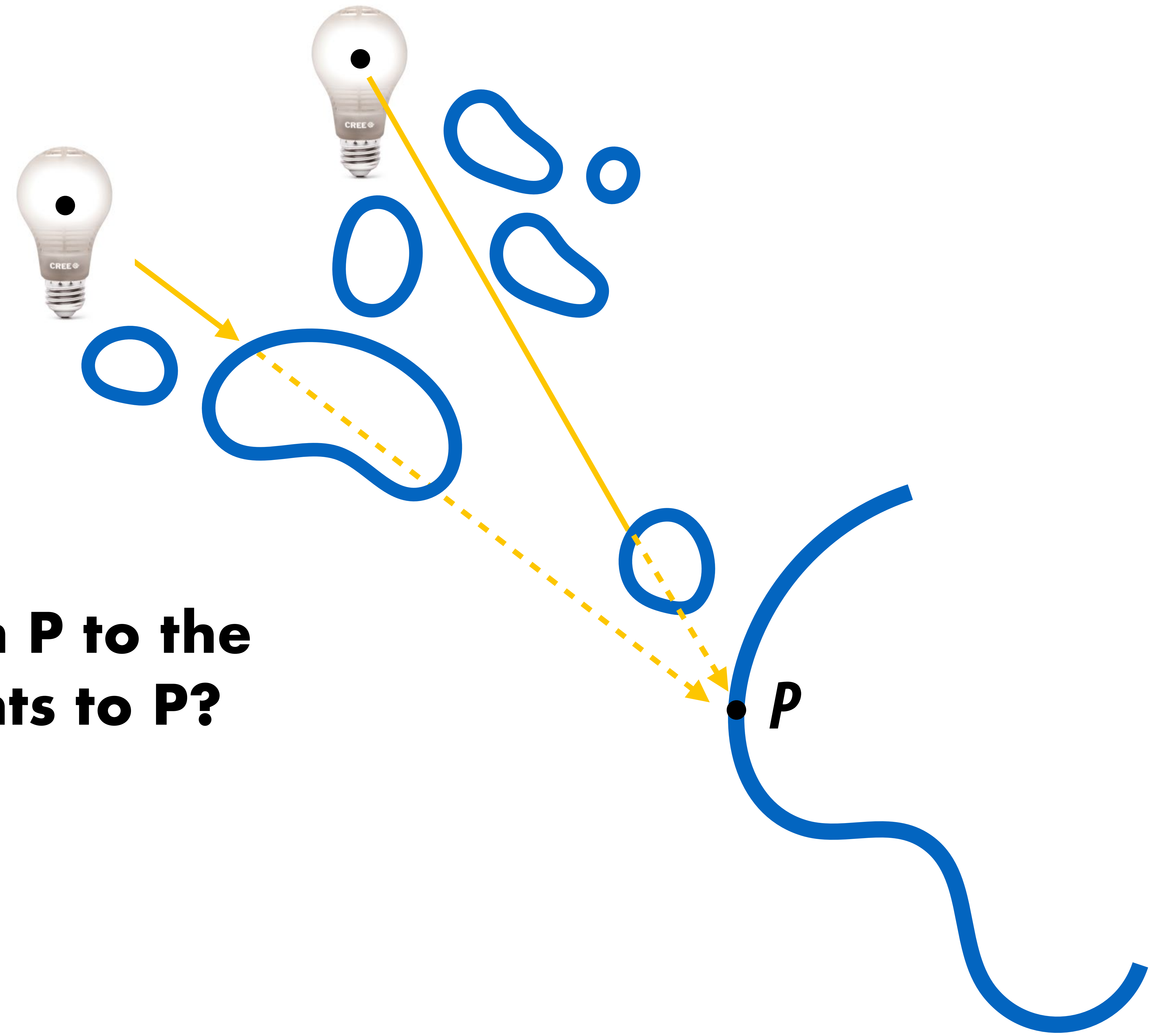
Ray Tracing 2:
Practical Realities

Today

- Discussion of many practical realities of ray tracing
- Building acceleration structures quickly
  - Two-level hierarchies
  - Refitting
  - Incremental builds
- Ray tracing subdivision/tessellated surfaces
  - Cracks
- Geometry instancing

Warm up

Consider tracing shadow rays through a scene to determine if point P is in shadow from the two light sources.

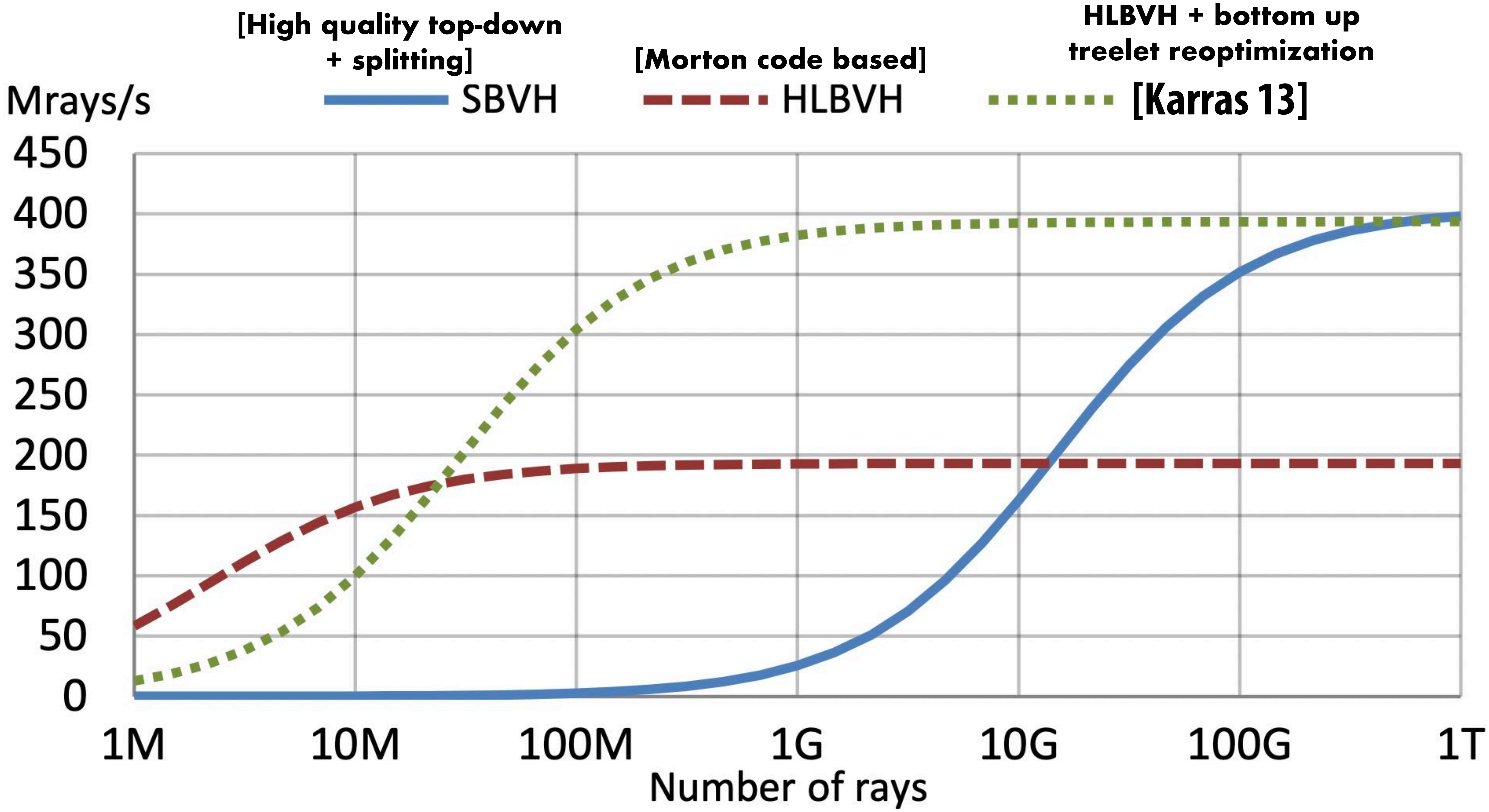


Do you trace rays from P to the lights? Or from the lights to P?

Does it matter?

Can afford to build a better BVH if you are shooting many rays (can amortize cost)

- The graph below plots effective ray throughput (Mrays/sec) as a function of the number of rays traced per BVH build
  - More rays = can amortize costs of BVH build across many ray trace operations



Real time ray tracing demo

Battlefield V (EA/DICE)

Battlefield V (EA/DICE)

Ray Tracing Dynamic Scenes

Challenge #1: scenes have millions of triangles, many objects are in motion

Challenge #2: relatively few rays traced per frame

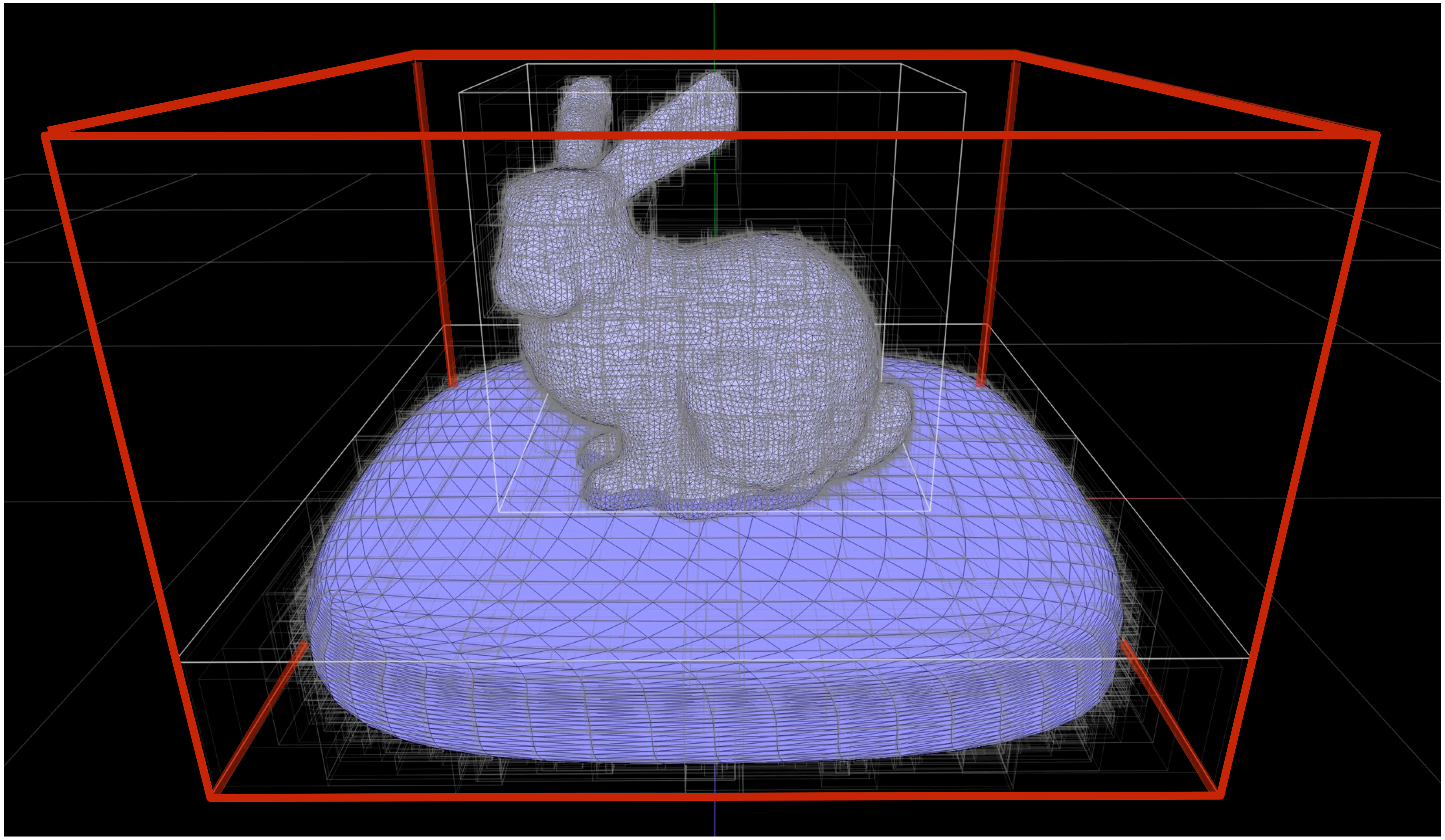
For real time, can allow a few ms / frame

- e.g. @10M tris, 60fps, need 600M tris / second
- pbrt BVH: ~2.5M tris / second

➡ Hierarchy construction efficiency really matters

A BVH is an intersectable primitive

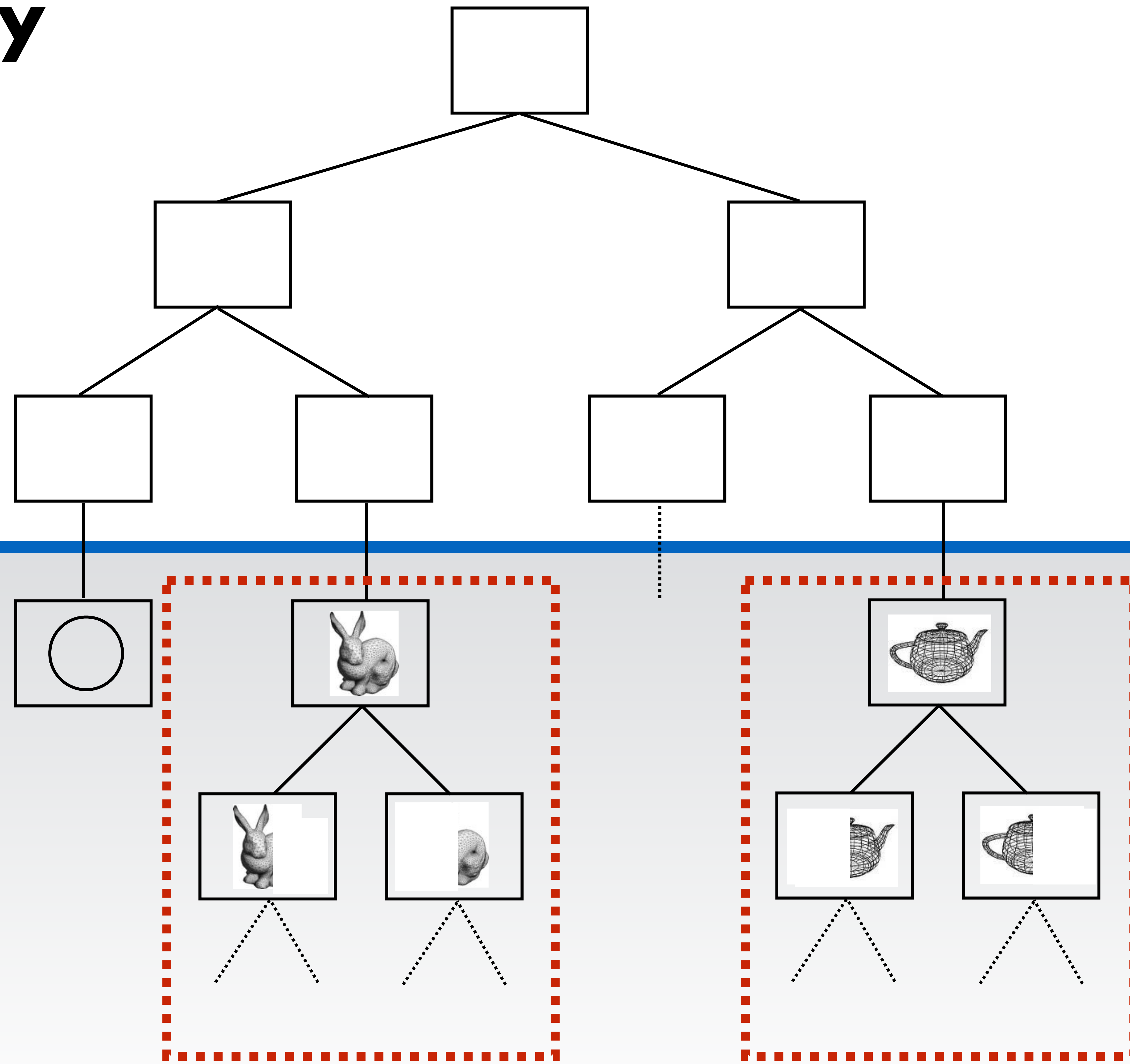
It has a bounding box
It supports ray-primitive intersection
So it can be used as a primitive in another BVH.



