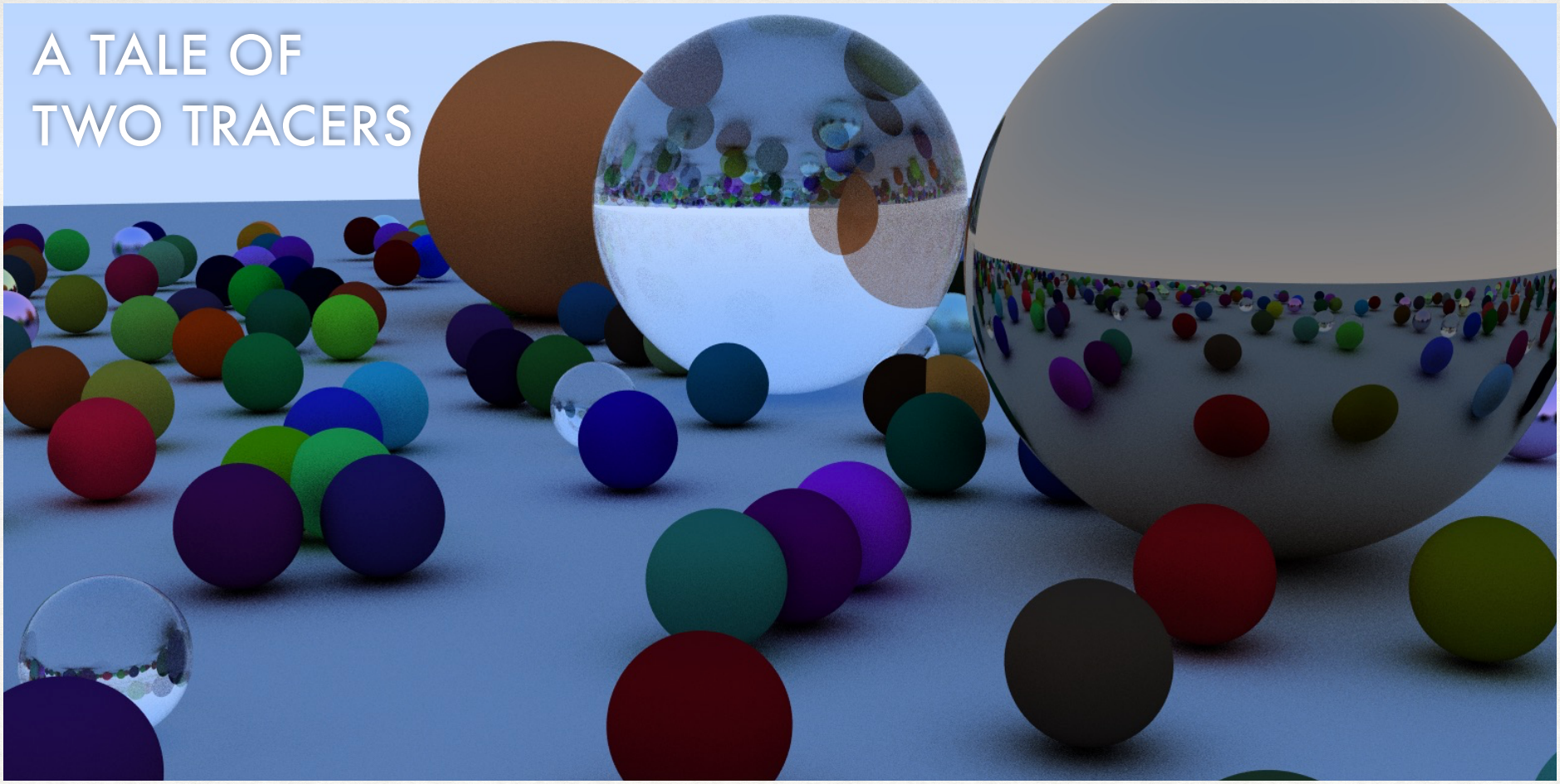
A TALE OF TWO TRACERS



Jonathan Blocksom
@jblocksom
Capital One

Jeff Biggus
@hyperjeff
http://blog.hyperjeff.net

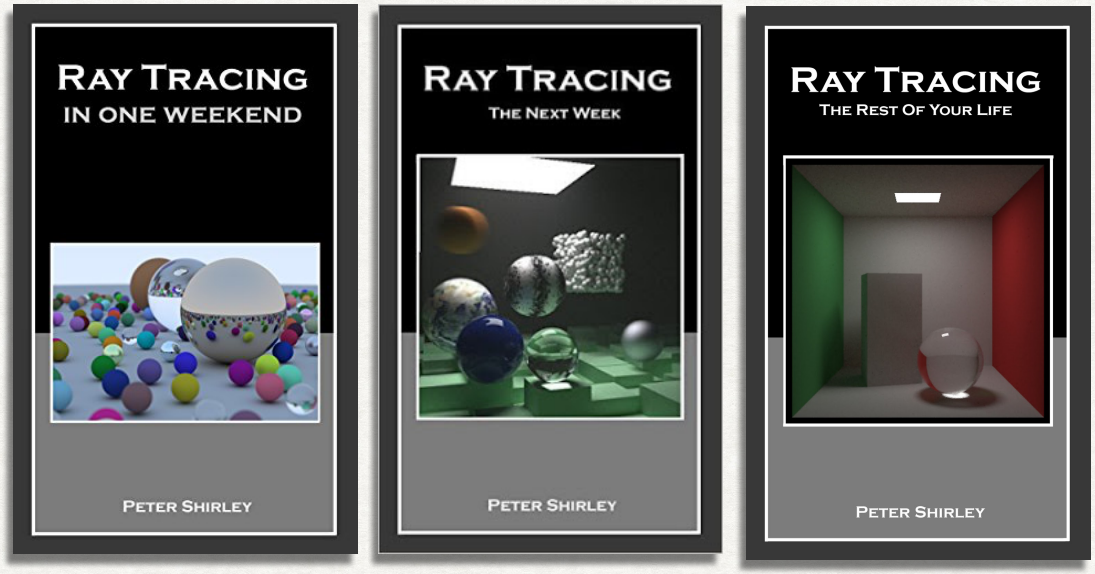
360iDev 2016
August 23
Denver, CO

WHY?

IMPLEMENT A RAY TRACER IN SWIFT AND AGAIN IN METAL

- Ray Tracing:
  Well defined problem, tons or prior art, resources, simple, extendable
- Why Swift?
  It's new (or was when we started)
- Why Metal?
  It's new and on the GPU
- Why Both?
  Compare type-safe, fancy approach with performance oriented GPU approach

QUICK SHOUT OUT

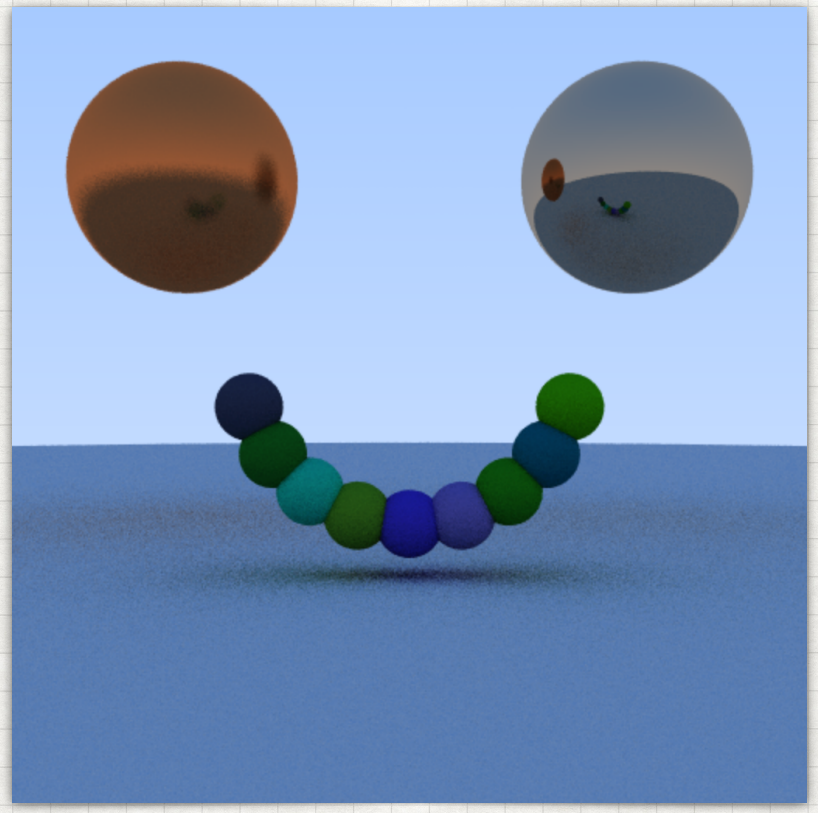


Peter Shirley — @Peter_shirley, http://psgraphics.blogspot.com
https://www.amazon.com/Ray-Tracing-Weekend-Minibooks-Book-ebook/dp/B01B5AODD8

AGENDA

WE HAVE A PLAN

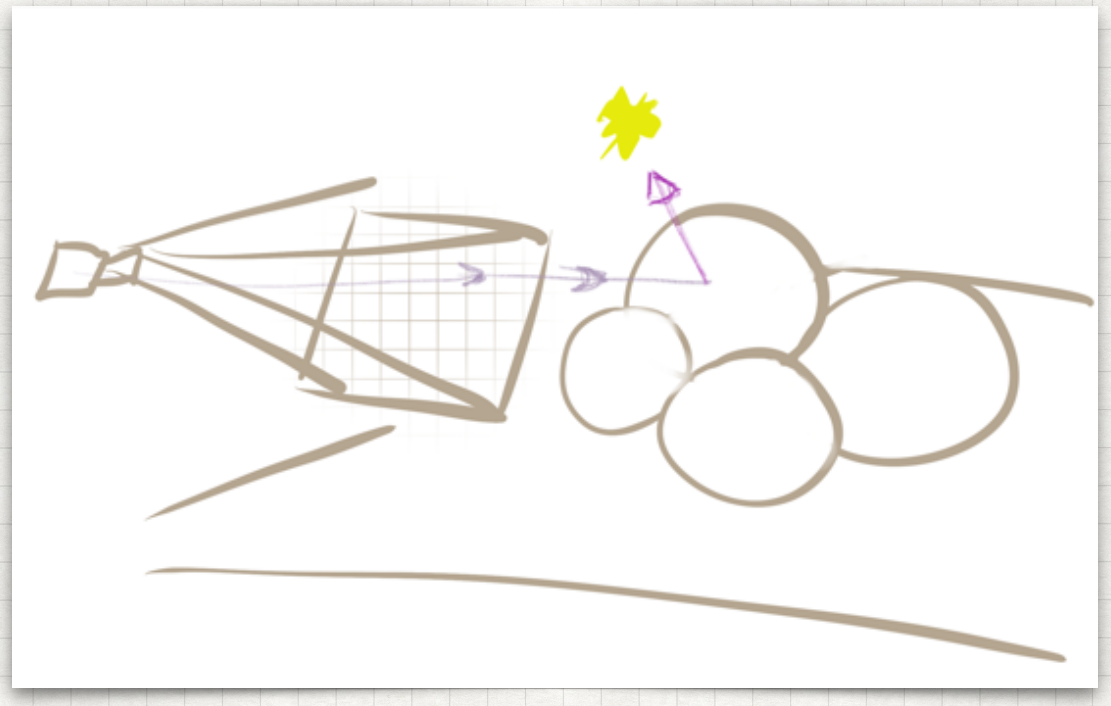
- How Ray Tracing works
- Swift Ray Tracer
- Metal Ray Tracer