Two-level Acceleration Structures

2-level hierarchy

Top-level acceleration structure



Bottom-level acceleration Structures (Primitives in top-level BVH)

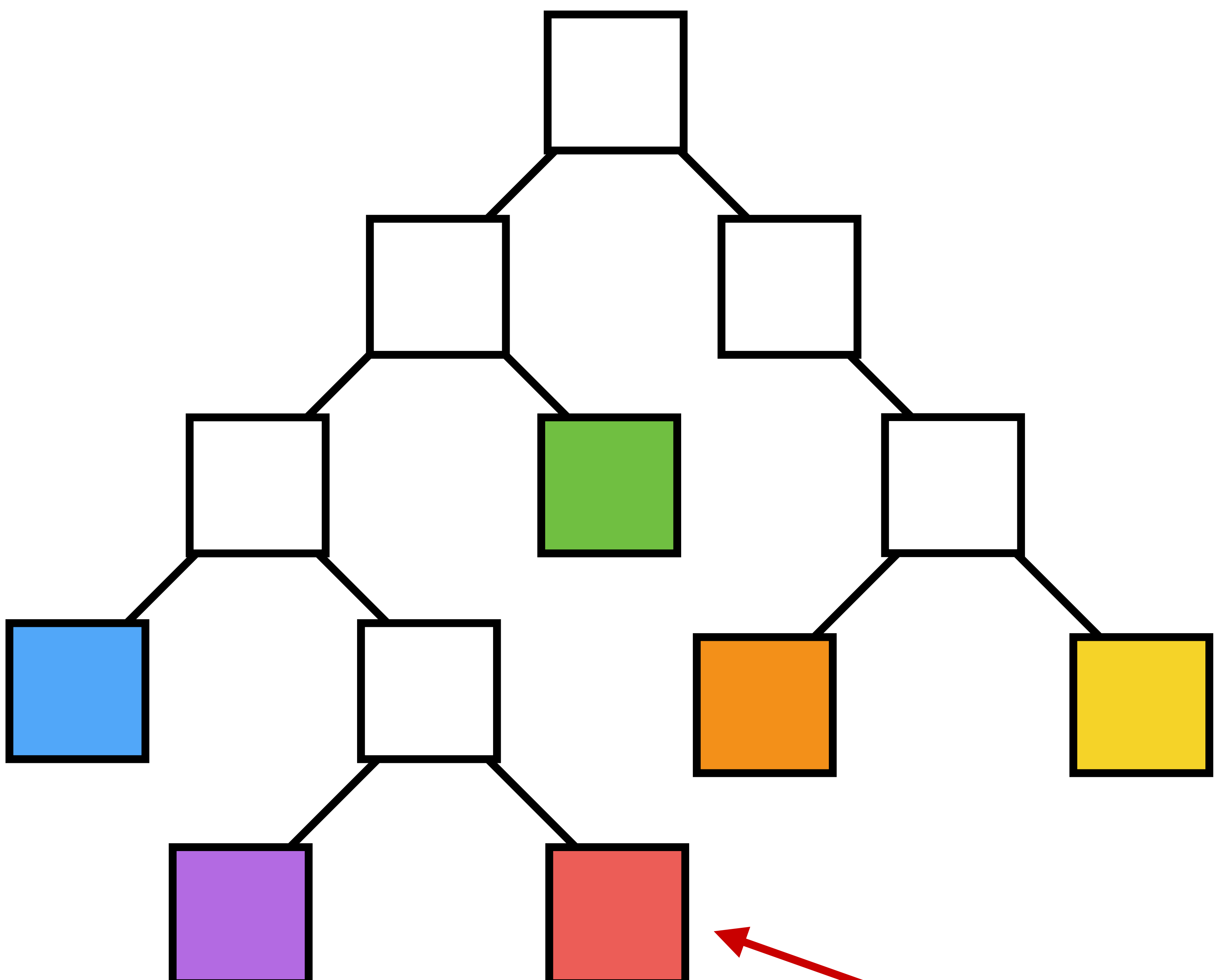
Build BVH for each object in a scene (one time, up front)

Each frame...
Build top-level BVH of BVH's based on current object positions.

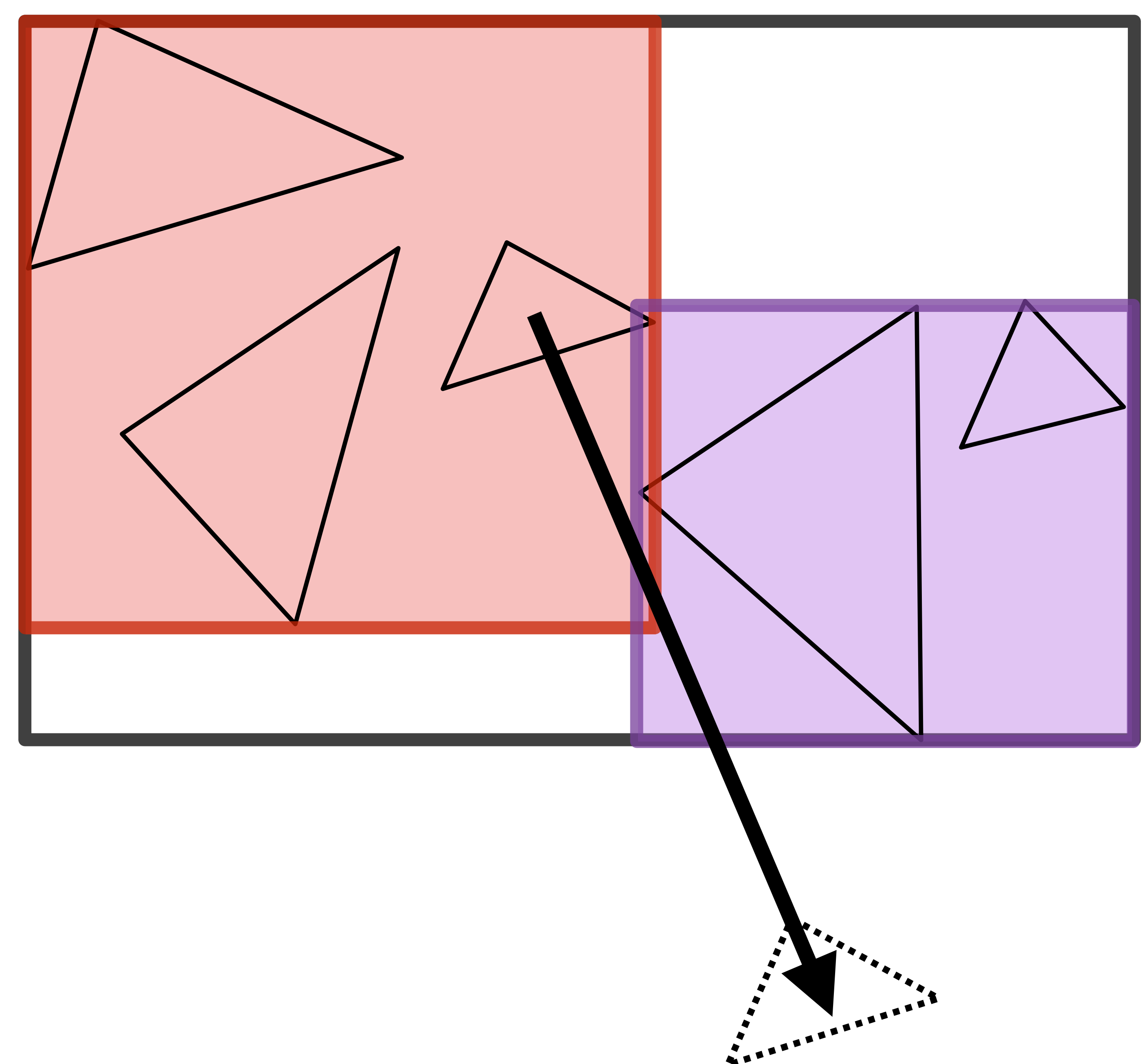
Image credit: Brennan Shacklett

Refit rather than rebuild

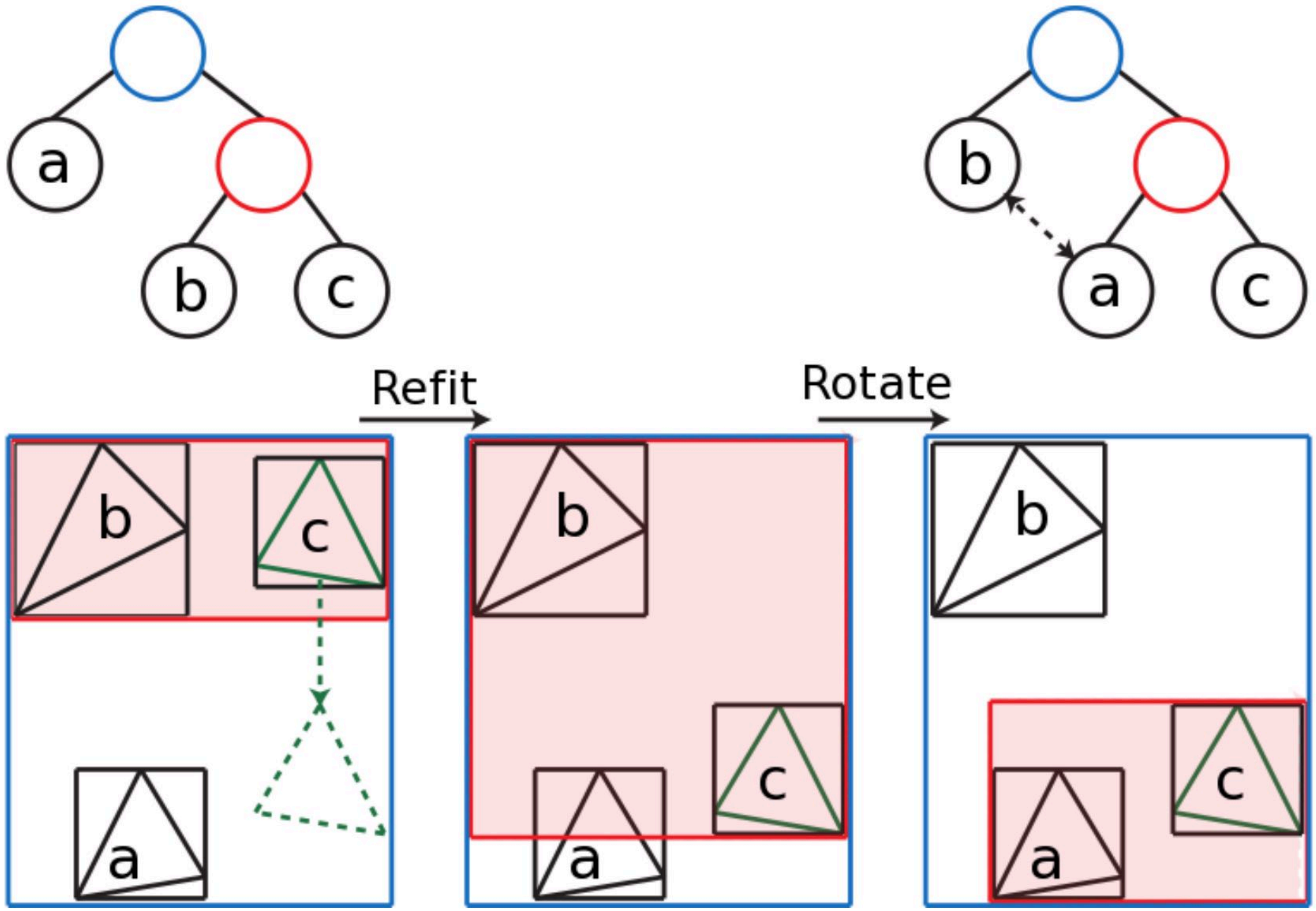
- Imagine you have a valid BVH
- Now move one of the triangles in the scene to a new location
- How do you adjust the BVH's bboxes so it is a valid BVH?



Imagine I moved a triangle in this red leaf node.

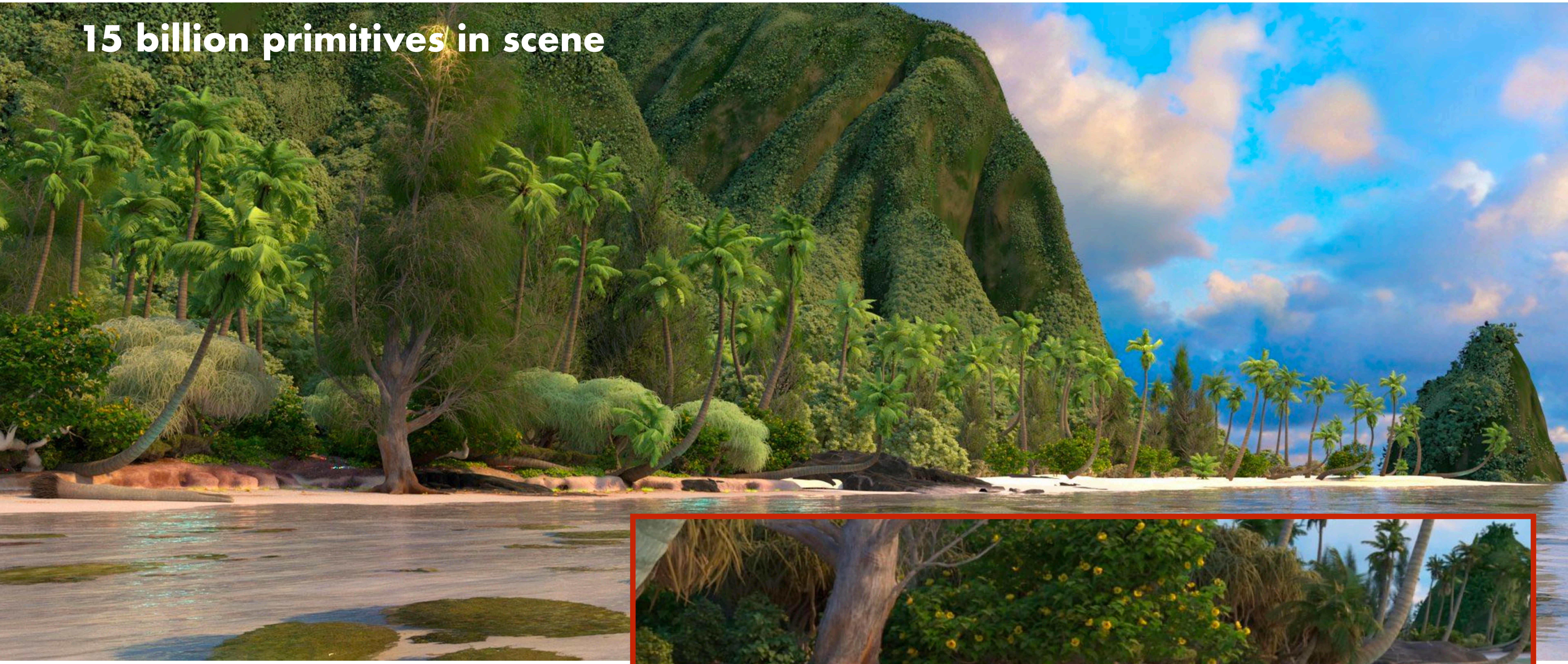


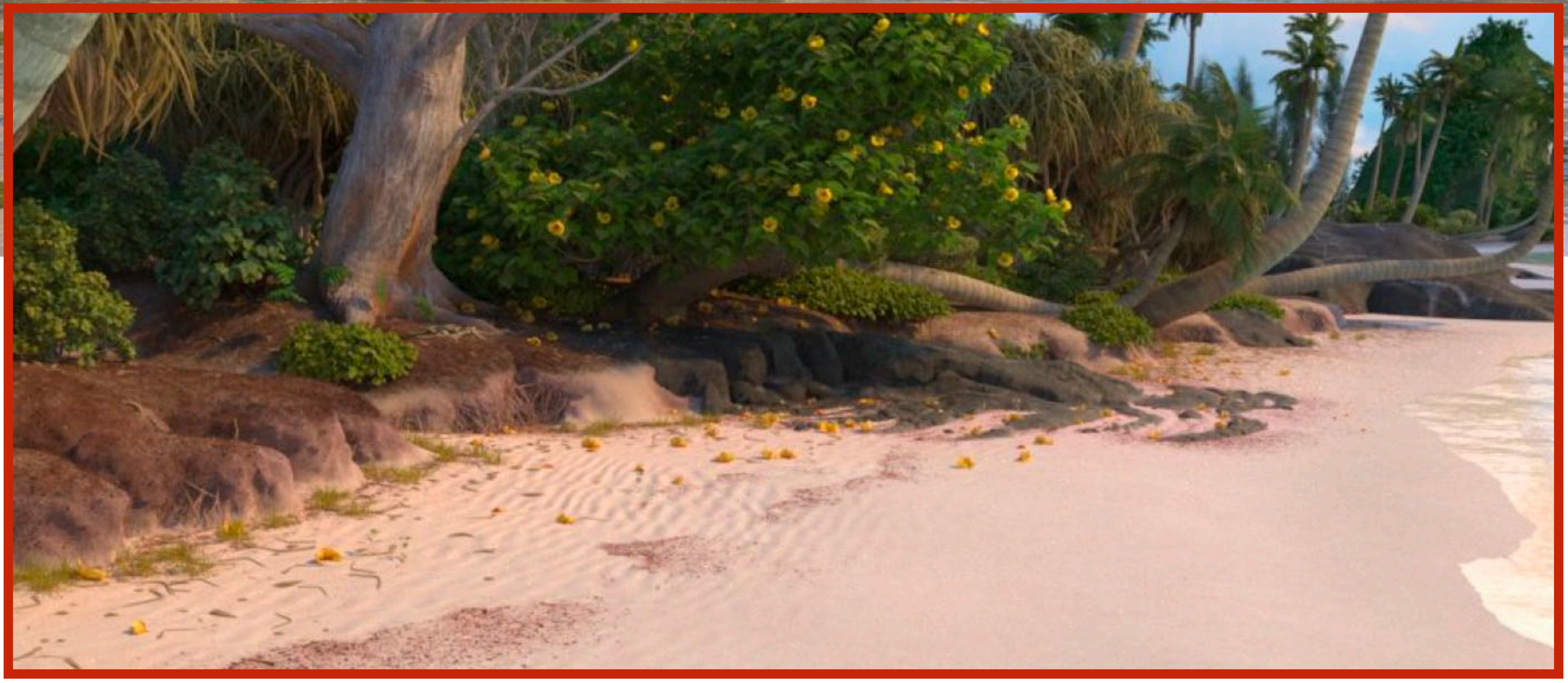
Refit rather than rebuild (then fix-up)



[Kopta et al. 2012]

Large, complex scenes



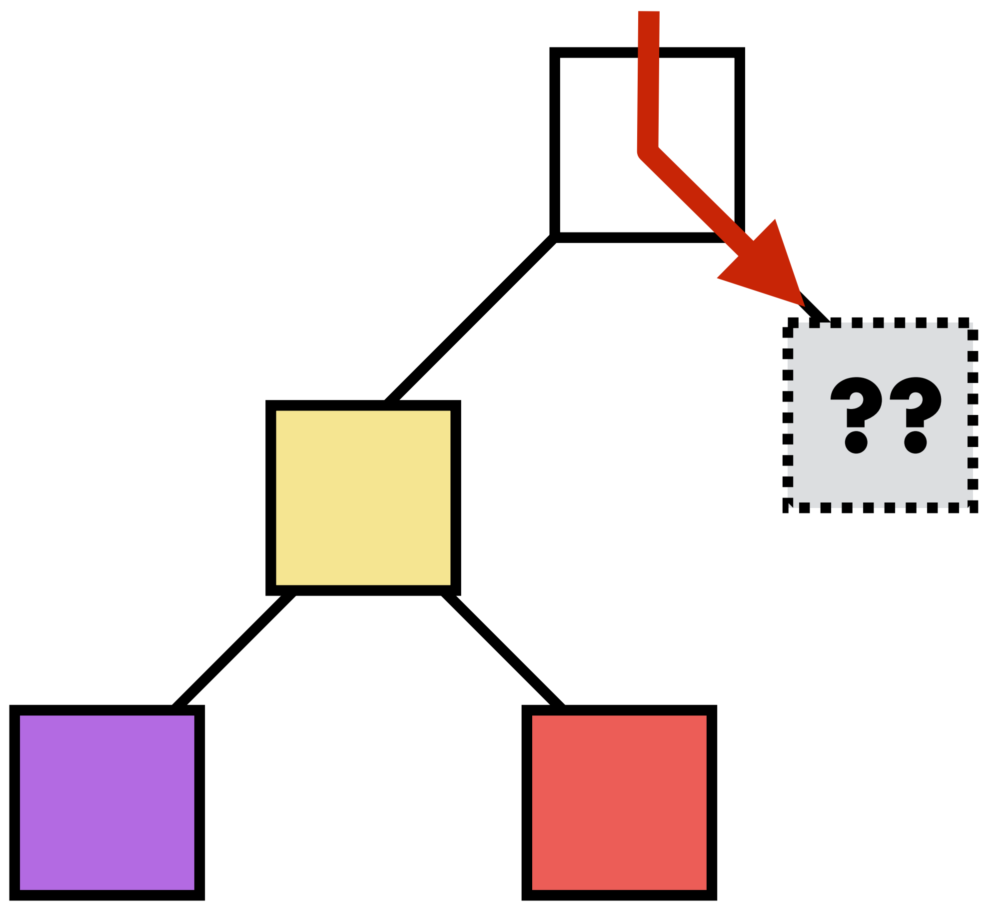


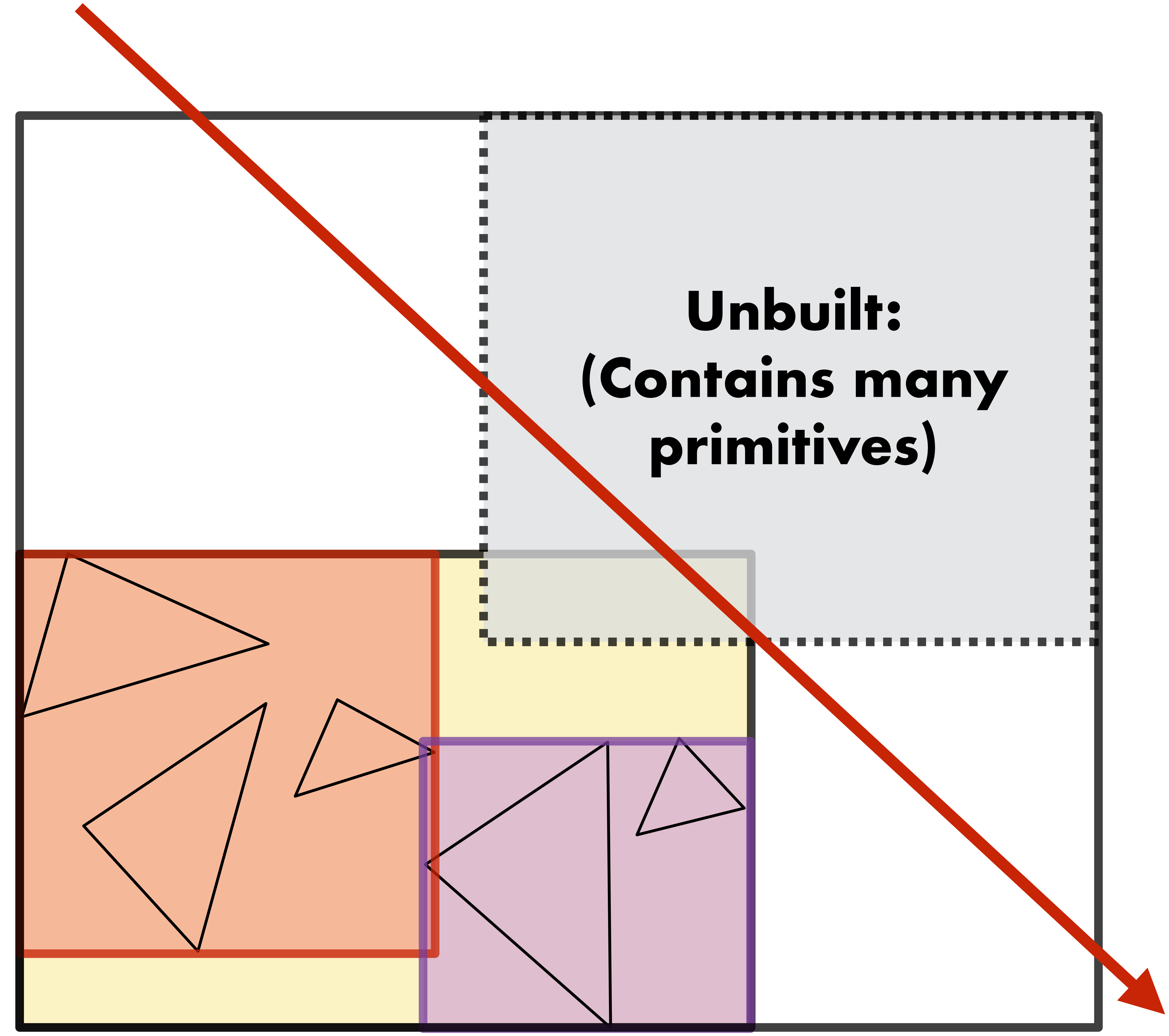
Consider a camera viewpoint where only a small fraction of the scene in visible.

Large parts of scene BVH may never be traced.*

* Even when accounting for shadow and bounce rays.