RAY TRACING

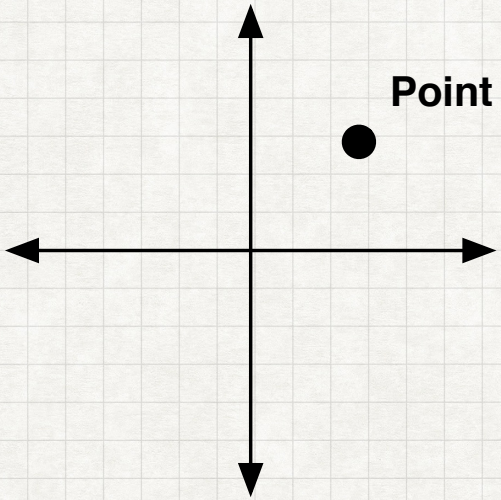
HOW DOES IT WORK?

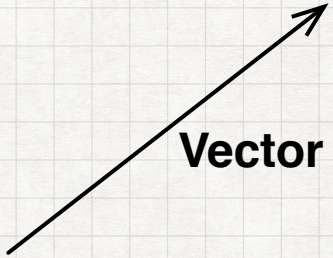
- Generate an image by
  creating a ray from the camera
  to each pixel
  seeing what it hits
- Determine color of what it hits by
  shooting new ray
  from camera ray intersection point
- Ray tracing: work from eye
  Conventional 3D: work from objects