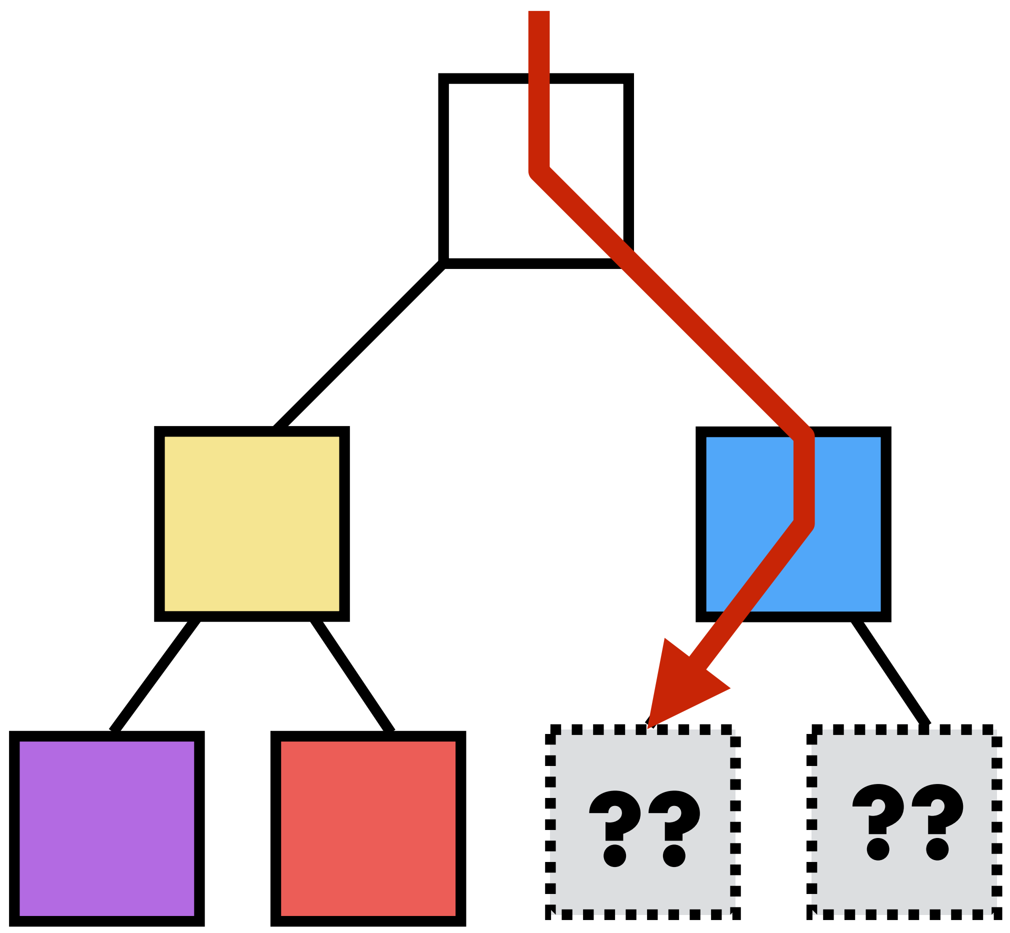
Build BVH "lazily" as needed

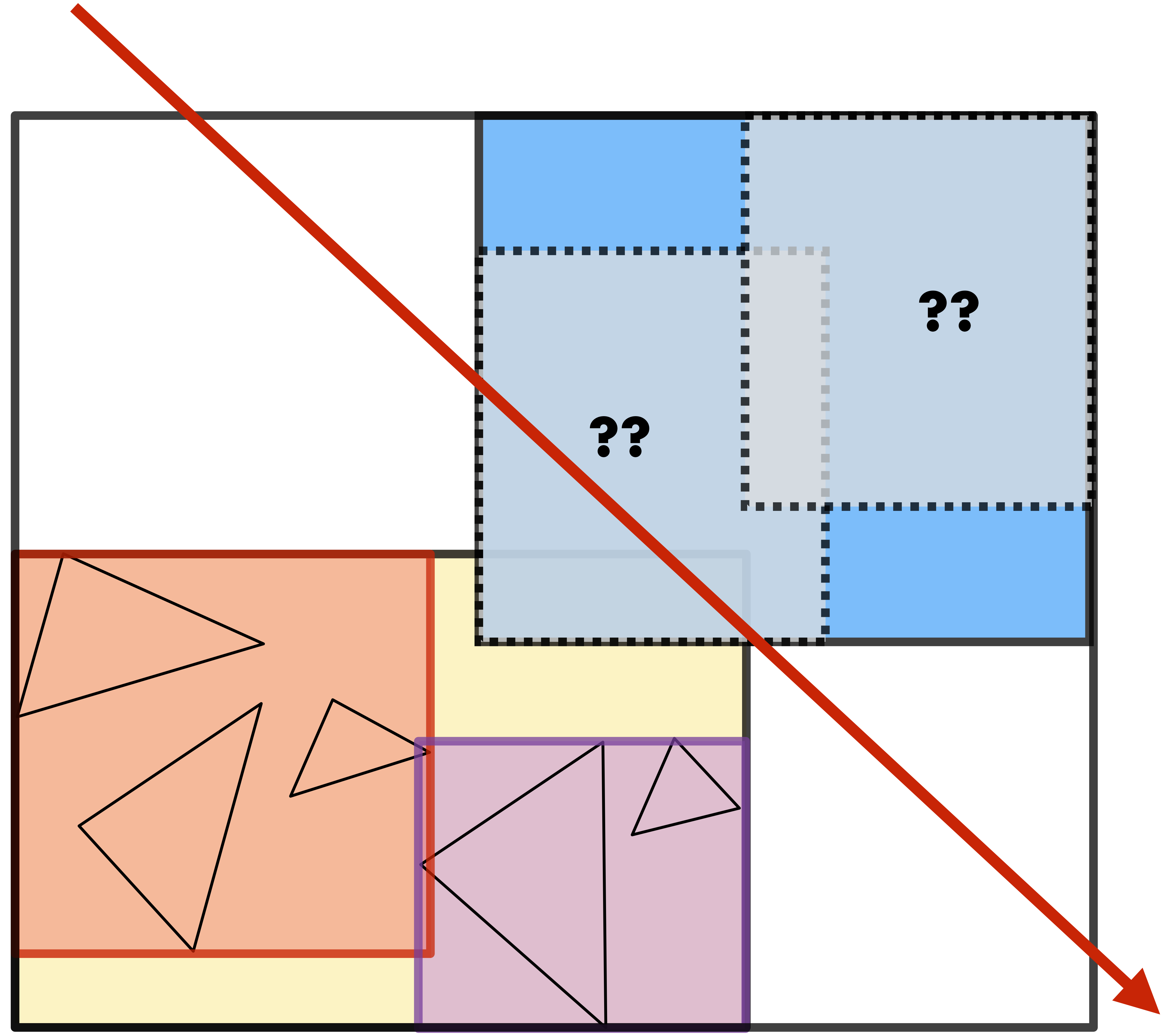




Build required subtree the first time its root node is "hit" by a ray

Build BVH "lazily" as needed

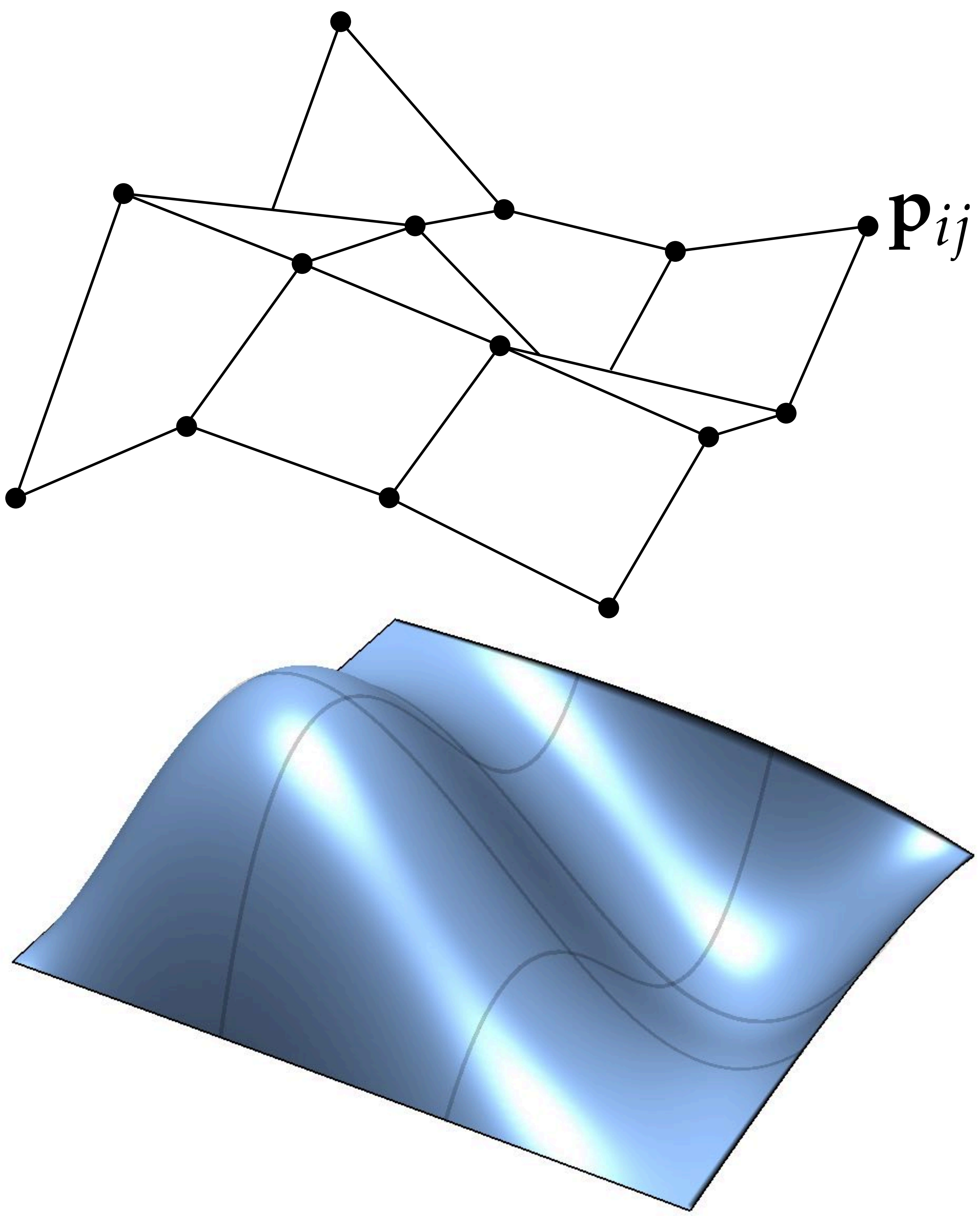


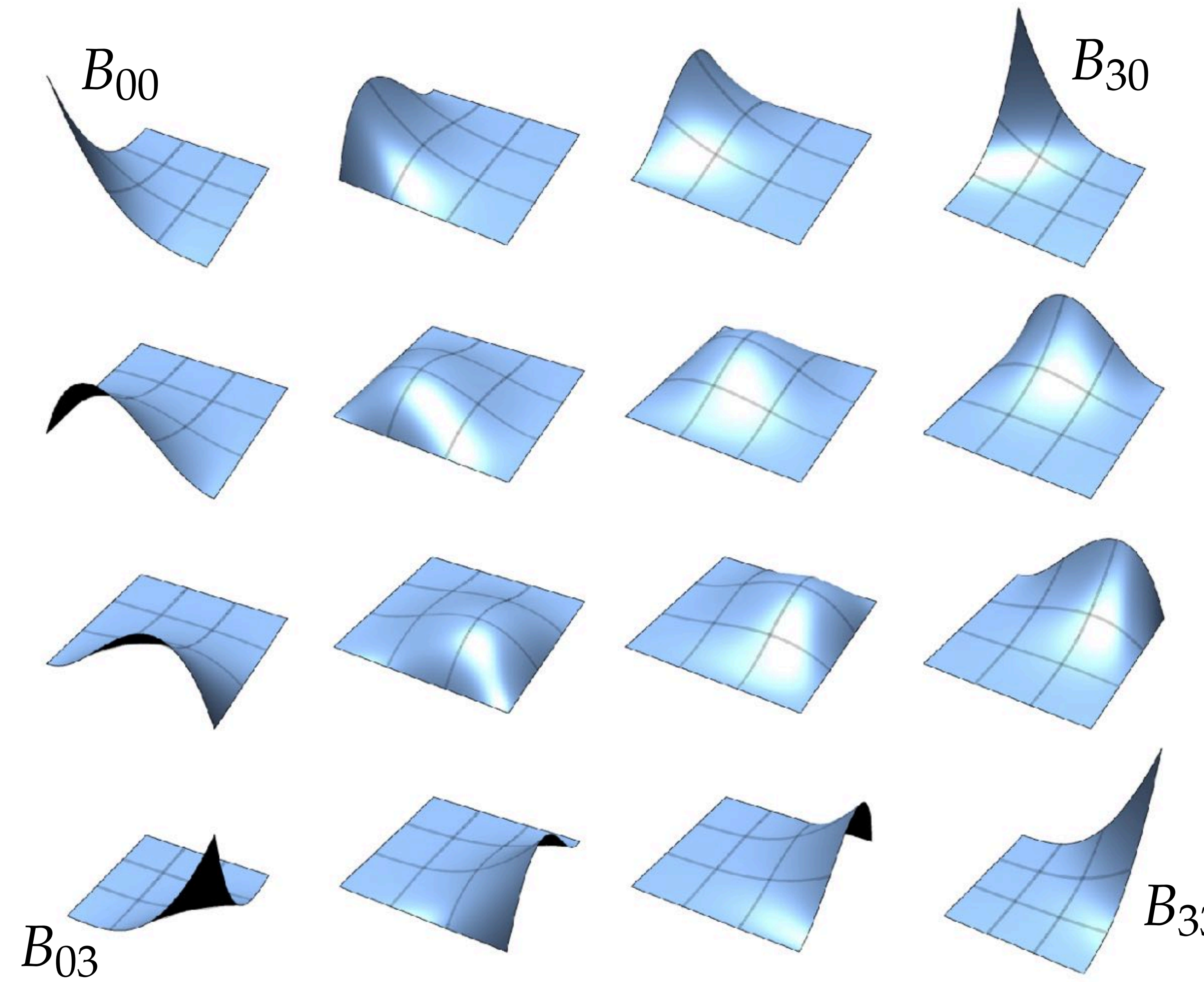


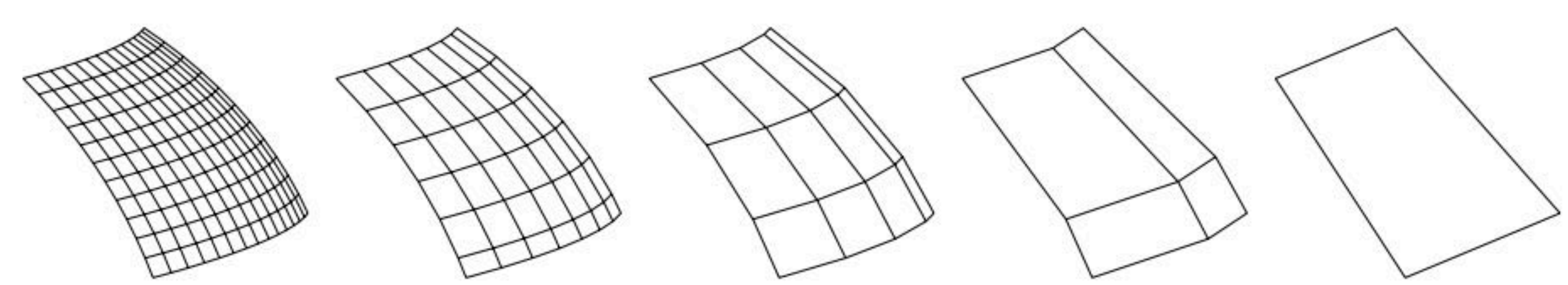
Build required subtree the first time its root node is "hit" by a ray