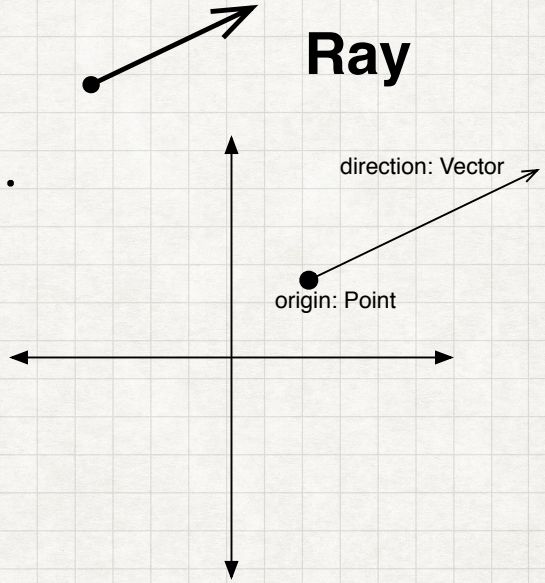


THINGS WE NEED

POINT, VECTOR, RAY

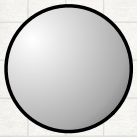




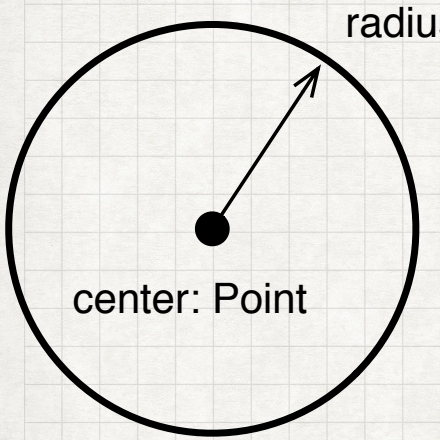


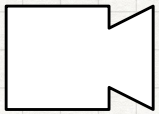
THINGS WE NEED

CAMERA, SPHERE, LIGHT

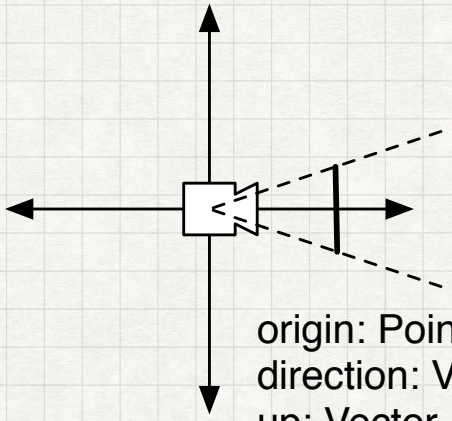


SPHERE

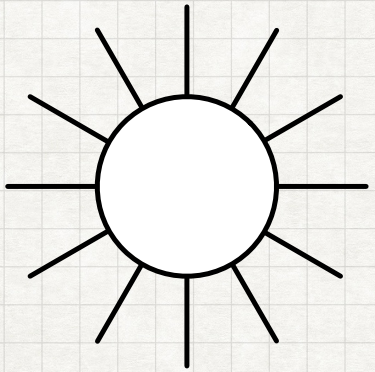




Camera



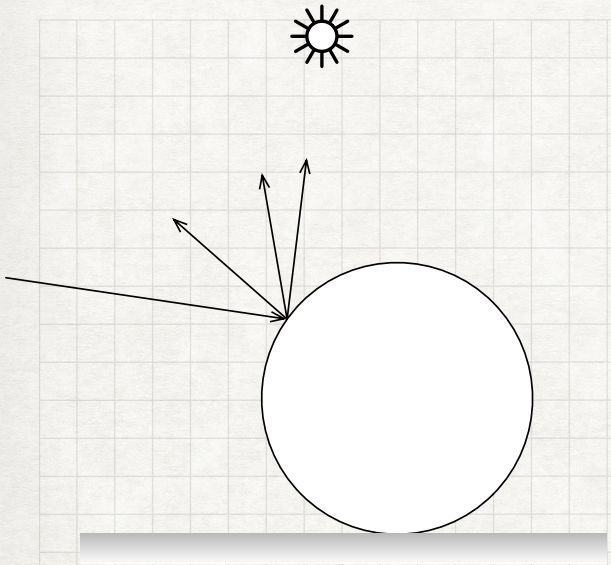
Light



position: Point
color: RGBColor

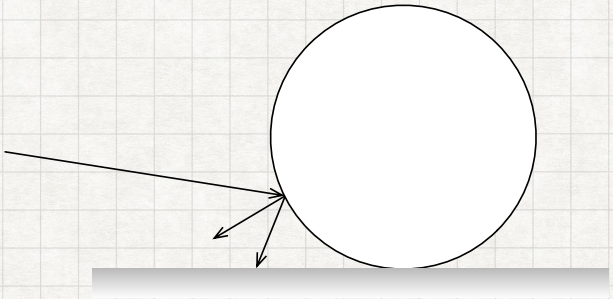
THINGS WE NEED

MATERIALS



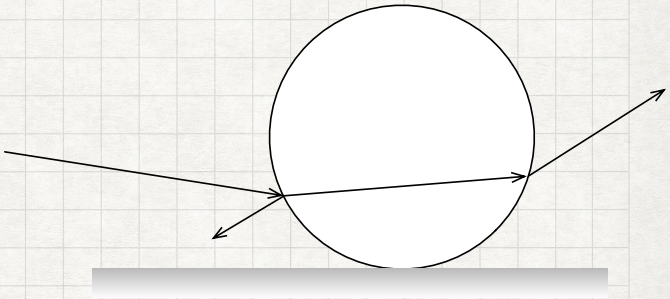
Diffuse

Shade based on color, direction to light



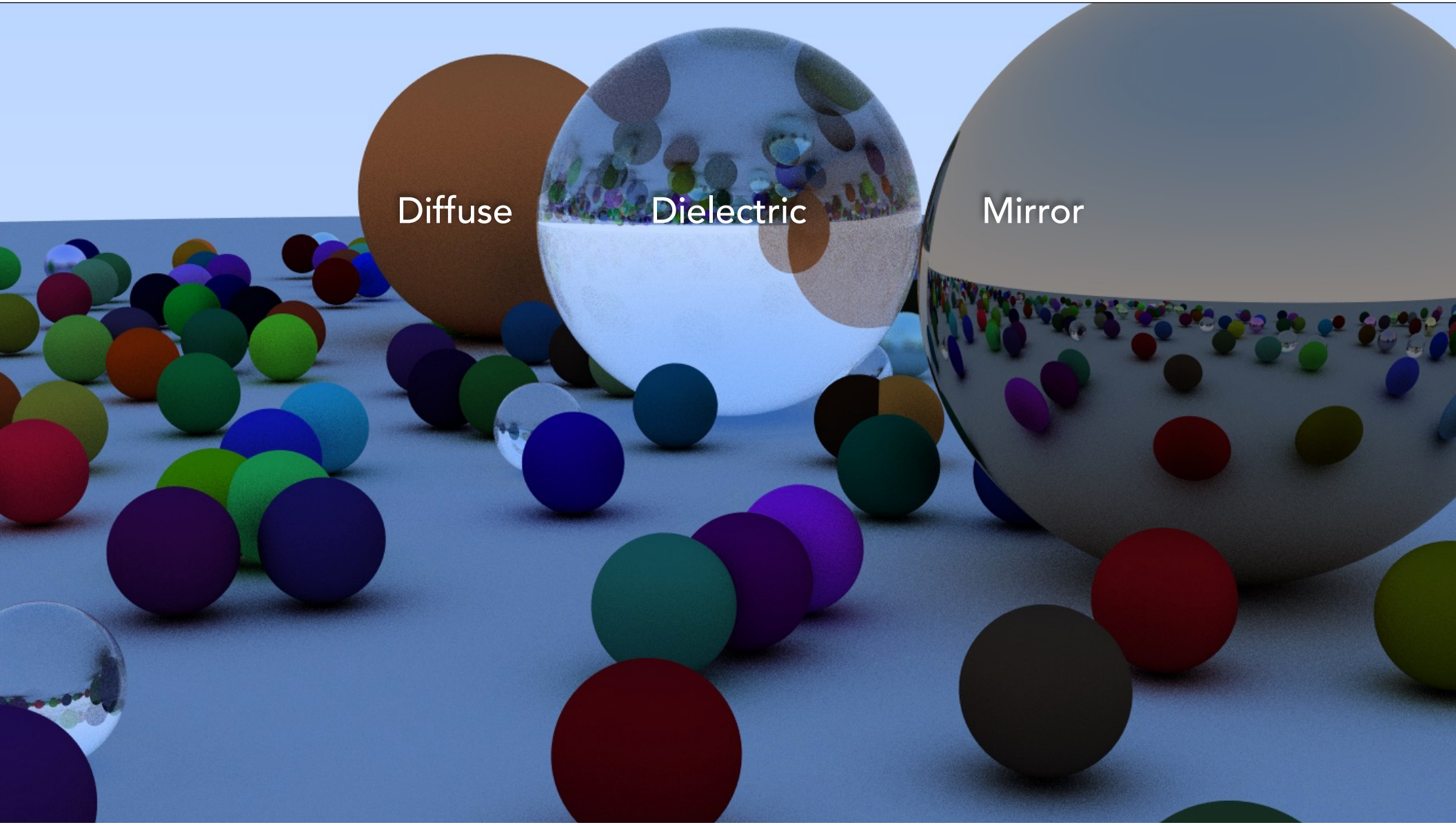
Mirror

Bounce ray, shade based on where it hits
May attenuate



Dielectric

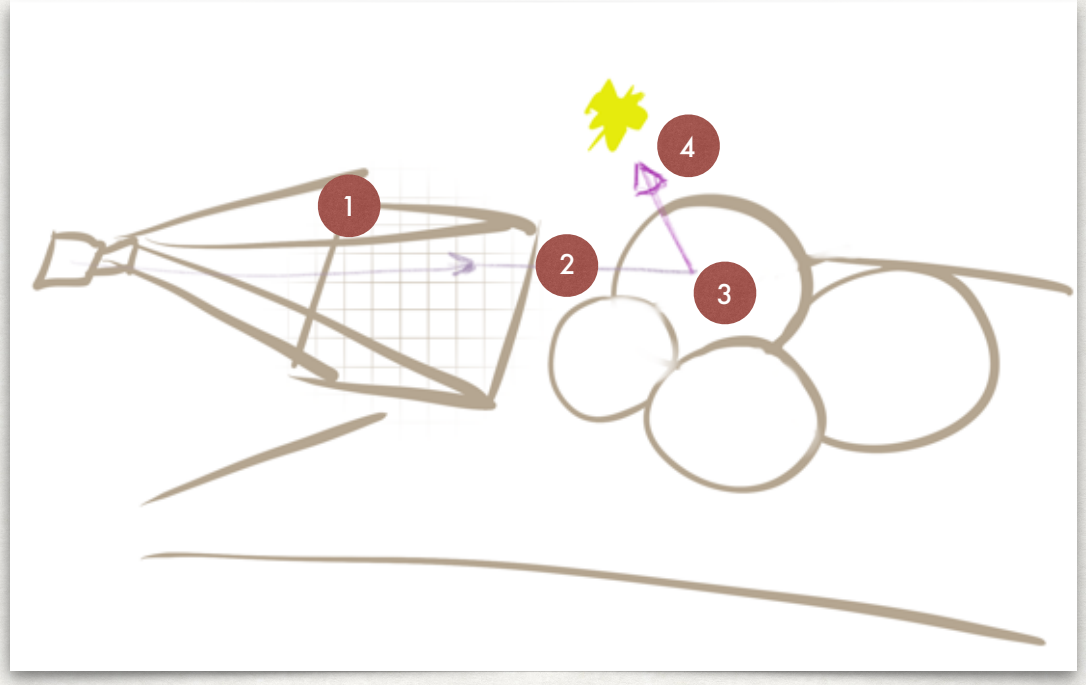
Ray may go through object


Diffuse
Dielectric
Mirror

BASIC ALGORITHM

DONEC QUIS NUNC

1. For all the pixels in the frame
    2. Generate a ray through the screen
    3. Figure out what the ray hits
    4. Figure out the color of surface



Computational Complexity:

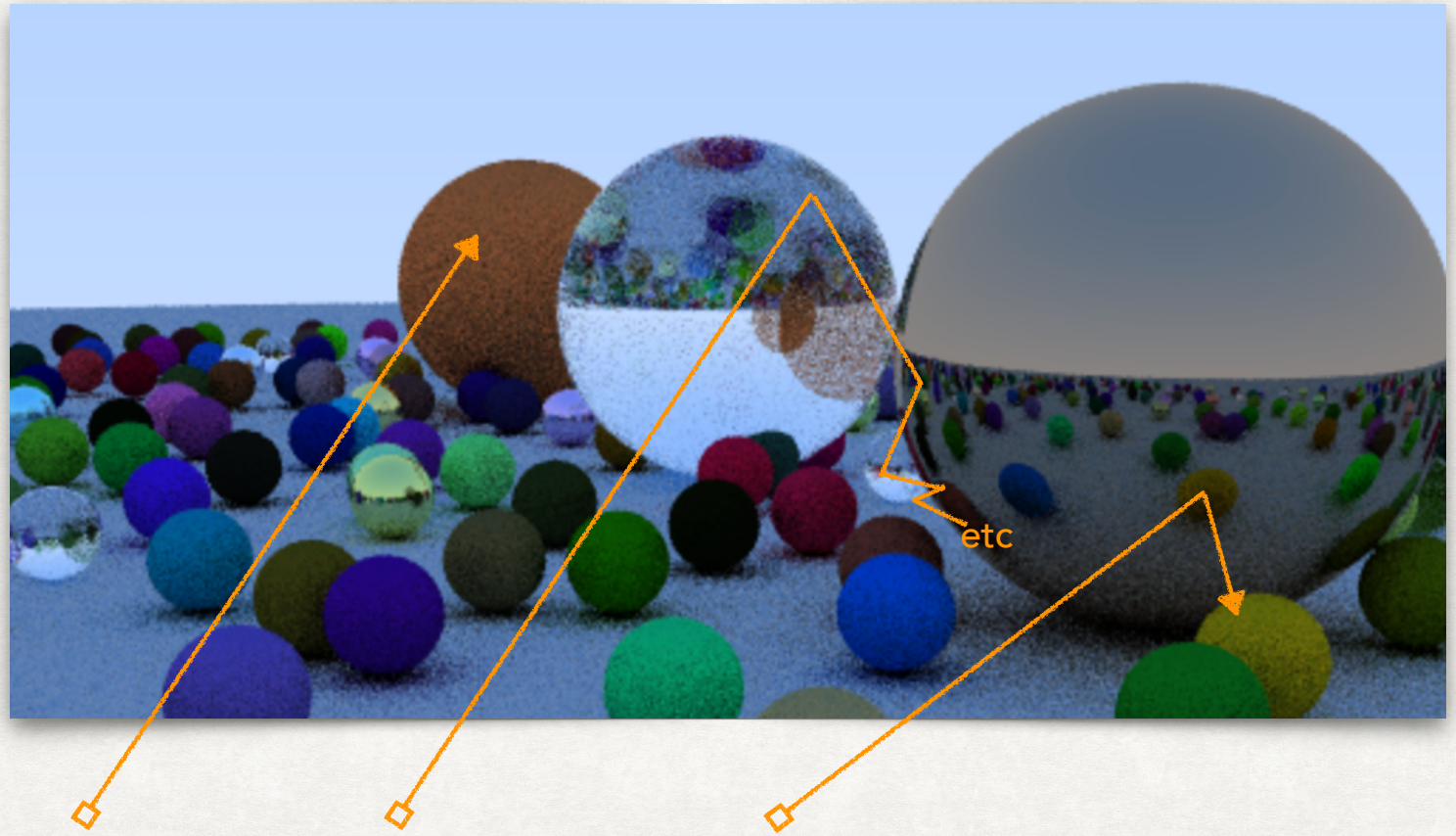
Number of pixels * number of objects

BETTER IMAGES

DONEC QUIS NUNC

- Bounce Depth
- Anti-Aliasing

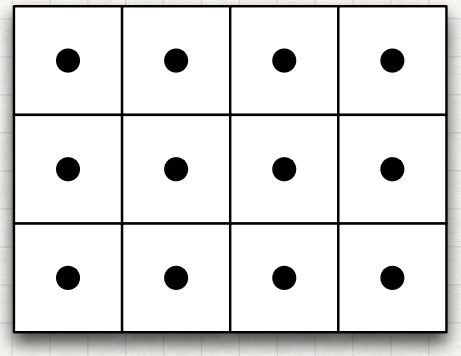
BOUNCE DEPTH

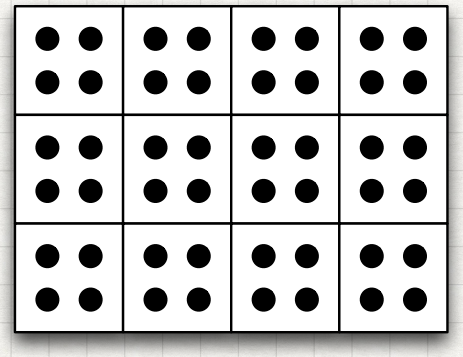


ANTI-ALIASING

SAMPLES PER PIXEL

- One sample per pixel causes jaggies
- Multiple samples smooth out edges

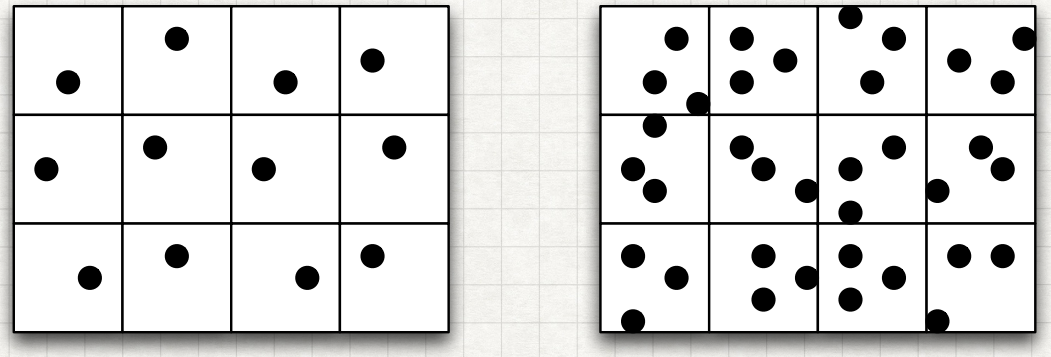




ANTI-ALIASING

JITTERED SAMPLES

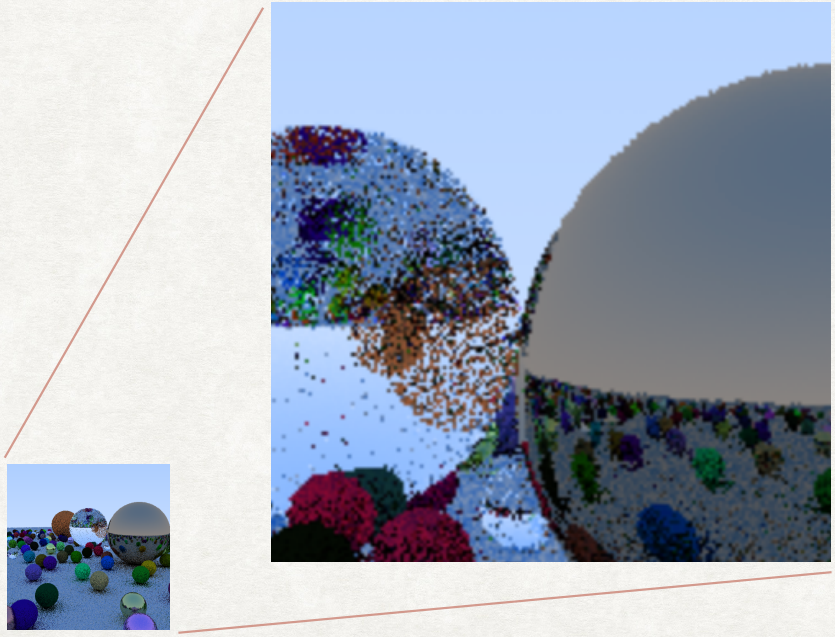
- Avoid regular patterns in objects
- Add random offset to each ray

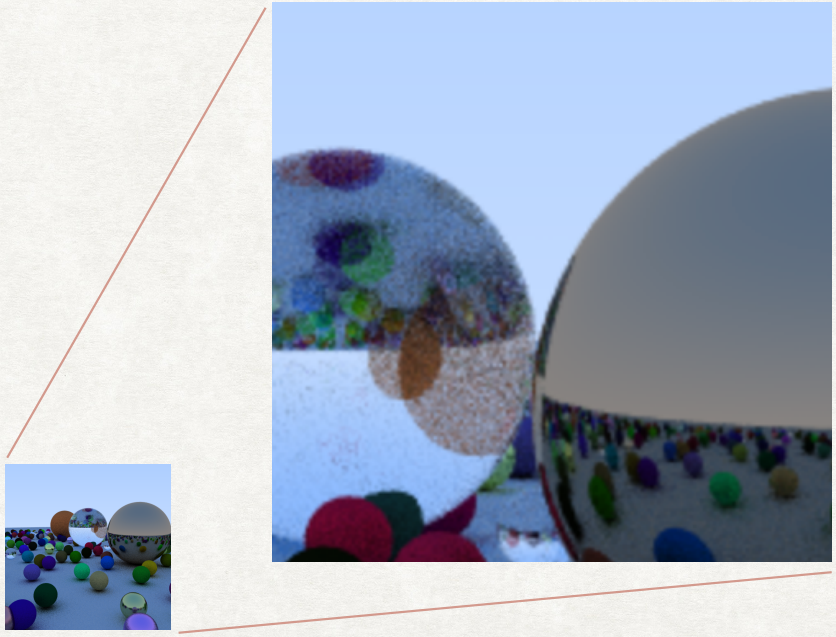


ANTI-ALIASING

RESULTS

- From one sample to 20 samples





A SWIFT RAY TRACER

GOALS

- Try to use value types, protocols
- Functional where appropriate
- Not very concerned with rendering time

- One and a half of them!

GEOMETRIC TOOLBOX

MATH IS FUN

- Point
- Vector
- Rays
- Objects
- Ray / Object Intersection

VECTOR / POINT: ONE OR TWO TYPES?

- Both are just three values: x, y, z
- Can use Swift type safety to make sure your math is right
- Make conversions explicit
- Won't accidentally use ray direction (vector) instead of ray origin (point)

```
public struct Ray {
    let origin: Point
    let direction: Vector

    init(origin: Point, direction: Vector) {
        self.origin = origin
        self.direction = normalize(direction)
    }

    func pointAlong(distance: Double) -> Point {
        return origin + distance * direction
    }
}
```

VECTOR AND POINT STRUCTS

TWO TYPES VERSION

```
public struct Vector {
    let x, y, z: Double

    public init(_ x: Double, _ y: Double, _ z: Double) {
        self.x = x
        self.y = y
        self.z = z
    }

    func length () -> Double {
        let magSquared = x*x + y*y + z*z
        let length = sqrt(magSquared)
        return length
    }
}

public func normalize(v: Vector) -> Vector {
    let vLen = v.length()
    if vLen > 0.0 {
        return (1.0 / vLen) * v
    } else {
        return v
    }
}

public func +(v1: Vector, v2: Vector) -> Vector {
    return Vector(v1.x + v2.x, v1.y + v2.y, v1.z + v2.z)
}

public prefix func -(v1:Vector) -> Vector {
    return Vector(-v1.x, -v1.y, -v1.z)
}

public func -(v1: Vector, v2: Vector) -> Vector {
    return v1 + (-v2)
}

infix operator · { associativity left precedence 150 }
public func ·(v1: Vector, v2: Vector) -> Double {
    return (v1.x * v2.x + v1.y * v2.y + v1.z * v2.z)
}

public func *(a: Double, v: Vector) -> Vector {
    return Vector(a * v.x, a * v.y, a * v.z)
}
```

```
public struct Point {
    let x, y, z: Double

    public init(_ x: Double, _ y: Double, _ z: Double) {
        self.x = x
        self.y = y
        self.z = z
    }

    func relativeTo(origin: Point) -> Point {
        return Point(x - origin.x, y - origin.y, z - origin.z)
    }

    func vectorTo(p: Point) -> Vector {
        return Vector(p.x - x, p.y - y, p.z - z)
    }

    func distanceBetween(p: Point) -> Double {
        return self.vectorTo(p).length()
    }
}

let Origin = Point(0, 0, 0)

public func +(p: Point, v: Vector) -> Point {
    return Point(p.x + v.x, p.y + v.y, p.z + v.z)
}

// a ← b  a₀ a₁ b₂•
infix operator ⟶ { associativity left precedence 150 }
func ⟶(p1: Point, p2: Point) -> Vector {
    return Vector(p2.x - p1.x, p2.y - p1.y, p2.z - p1.z)
}
```

SWIFT + SIMD

DONEC QUIS NUNC

- Expressiveness of Swift,
  Speed of SIMD library
- Implements operators, vector operations

```
import simd

public typealias Vector = float3
```

RAY - OBJECT INTERSECTION

- Optionals shine!

```
public override func hitTest(passingRay: Ray, limits: FloatRange) -> HitResponse? {
…
}
```

- Easy to keep track of intersections

```
public func castRay(r: Ray) -> RaySurfaceIntersection? {

    var closestIntersection: RaySurfaceIntersection? = nil
    for object in objects {
        if let currentIntersection = object.intersect(r) {
            if closestIntersection == nil {
                closestIntersection = currentIntersection
            } else {
                if currentIntersection.closerThan(closestIntersection!) {
                    closestIntersection = currentIntersection
                }
            }
        }
    }

    return closestIntersection
}
```

WITHOUT OPTIONALS

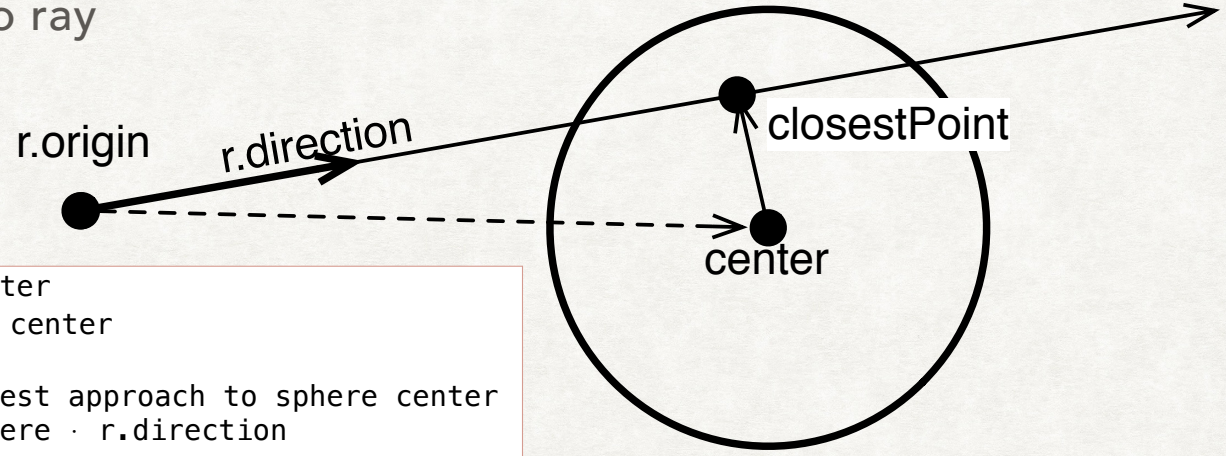
- C++ implementation from Ray Tracing in One Weekend

```
#ifndef HITABLELISTH
#define HITABLELISTH

#include "hitable.h"

class hitable_list: public hitable  {
    public:
        hitable_list() {}
        hitable_list(hitable **l, int n) {list = l; list_size = n; }
        virtual bool hit(const ray& r, float tmin, float tmax, hit_record& rec) const;
        hitable **list;
        int list_size;
};

bool hitable_list::hit(const ray& r, float t_min, float t_max, hit_record& rec) const {
        hit_record temp_rec;
        bool hit_anything = false;
        double closest_so_far = t_max;
        for (int i = 0; i < list_size; i++) {
            if (list[i]->hit(r, t_min, closest_so_far, temp_rec)) {
                hit_anything = true;
                closest_so_far = temp_rec.t;
                rec = temp_rec;
            }
        }
        return hit_anything;
}

#endif
```

RAY-SPHERE INTERSECTION

ONE WAY: 1/3

- Project vector to sphere onto ray direction



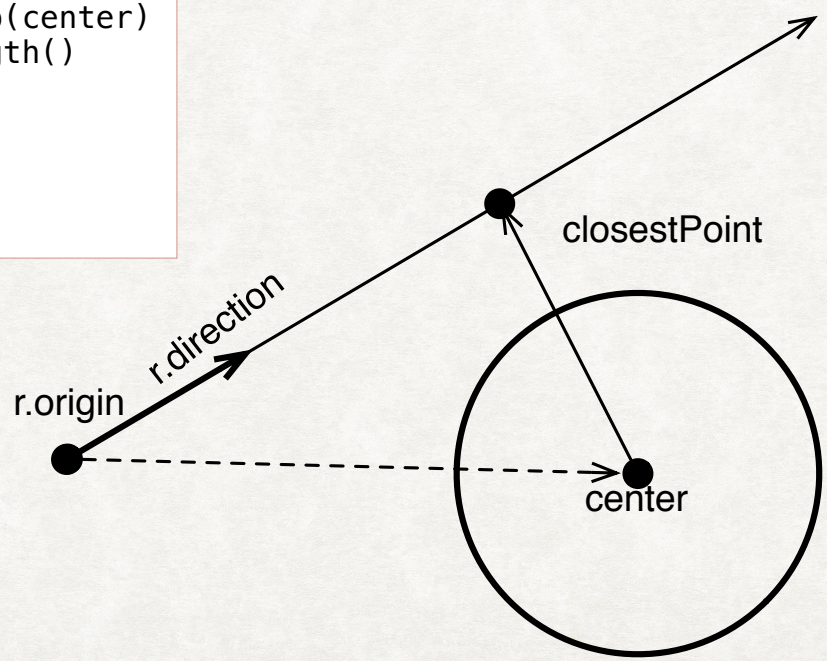

```
// Get origin relative to sphere center
let rayOriginToSphere = r.origin ⟶ center

// Calculate point along ray of closest approach to sphere center
let closestDistance = rayOriginToSphere · r.direction
if (closestDistance < 0) {
    // Sphere is behind the ray, no intersection
    return nil
}
let closestPoint = r.pointAlong(closestDistance)
```

RAY-SPHERE INTERSECTION

ONE WAY: 2/3

- Reject rays outside of sphere

```
// Calculate distance of closest approach to center
let closestPointToCenter = closestPoint.vectorTo(center)
let distanceToCenter = closestPointToCenter.length()
if (distanceToCenter > radius) {
    // Sphere does not hit the ray
    return nil
}
```

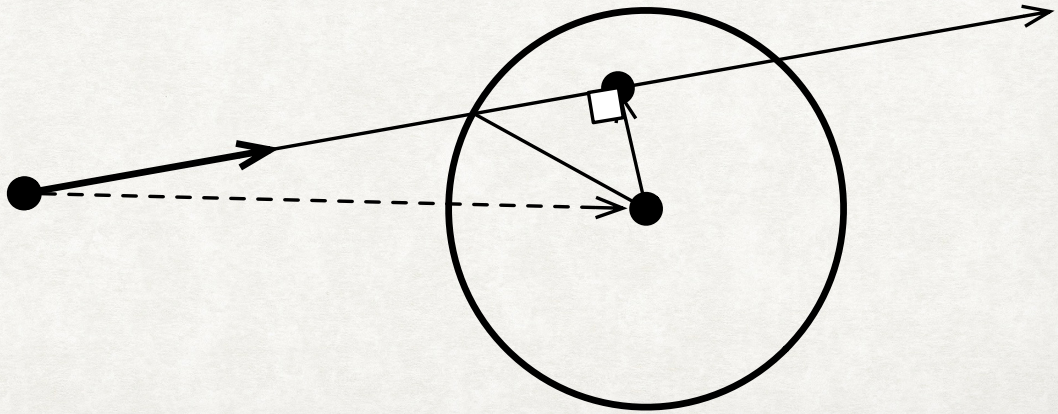


RAY-SPHERE INTERSECTION

ONE WAY: 3/3

- Find point on sphere using Pythagorean Theorem

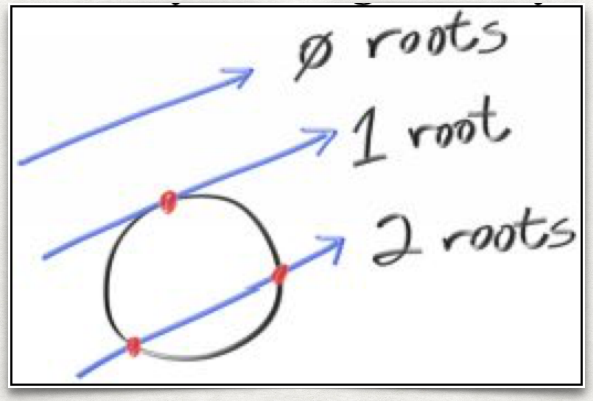
```
// Ray intersects sphere, figure out where
// Calculate distance between intersection and closest approach
let distanceToIntersection = sqrt(radius*radius - distanceToCenter*distanceToCenter)
let directionSign = (distanceToIntersection > closestDistance) ? -1.0 : 1.0

let intersectionDistance = closestDistance - directionSign * distanceToIntersection
let intersectionPoint = RayPoint(r, intersectionDistance)
```



RAY - SPHERE INTERSECTION

ANOTHER WAY

- Equation of ray:
  p = ray.origin + t * ray.direction
- Equation of sphere:
  (x-cx)^2 + (y-cy)^2 + (z-cz)^2 = R^2
- Solve simultaneous equations




```
let centerToRayOrigin = passingRay.origin - center

let c = vector_length_squared(centerToRayOrigin) - radius * radius

let rayOriginWithinSphere = 0 < c
let rayDirectionTowardSphere = 0 < dot(passingRay.direction, centerToRayOrigin)

guard rayOriginWithinSphere || rayDirectionTowardSphere else { return nil }

let a = vector_length_squared(passingRay.direction)
let b = dot(centerToRayOrigin, passingRay.direction)

let discriminant = b * b - a * c

guard 0 <= discriminant else { return nil }

let partA = -1 * b / a
let partB = sqrt(discriminant) / a
var solution = partA - partB

if !limits.contains(solution) {
   solution = partA + partB
}

guard limits.contains(solution) else { return nil }

let surfaceIntersection = passingRay.origin + passingRay.direction * solution
```

UNICODE FUN!

LET 🙄=TRUE

- a ⟶ b
  Vector from two points
- a · b
  Dot product
- 180°
  Degrees to radians
- let π = Float(M_PI)

- In variable names

```
let v‖ = dot(direction, upVector) * upVector
let v⊥ = direction - v‖
θ = atanf(v⊥.x/v⊥.z)

φ = acosf( length(v⊥) / length(direction) )
v™ = normalize(cross(v⊥, direction))
φRotationRequired = 0.0000001 < abs(φ)
```

ENUMS AND PROTOCOLS

- Hittable
- HitResponse
- Material

```
public class Hittable {

  public var material: Material = .lambertian

  func hitTest(passingRay: Ray, limits: FloatRange) -> HitResponse? {
      return nil
  }

}
```

```
public enum HitResponse {
   case bounce(distance: Float, ray: Ray, colorReflectivity: Color)
   case color(distance: Float, value: Color)
}
```

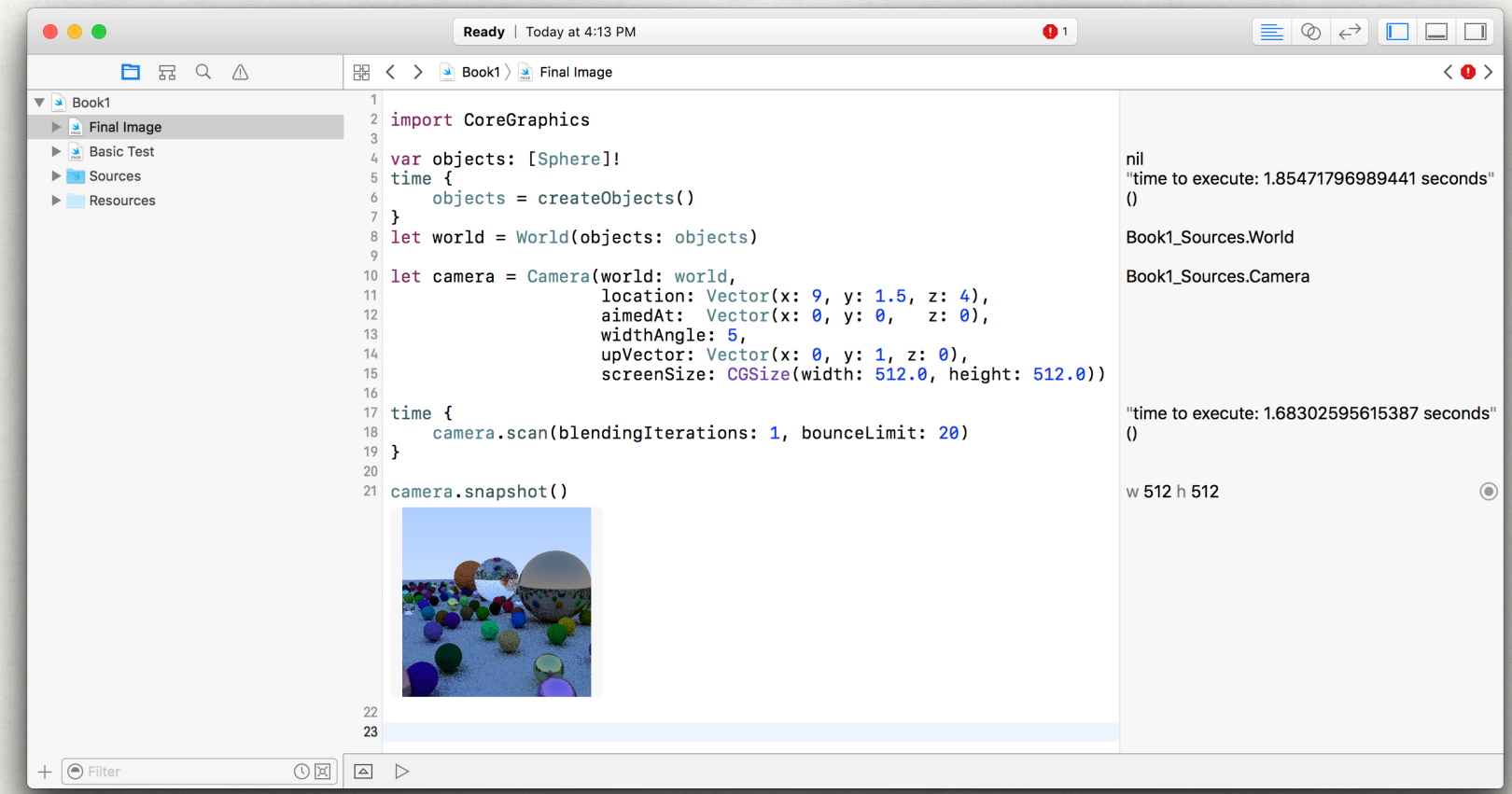
```
public enum Material {
  case lambertian
  case metal(fuzz: Float)
  case dialectric(indexOfRefraction: Float)

  func lightResponse(passingRay: Ray, normal: Vector,
        surfaceIntersection: Vector, colorReflectivity: Color,
        randomInteriorPoint: Vector) -> HitResponse?
```