It can be efficient to evaluate complex surfaces lazily

Example: Bezier bicubic patch



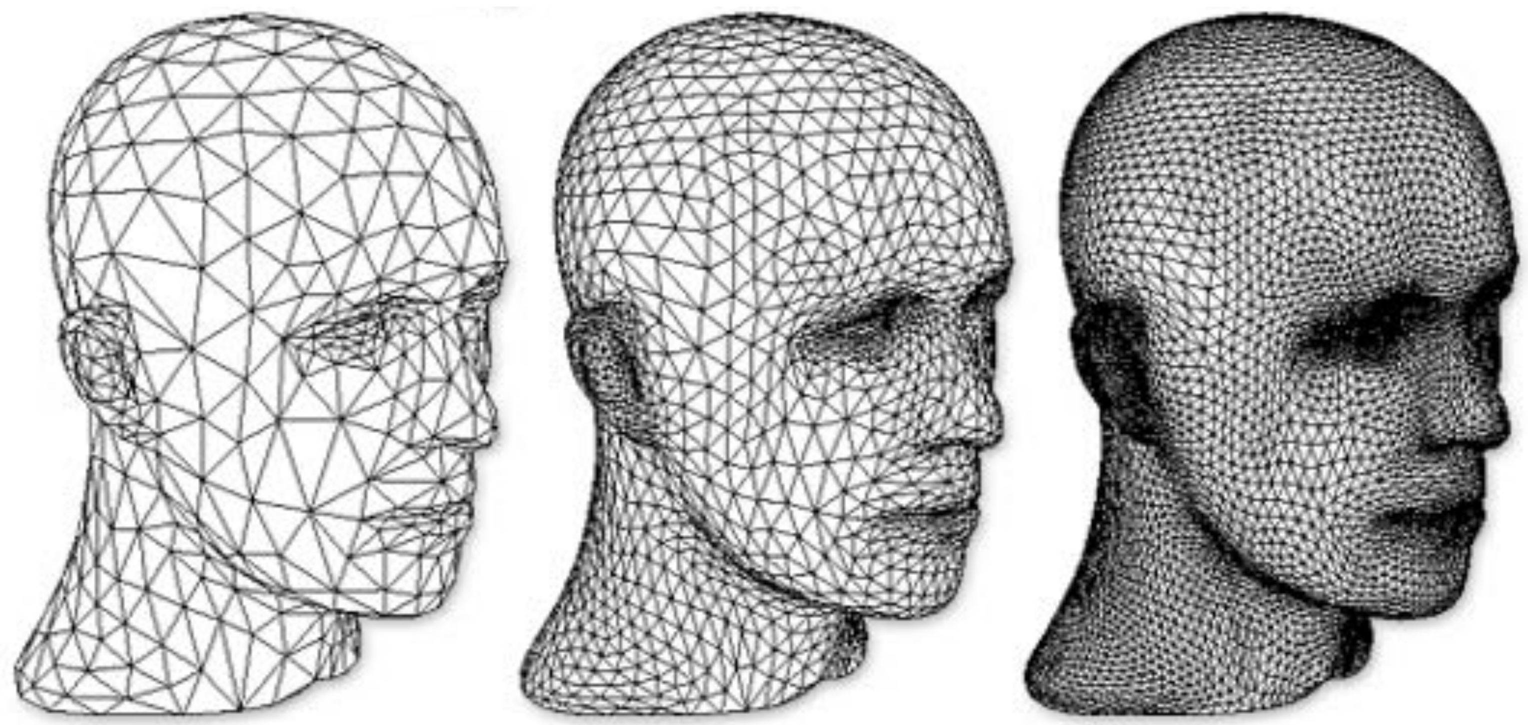




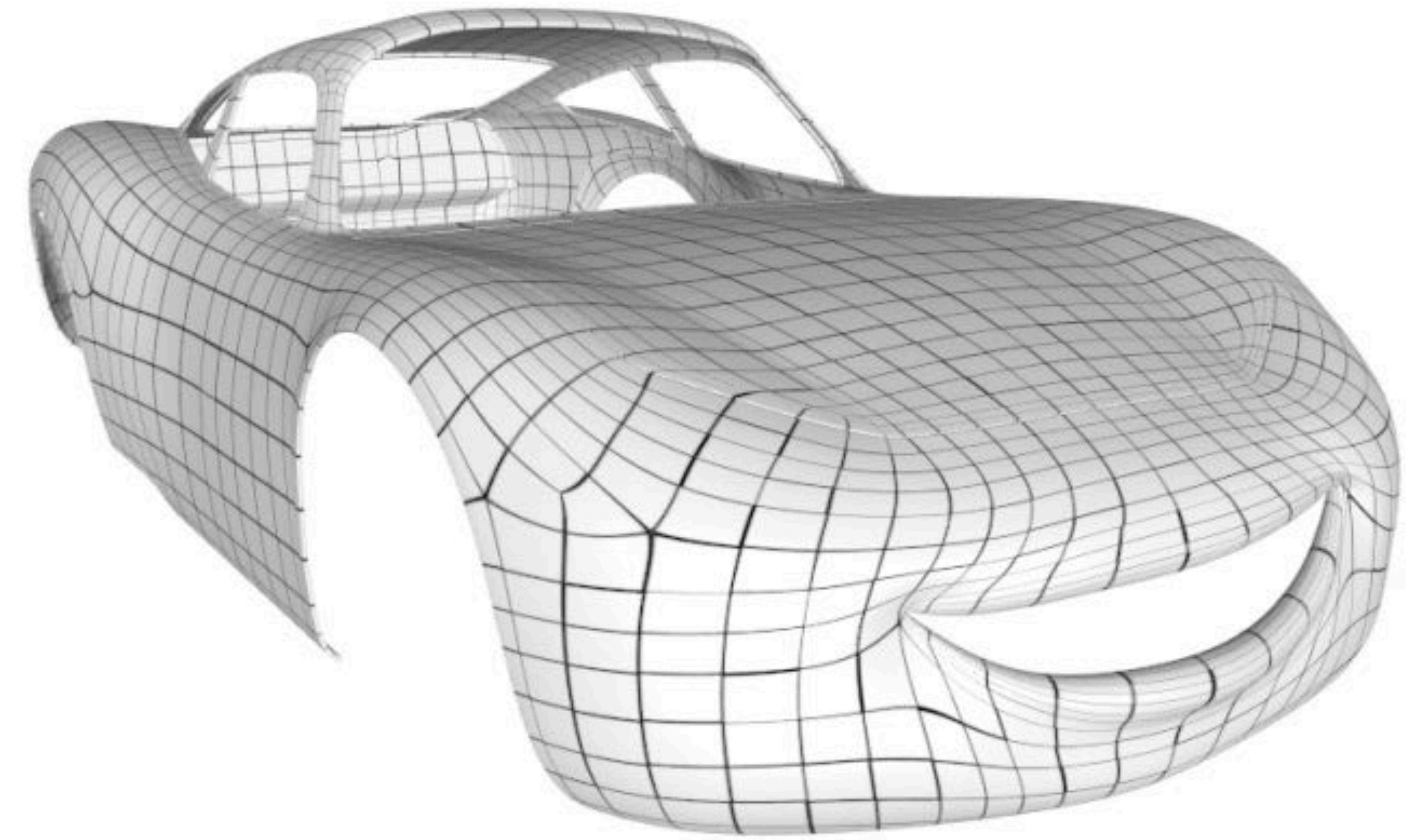
$$S(u,v) := \sum_{i=0}^{3} \sum_{j=0}^{3} B^3_{i,j}(u,v)\mathbf{p}_{ij}$$

It can be efficient to evaluate complex surfaces lazily

Example: Subdivision surfaces

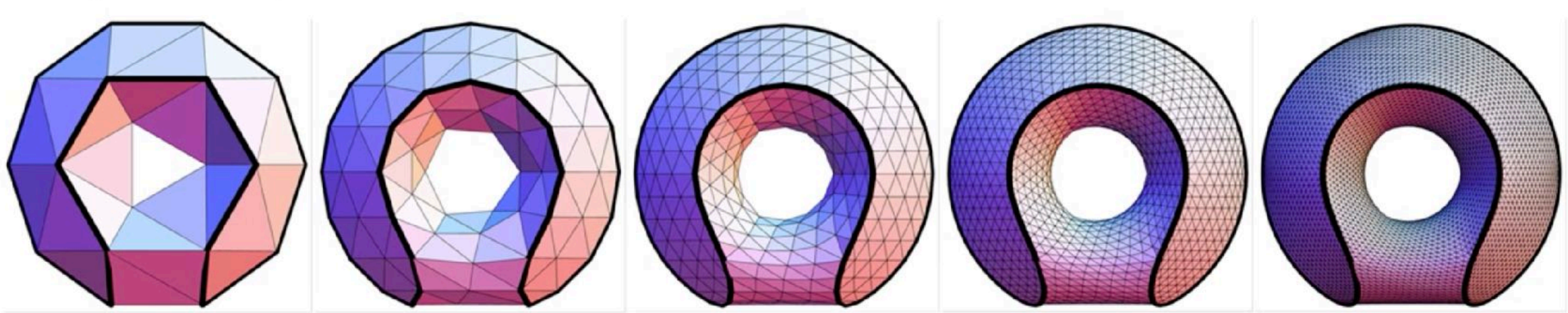


Loop subdivision

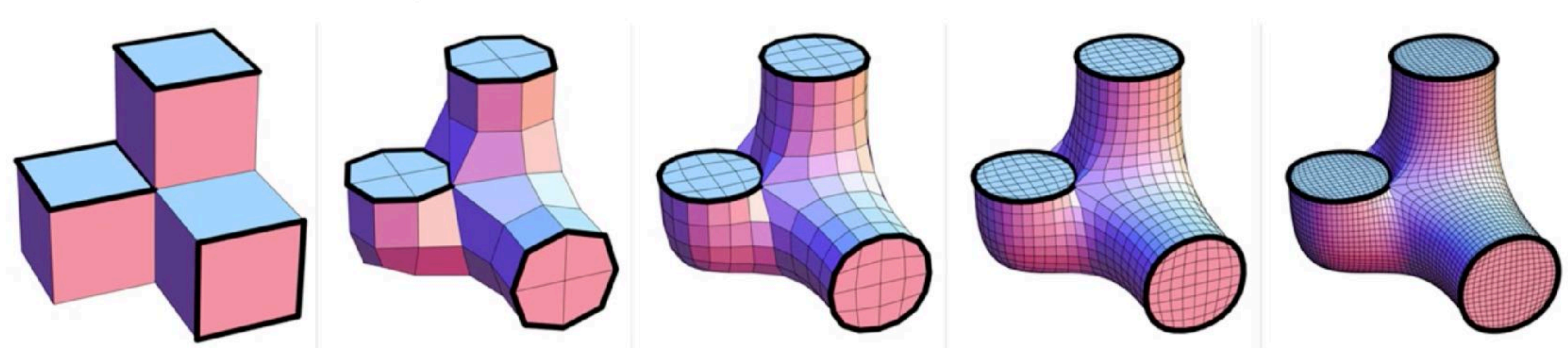


Catmull-Clark control mesh and limit surface

Loop with Sharp Creases

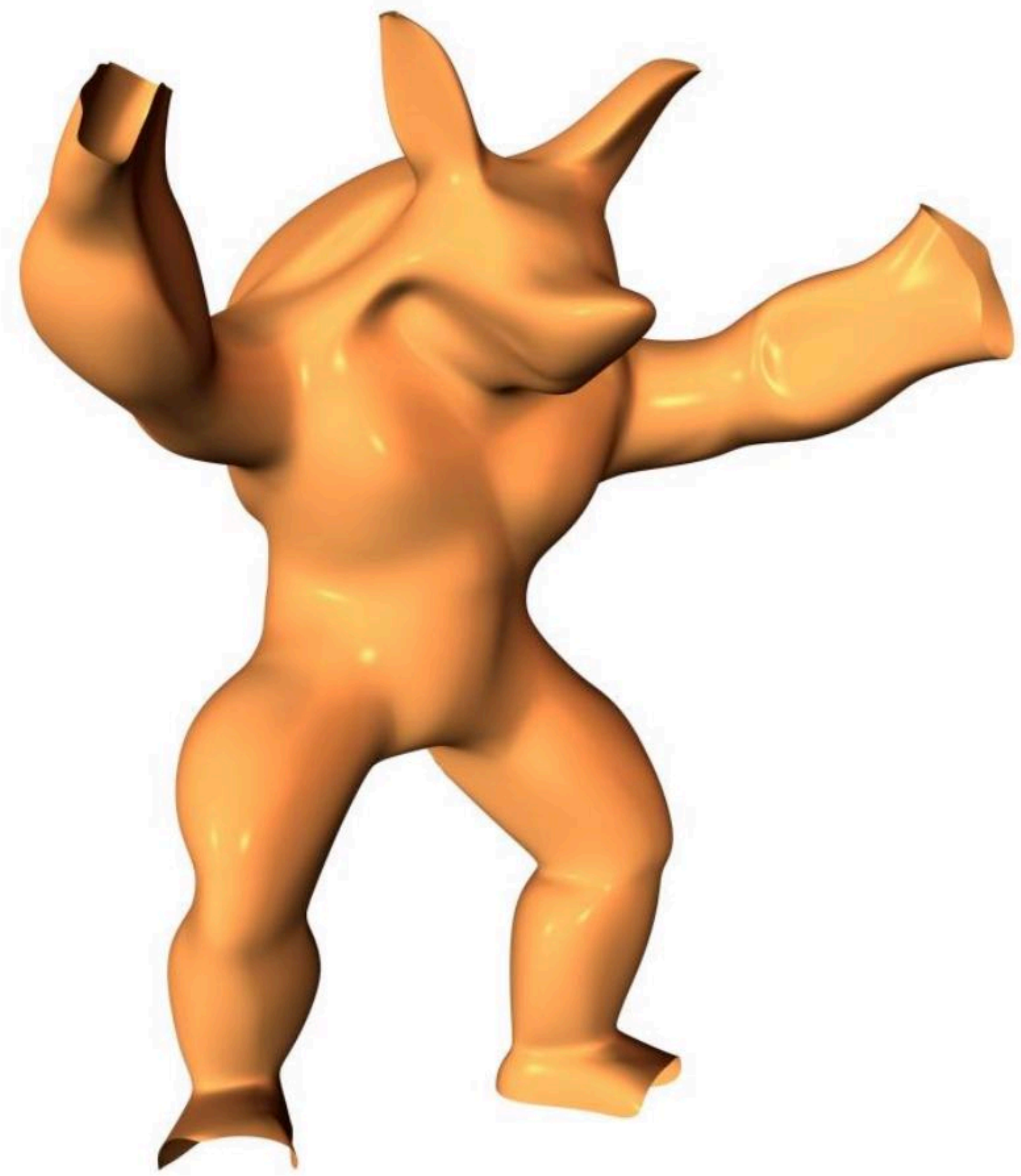


Catmull-Clark with Sharp Creases



Displaced subdivision surfaces

Control cage (Coarse triangle mesh)

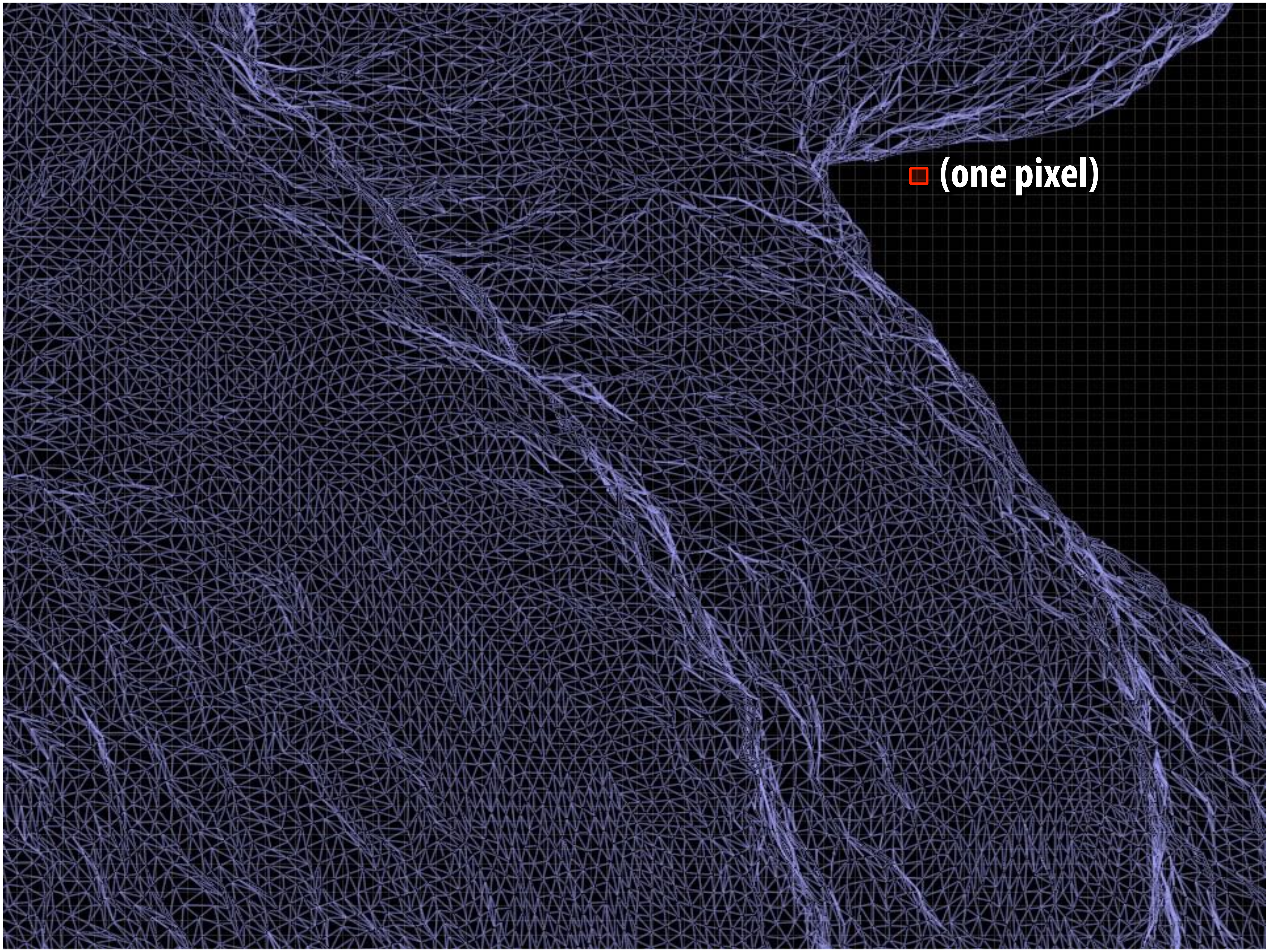


Limit surface (Renders from fine triangle mesh)