RENDERING LOOP DESIGN

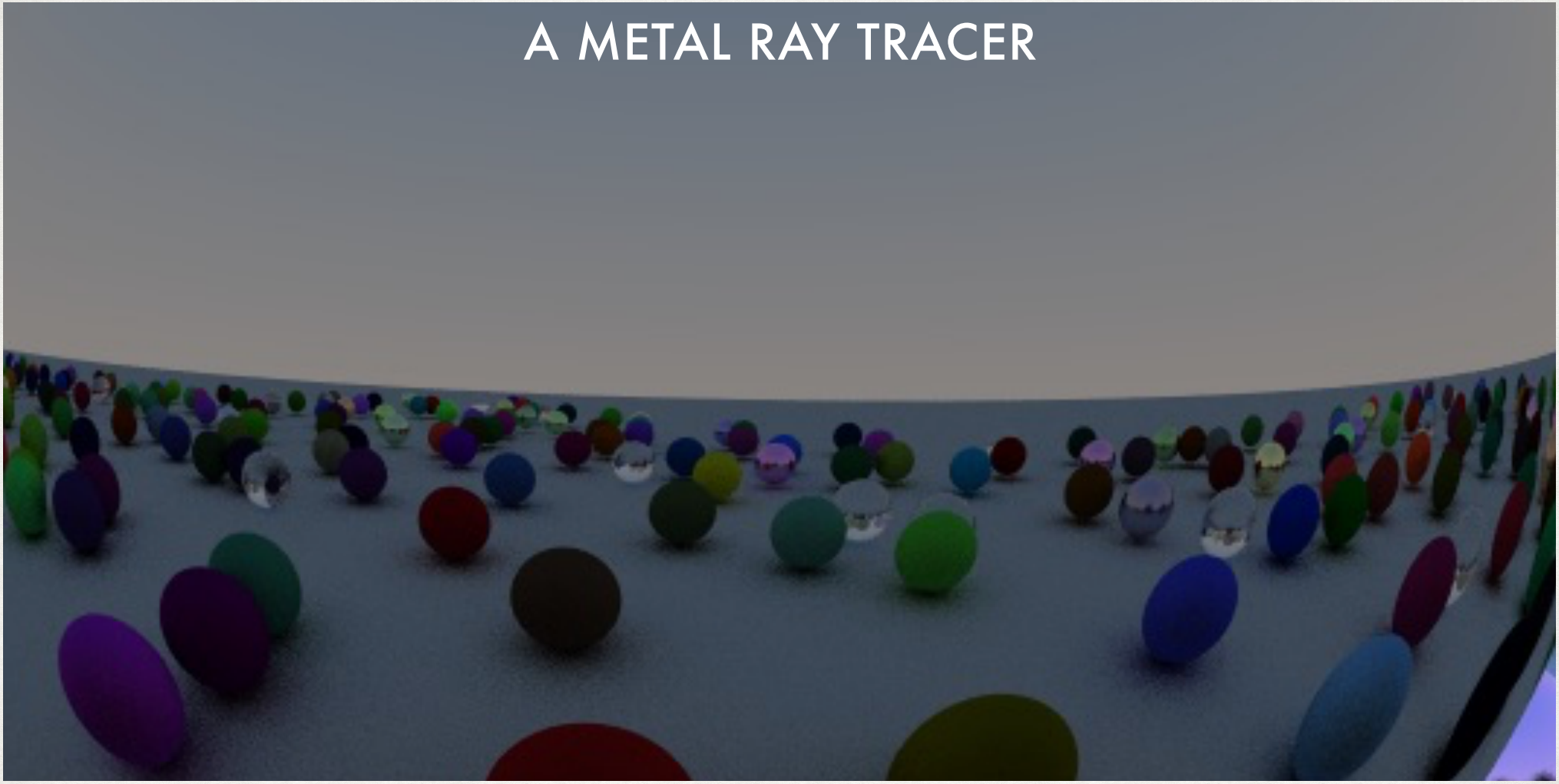
- Cast rays
- Light results

```
func lightResponse(…)
      -> HitResponse?
{

let length = distance(passingRay.origin, surfaceIntersection)
let reflected = vector_reflect(passingRay.direction, normal)

switch self {

case .lambertian:
   let nextDirection = normal + randomInteriorPoint

   return .bounce(distance: length,
                  ray: Ray(origin: surfaceIntersection, direction: nextDirection),
                  colorReflectivity: colorReflectivity)

case let .metal(fuzz):
   let nextDirection = reflected + min(fuzz, 1) * randomInteriorPoint

   guard 0 < dot(nextDirection, normal) else { return nil }

   return .bounce(distance: length,
              ray: Ray(origin: surfaceIntersection, direction: nextDirection),
              colorReflectivity: colorReflectivity)

case let .dialectric(indexOfRefraction):
   let attenuation = Color.white()
   …

   return .bounce(distance: length,
                  ray: Ray(origin: surfaceIntersection, direction: reflected),
                  colorReflectivity: attenuation)
}
}
```

ALL IN A PLAYGROUND

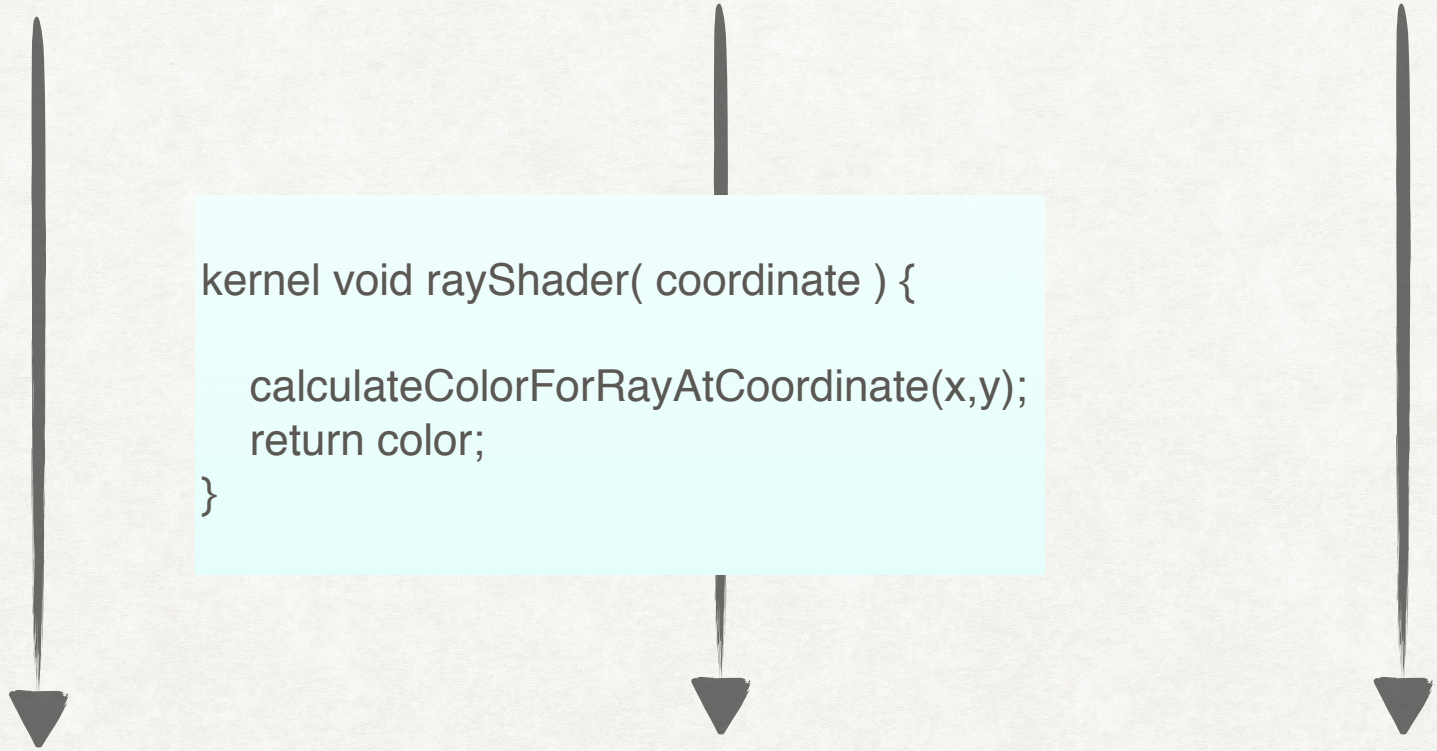


A METAL RAY TRACER

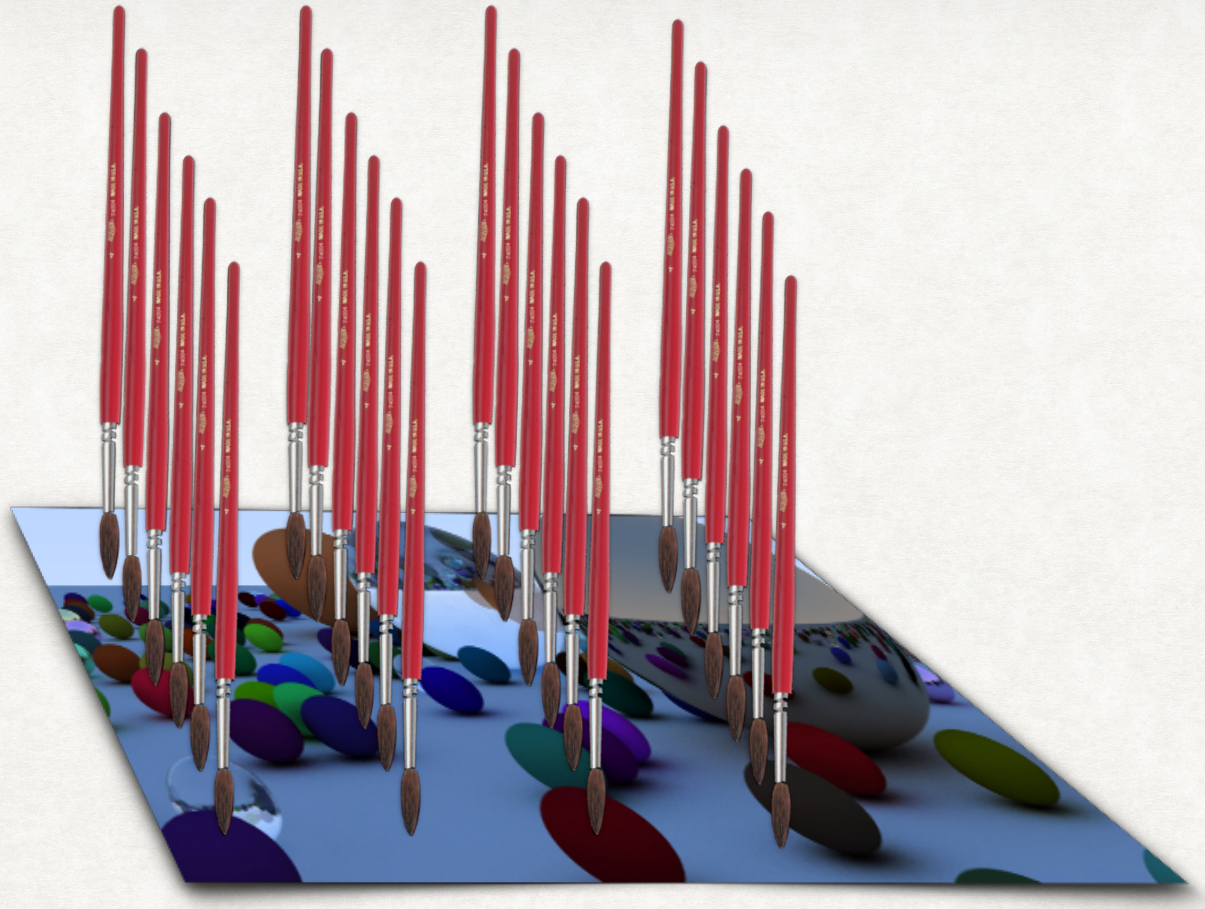


IDEA

- Using GPUs can hold out the hope of big speed gains
- (Perhaps hardest part) What is parallel about your problem?
- Set up shop in Metal to do each individual calculation
- Metal is a variant of C++-14, with much of the evil stuff stripped out
- Prepare what all Metal threads need in common, and pass that from the host
- Figure out how you need to get the results back



```
kernel void rayShader( coordinate ) {

    calculateColorForRayAtCoordinate(x,y);
    return color;
}
```



```
kernel void rayShader(const device Sphere* objectsIn  [[ buffer(0) ]],
                      const device ViewInfo* viewInfoIn    [[ buffer(1) ]],

                      texture2d<float, access::write> output [[texture(0)]],

                      uint2 upos [[thread_position_in_grid]]) {

// stuff!

    output.write(float4(totalColor.x, totalColor.y, totalColor.z, 1), upos);
}
```

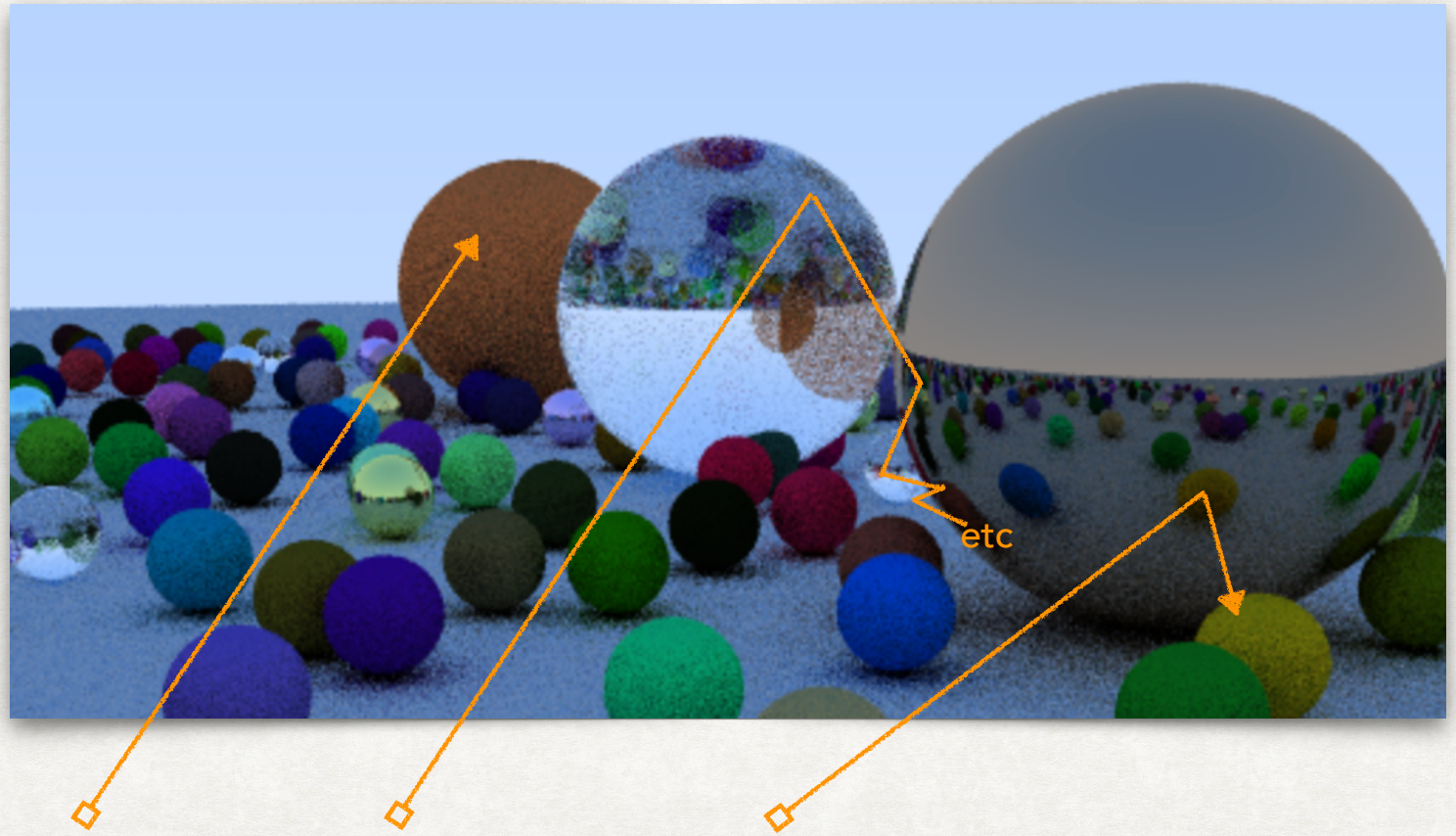
10 COLOR = 6
20 PLOT 12, 7

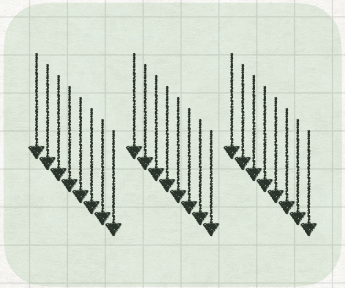
```
public func schlick(cosine: Float, indexOfRefraction: Float) -> Float {
    var r0 = (1 - indexOfRefraction) / (1 + indexOfRefraction)
    r0 *= r0
    return r0 + ((1 - r0) * powf(1 - cosine, 5))
}
```

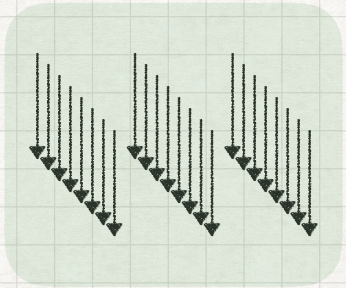
```
static float schlick(float cosine, float indexOfRefraction) {
    float r0 = (1 - indexOfRefraction) / (1 + indexOfRefraction);
    r0 *= r0;
    return r0 + ((1 - r0) * pow(1 - cosine, 5));
}
```

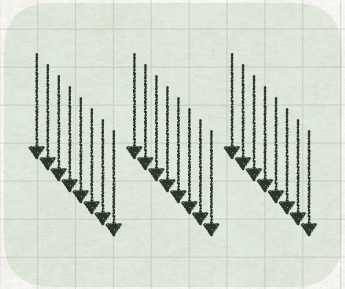
```
struct Sphere {
    var materialType: Int32 = 0
    var color = Color()
    var center = Vector()
    var radius: Float = 0
    var fuzz: Float = 0
    var indexOfRefraction: Float = 1
}
```

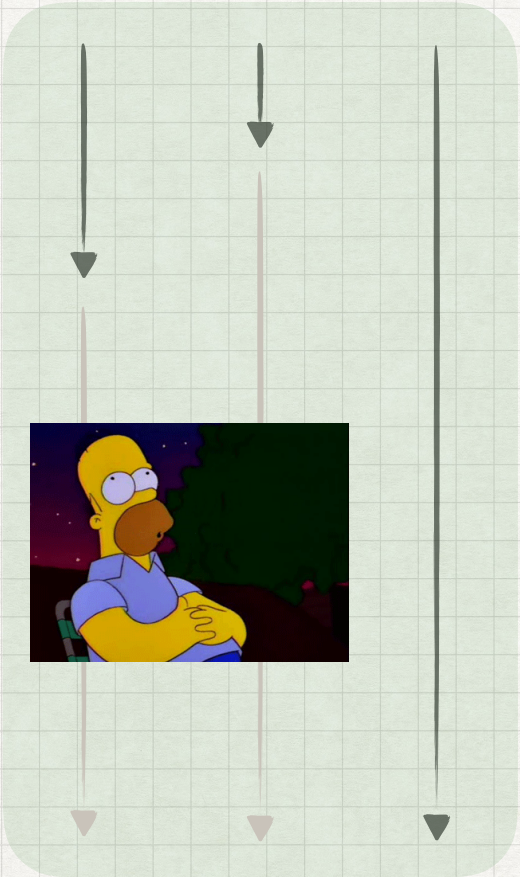
```
enum MaterialType {
    lambertian, metalic, dialectric
};

struct Sphere {
    MaterialType materialType = lambertian;
    float3 color = float3(1);
    float3 center = float3(0);
    float radius = 0;
    float fuzz = 0;
    float indexOfRefraction = 1;
};
```