Displaced surface

[Lee 2000]

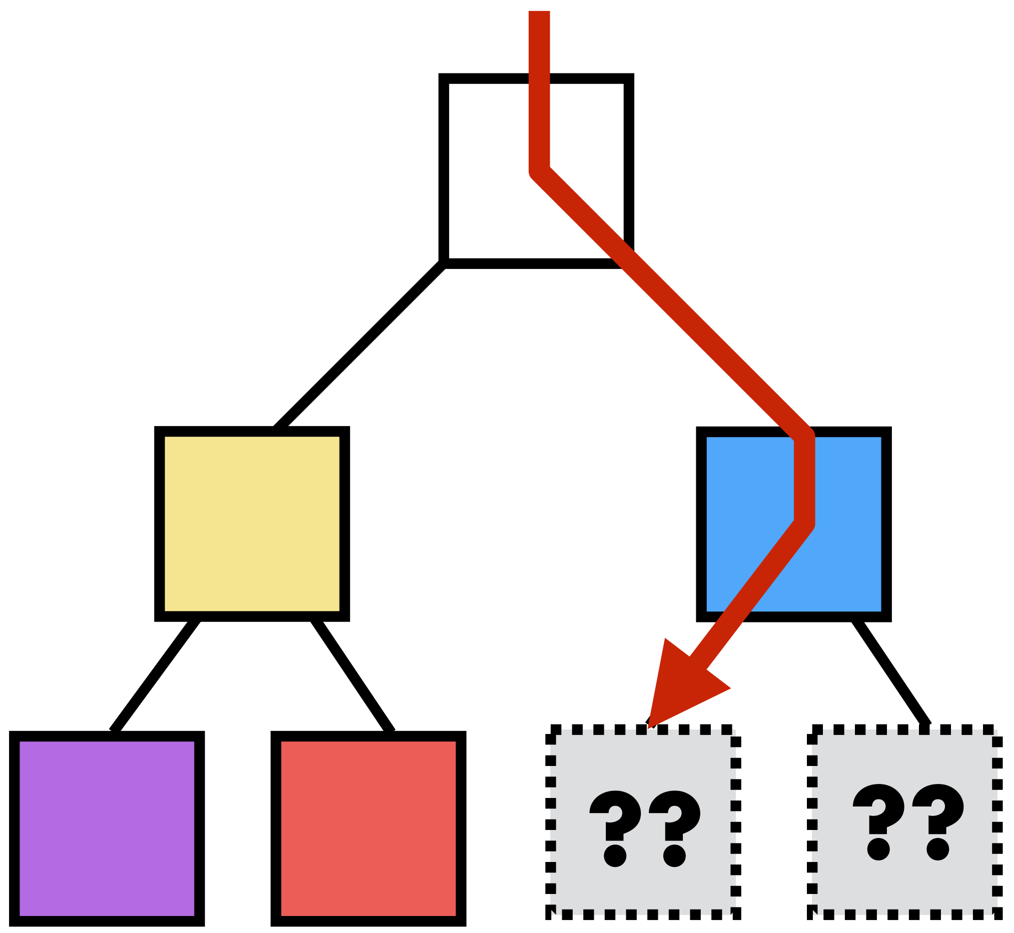
Result: high-resolution surface detail



Result: high-resolution surface detail

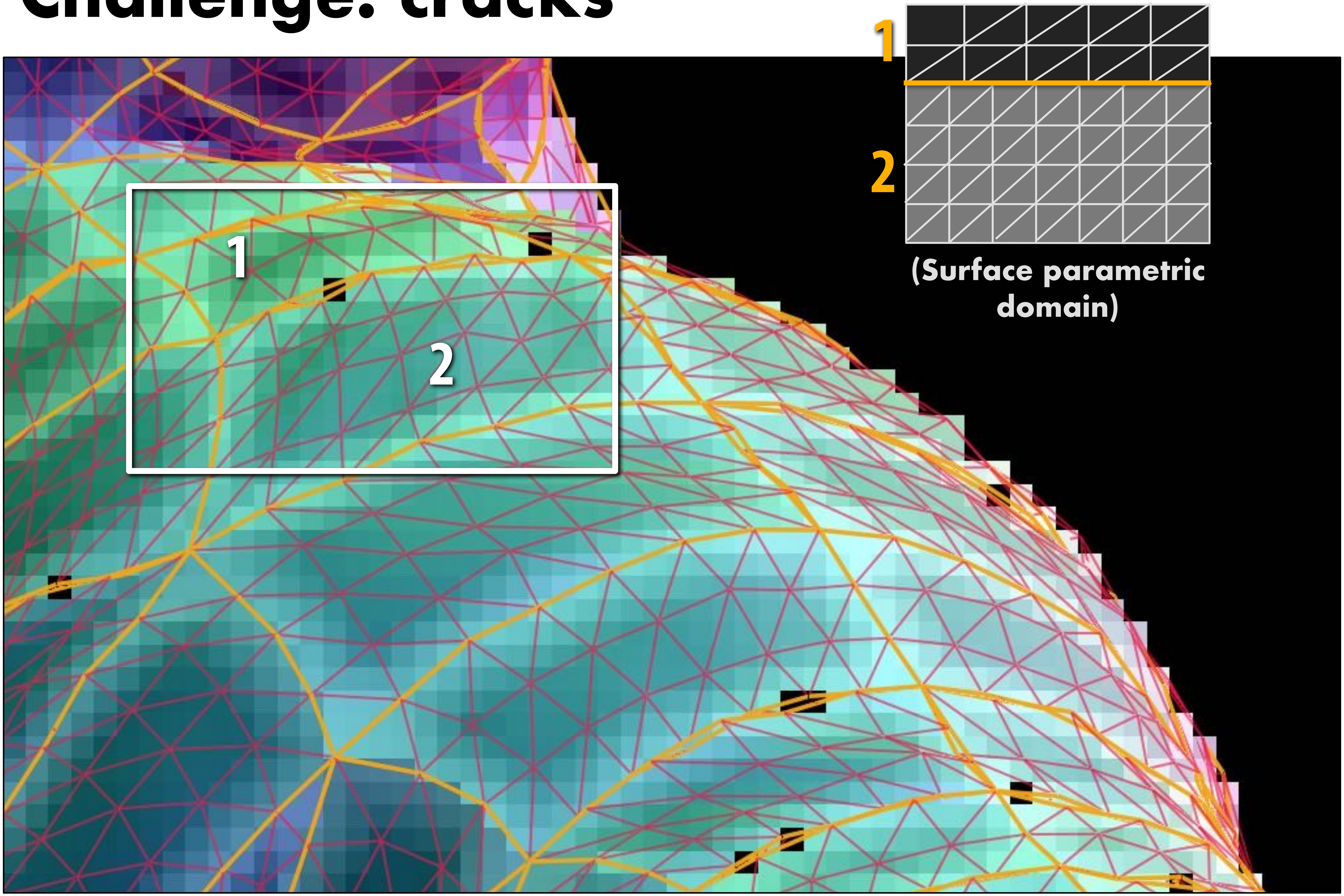
6M triangles after tessellation and displacement

Generate high-resolution geometry lazily



- Store patch bounding box + control points in memory
- Generate high-resolution mesh (via subdivision, evaluating bicubic, etc.) upon first ray entry
- Modern production renderers make different decisions about how to cache fine-geometry that is generated on-the-fly

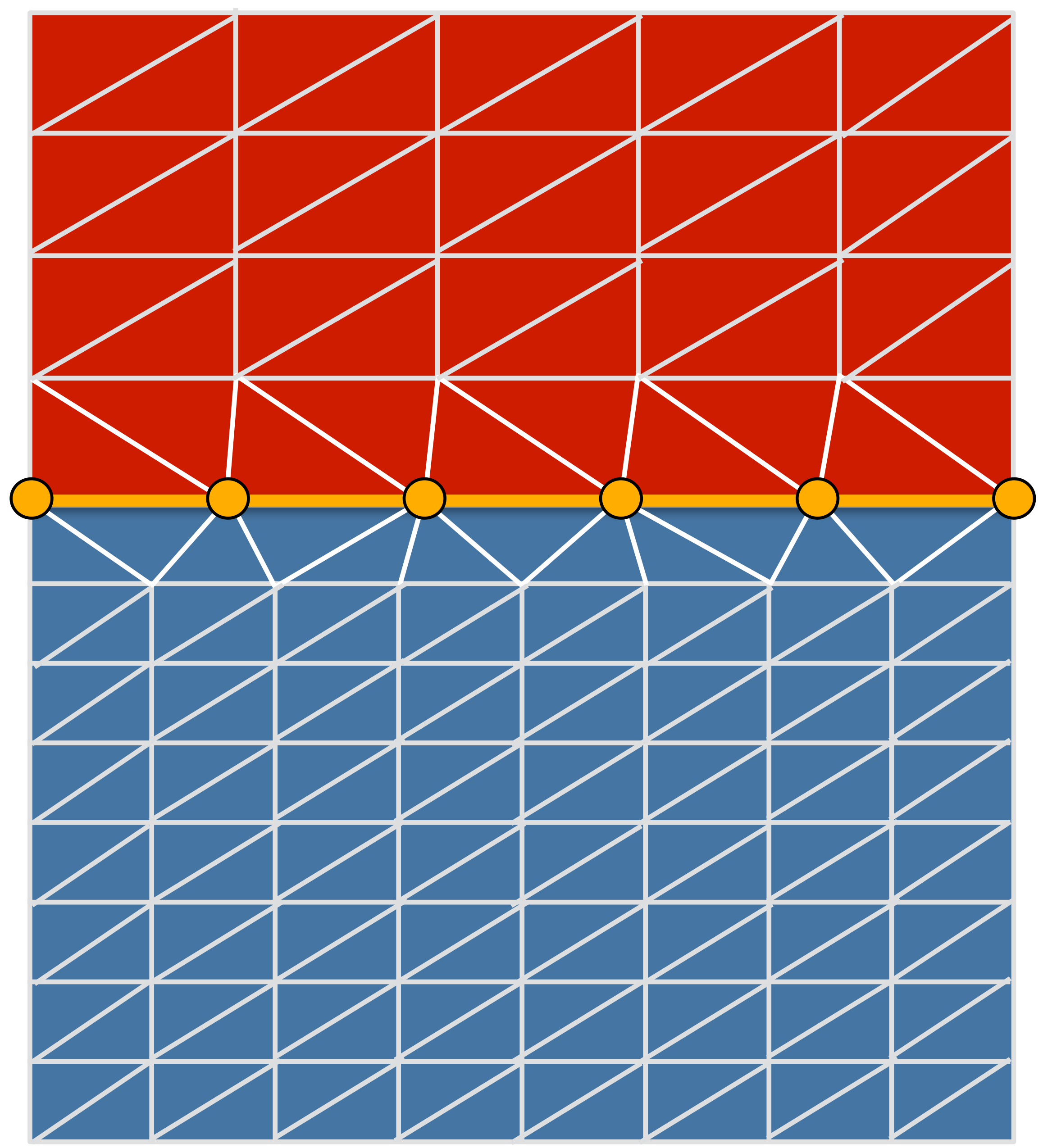
Challenge: cracks



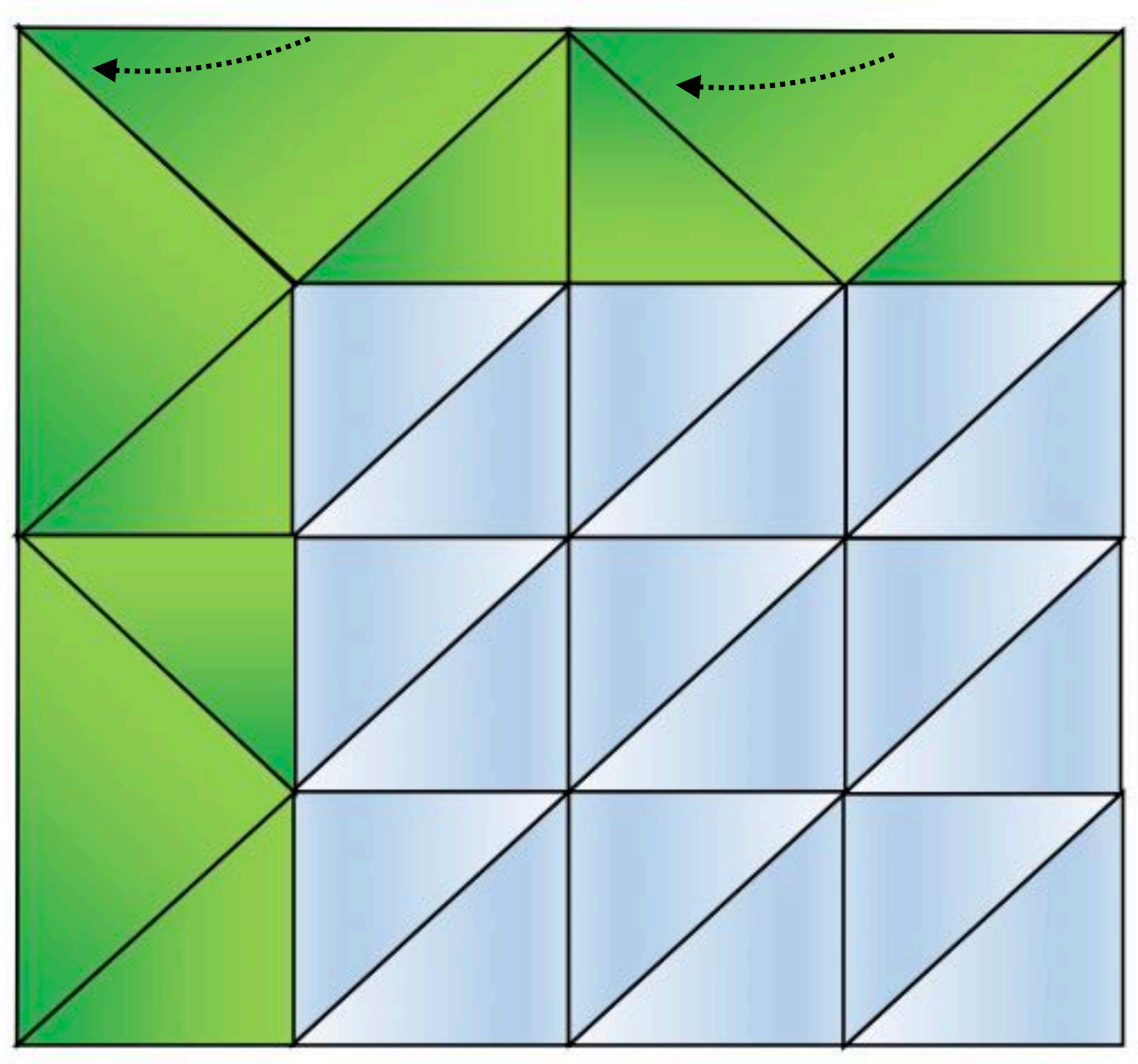
(Surface parametric domain)

Crack fixing solutions

Key idea: Adjacent regions agree on tessellation along edge



Generate irregular topology



5x5 regular vertex grid matching constraints on top & left edge of 3 segments (Vertices moves to create degenerate triangles)

Going further: accelerating intersections through hair and fur?