etc







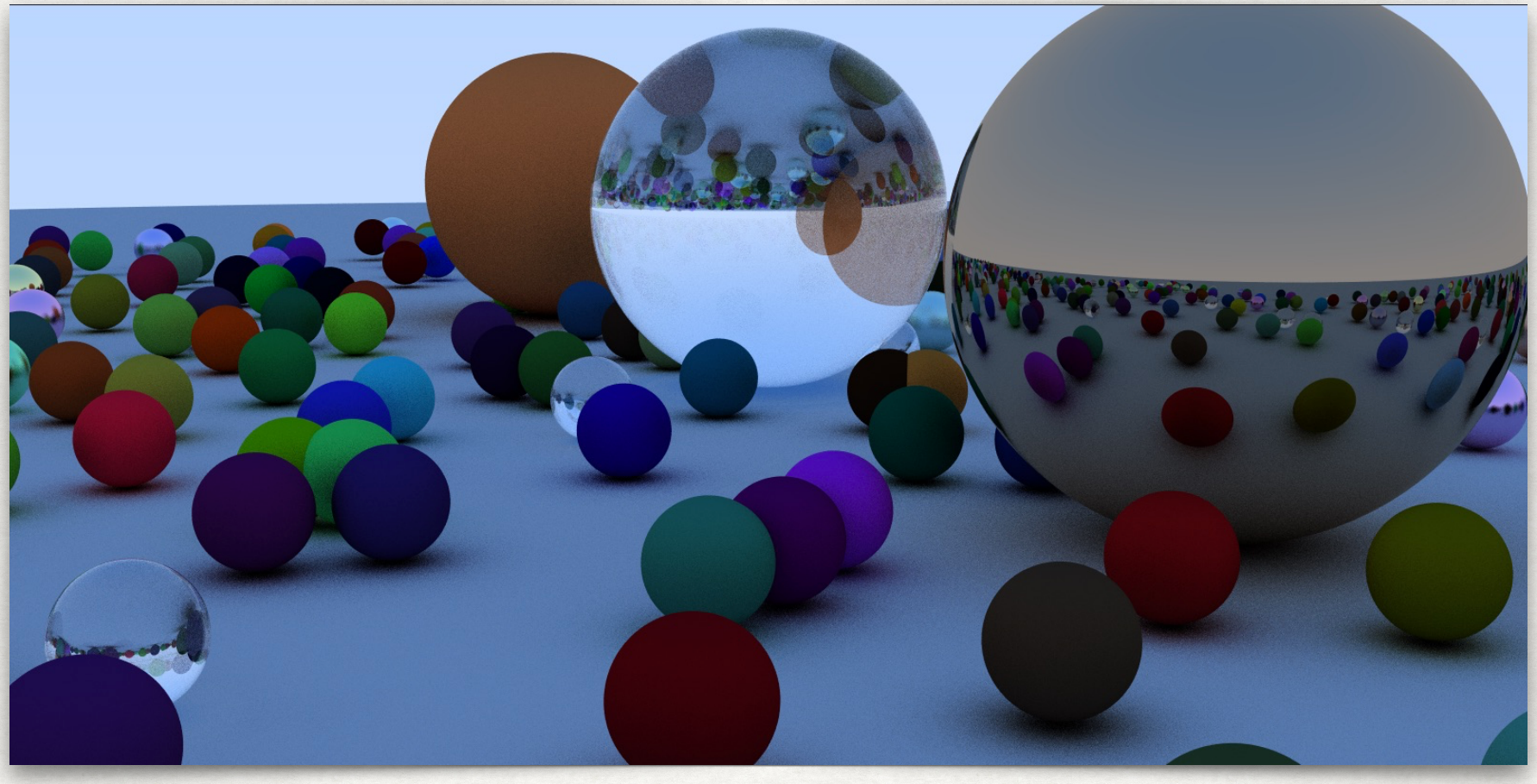


GOTCHAS & TIPS

- Artificial limit your loops on the metal side
- There are no random number generators native to metal
- Byte alignment will matter when passing structs back-n-forth
- There is no subclassing in MSL

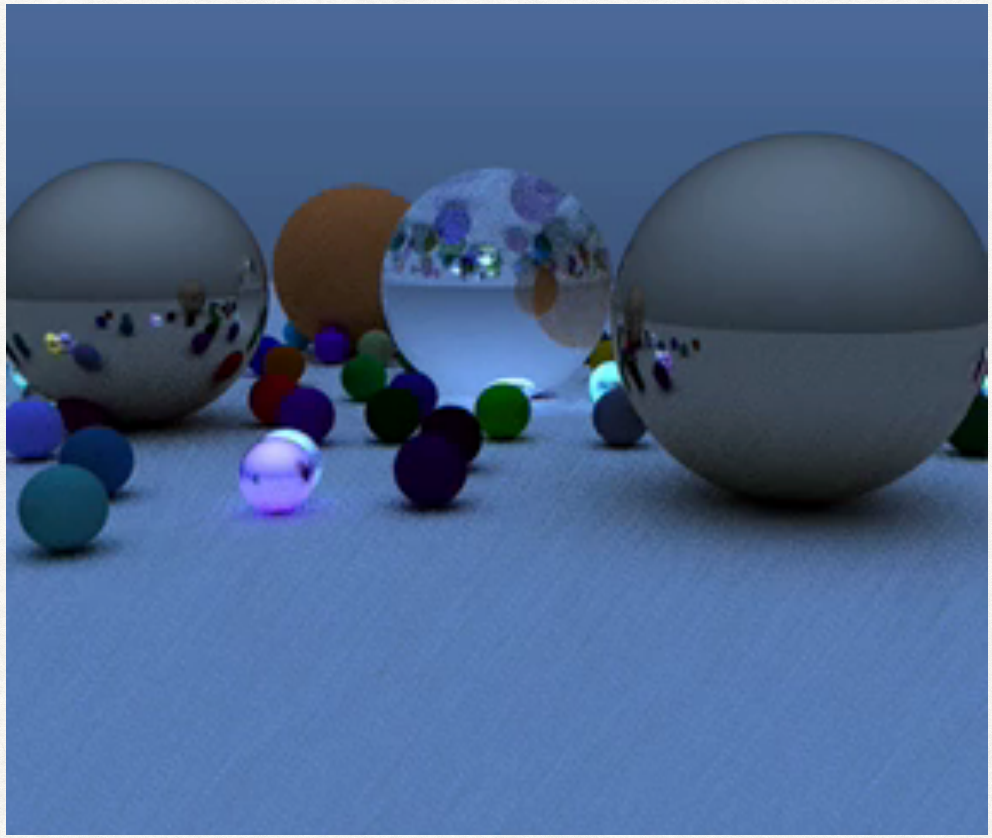
PROTOTYPING IN SWIFT

- Prototype in a real app so you can maximally debug it
- Avoid subclassing, or at least be aware that there is none in MSL
- If you need random numbers, pass them in or find an RNG to use
- Use **simd** framework



2012 MacBook Pro: ~60x

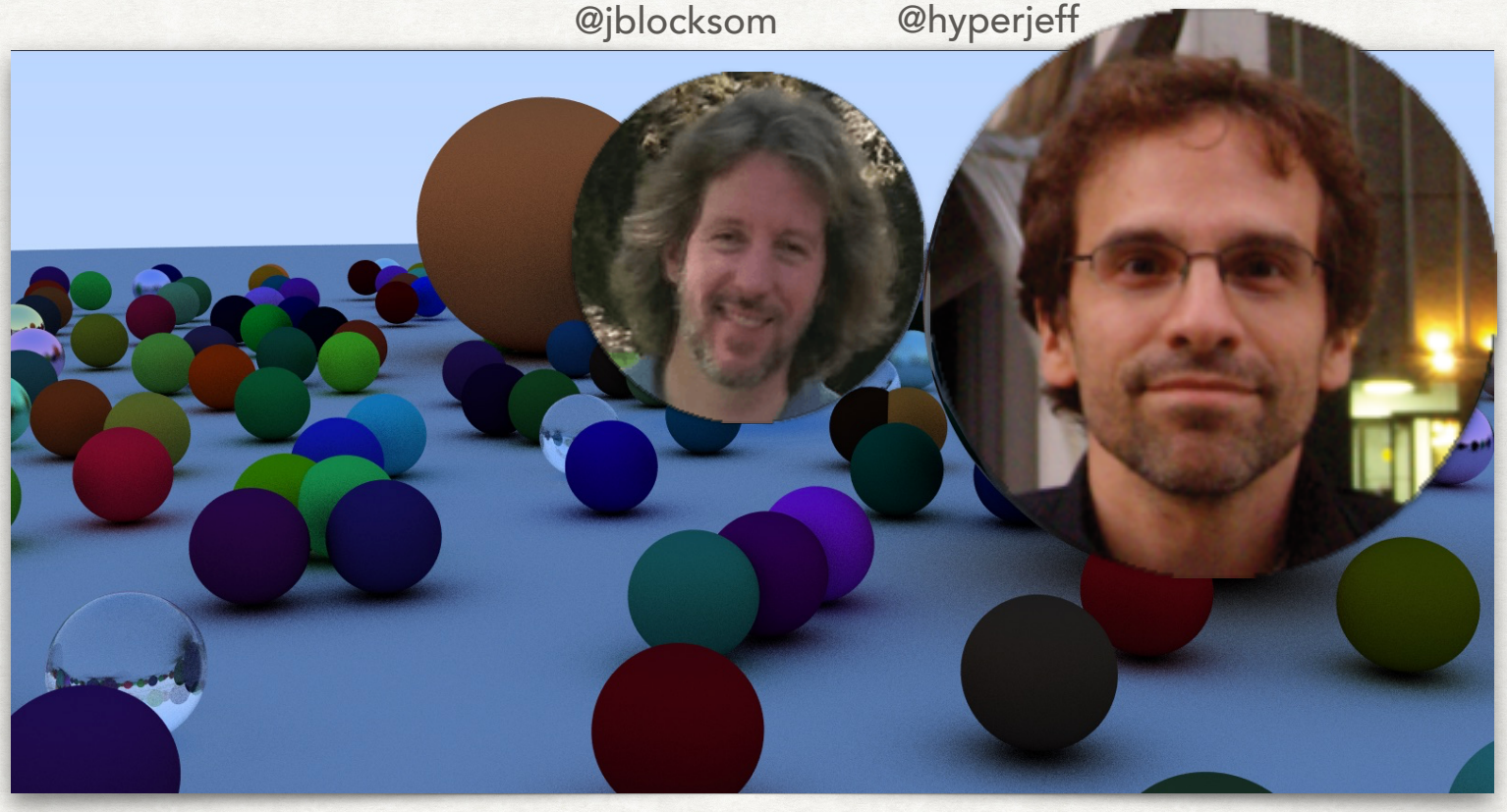
2013 MacPro: ~200x



Thanks!

Jonathan Blocksom @jblocksom

Jeff Biggus @hyperjeff



Code and slides up at:
https://github.com/hyperjeff/360iDev-2016-RayTracing