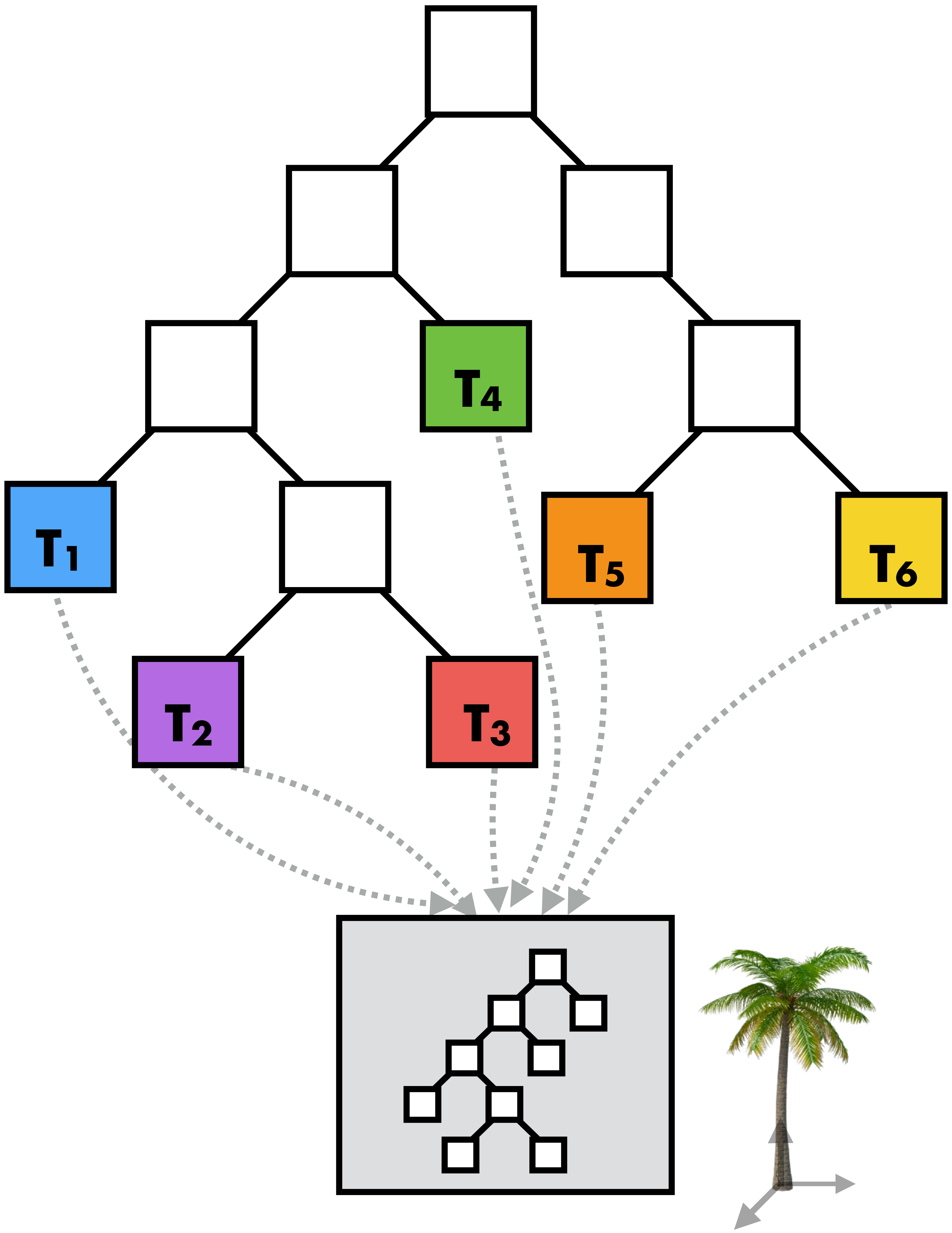
Instancing

Many high complexity scenes contain many copies of the same object

Disney Moana scene

15 billion total primitives in scene (BVH contains 15B prims)
But only 90M unique geometric primitives

Geometry instancing



Example:
BVH containing 6 primitives that share the same geometry.

Each instance:
- **Pointer to geometry**
- **Transformation to position the instance**

Q. Given an instance transform, how do we intersect the ray with the instance?

Interesting questions

What about instanced objects that are subdivision surfaces? (One instance might be close to camera, another is far.)

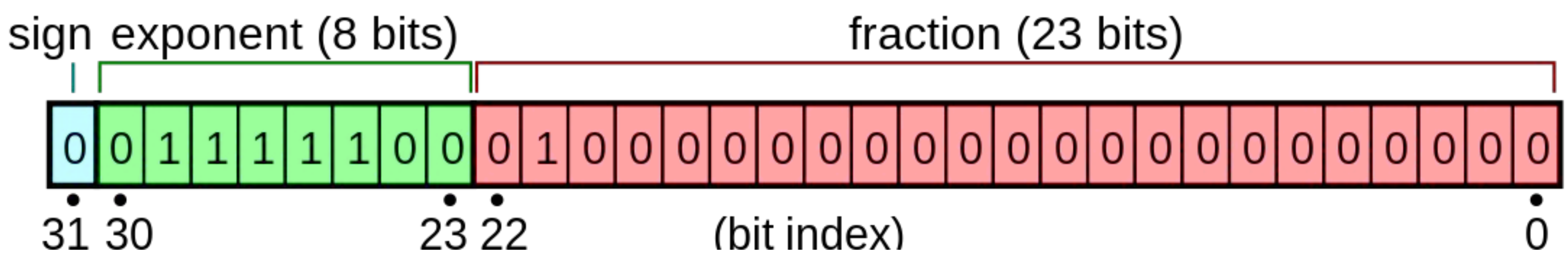
What level do we subdivide/tessellate to?

Floating-Point Error

Floating-point Representation

Scientific notation $\pm 1.m \times 2^e$

- **with a fixed sized mantissa (23-bits),**
- **a limited exponent range (8-bits, e-127),**
- **sign bit**

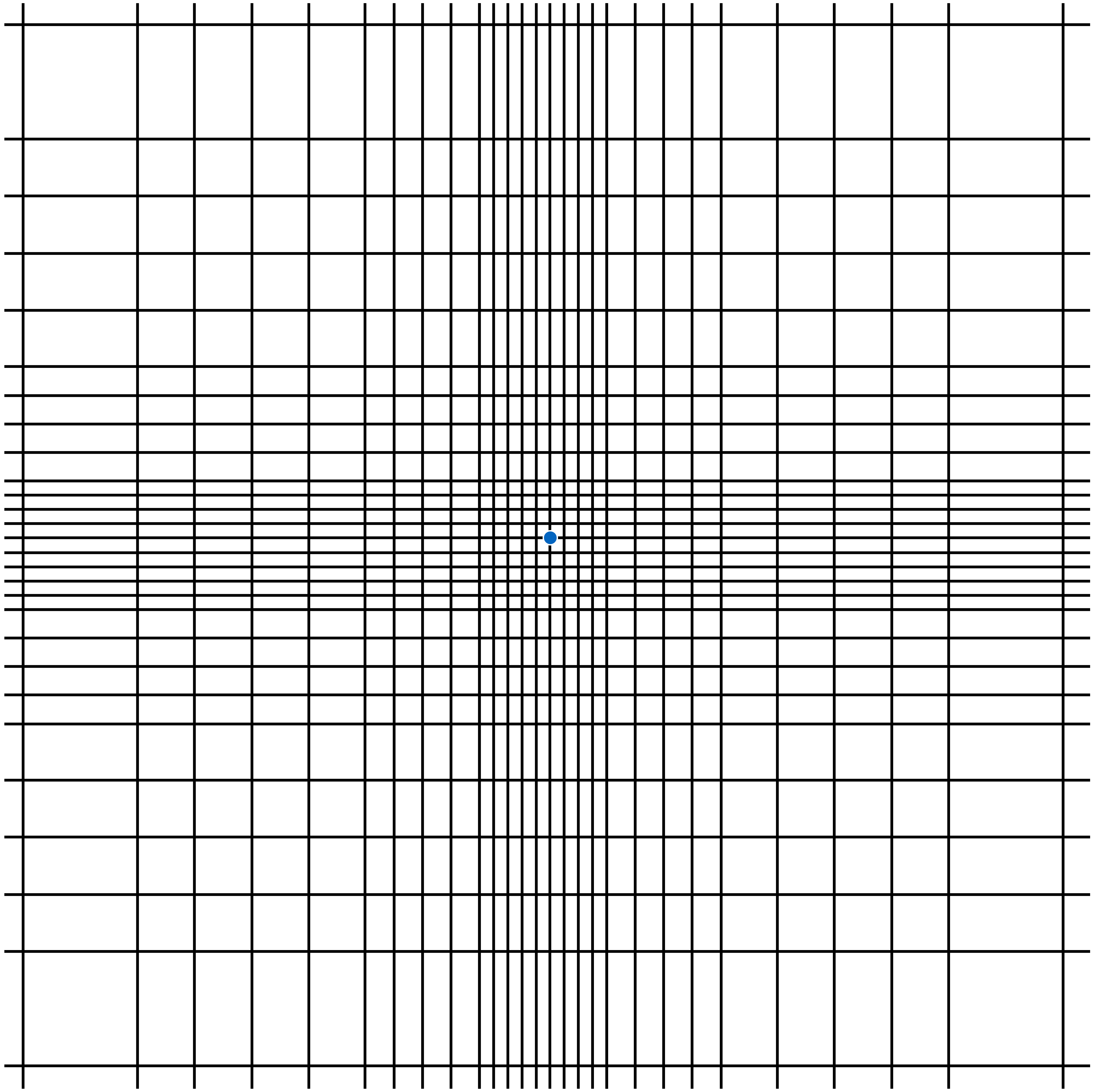


$$2.5 = 1.25 \times 2^1 = 1.01_b \times 2^1$$

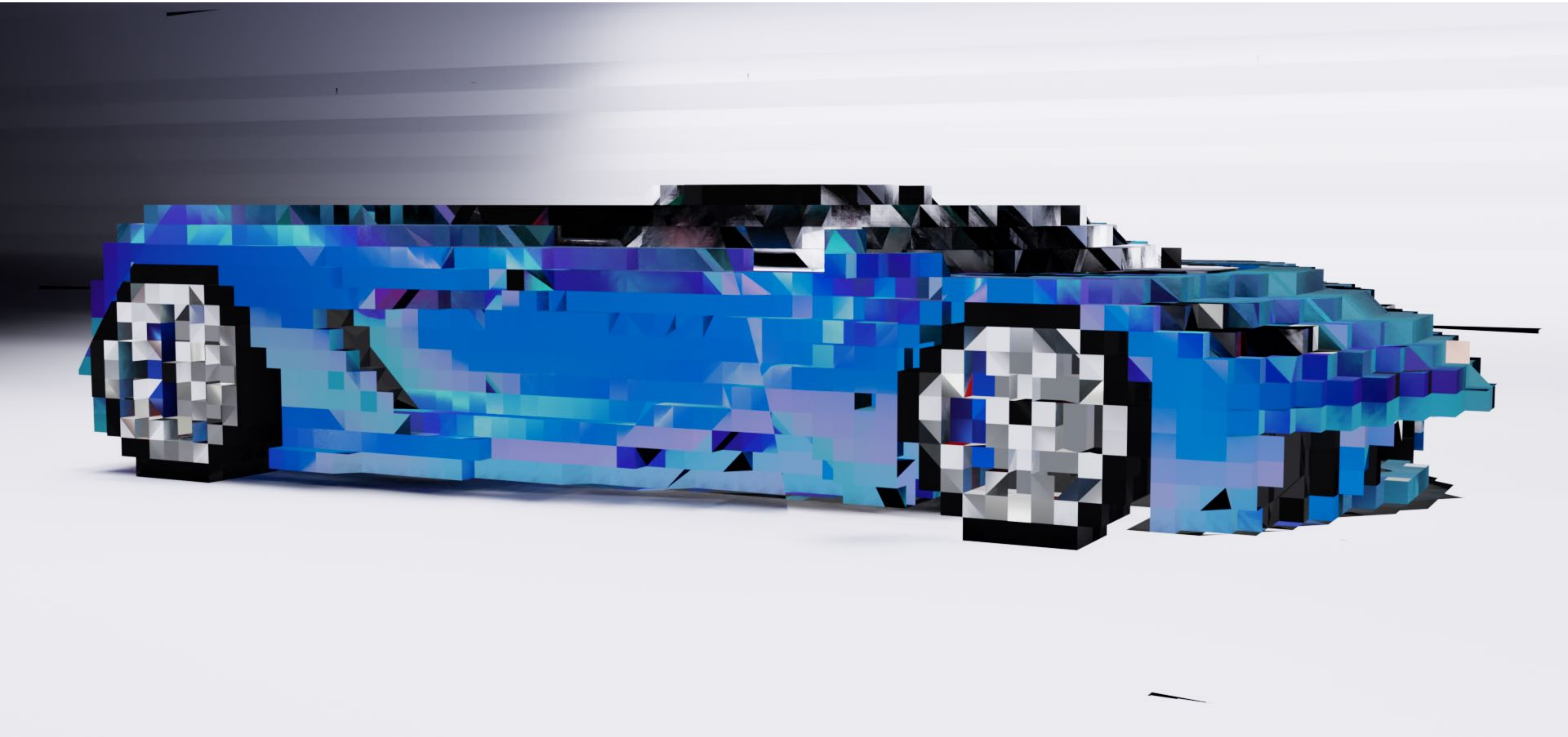
$$1/3 \approx 1.01010101010101010101011_b \times 2^{-2}$$

$$0 = ?$$

Floating-point Representation

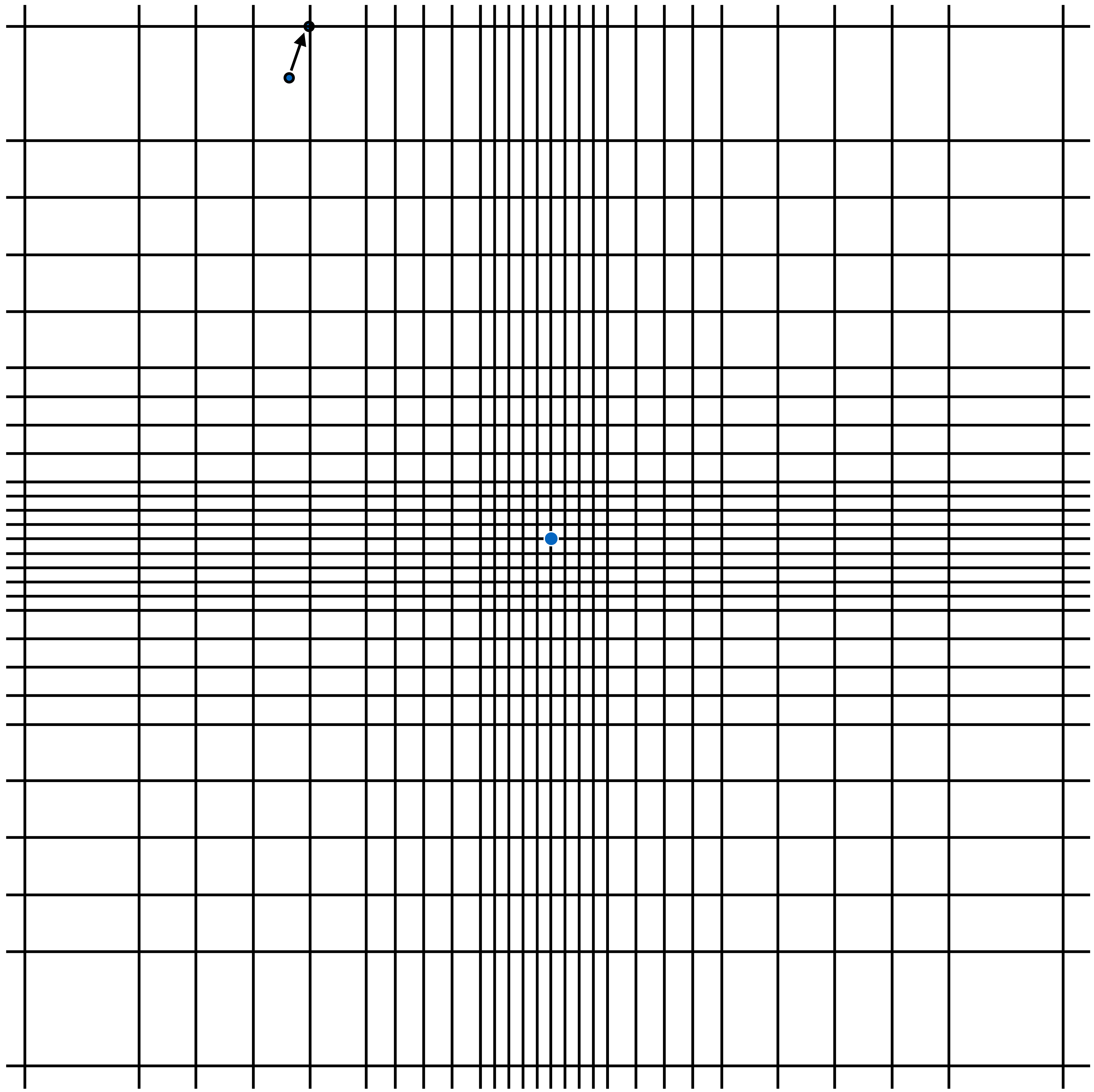


Near the origin

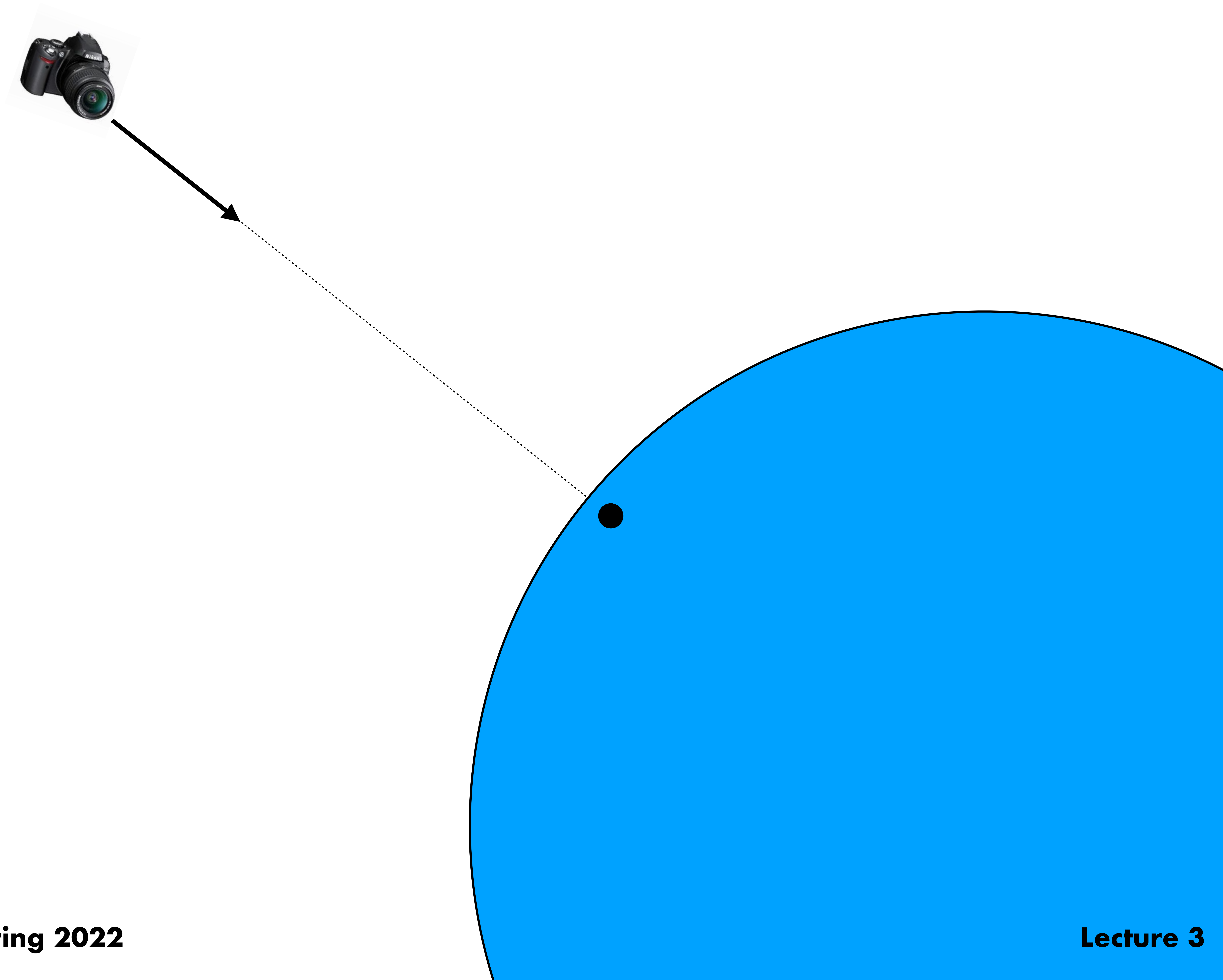


Translated 1M meters from the origin

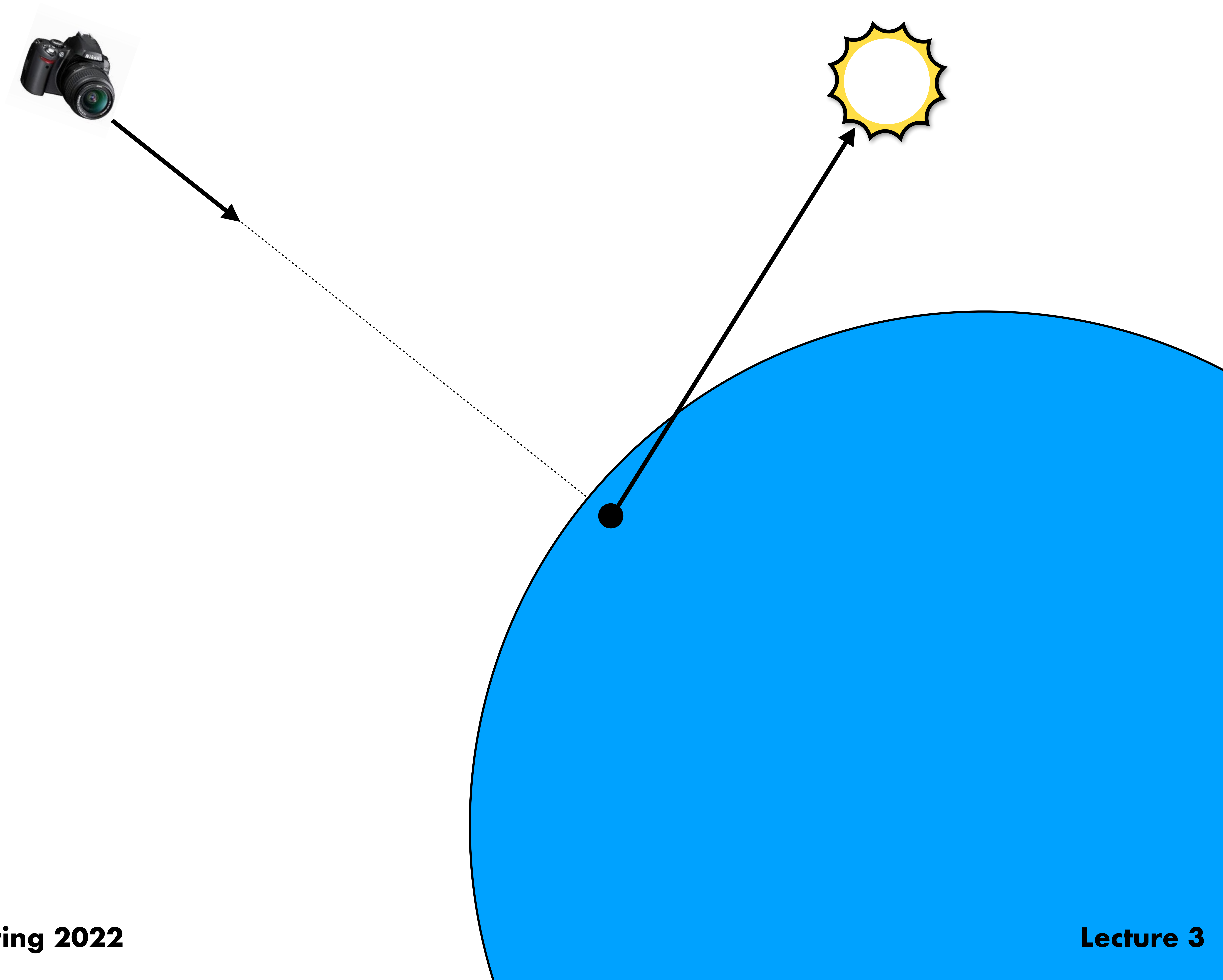
Roundoff Error



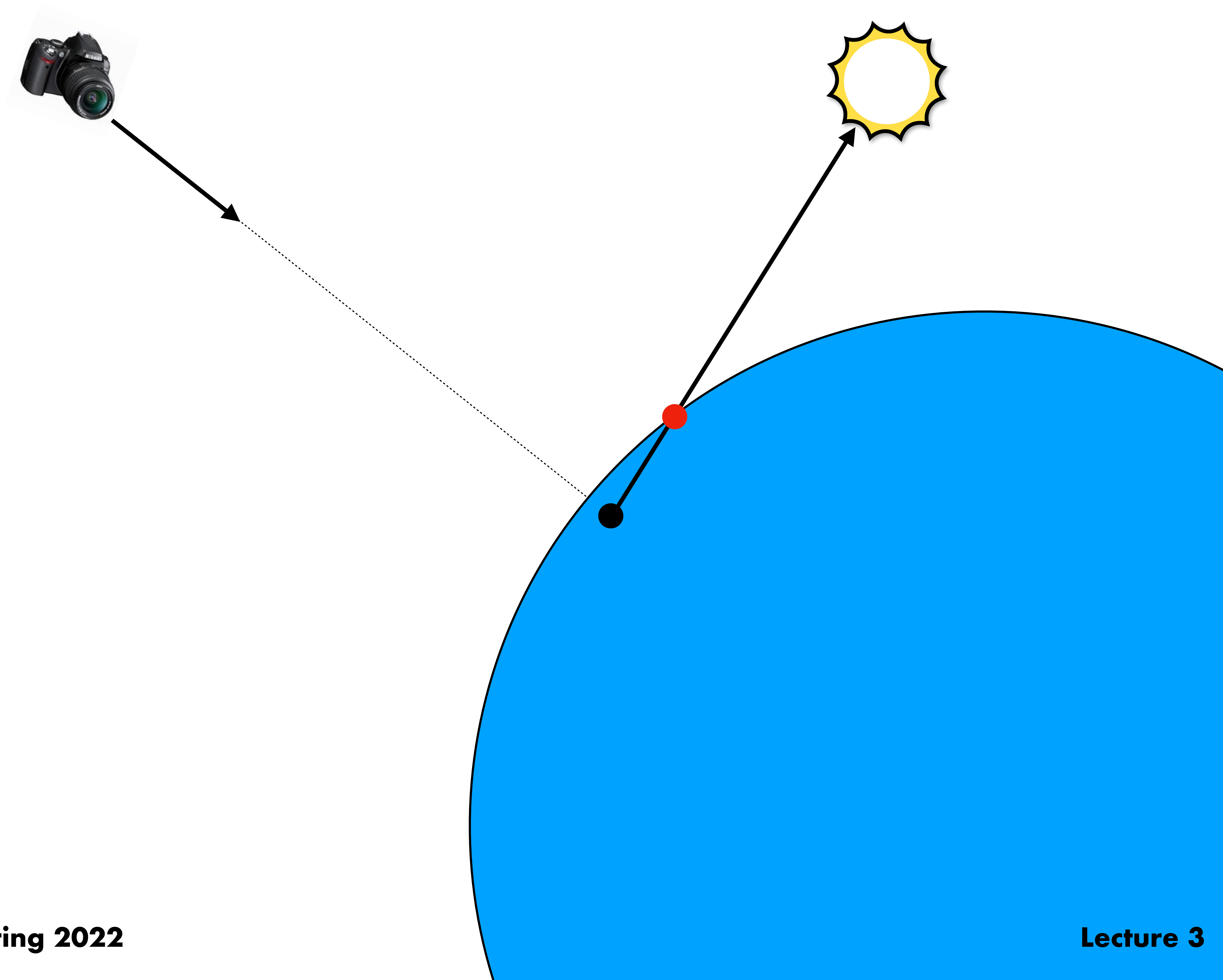
Roundoff Error and Ray Tracing

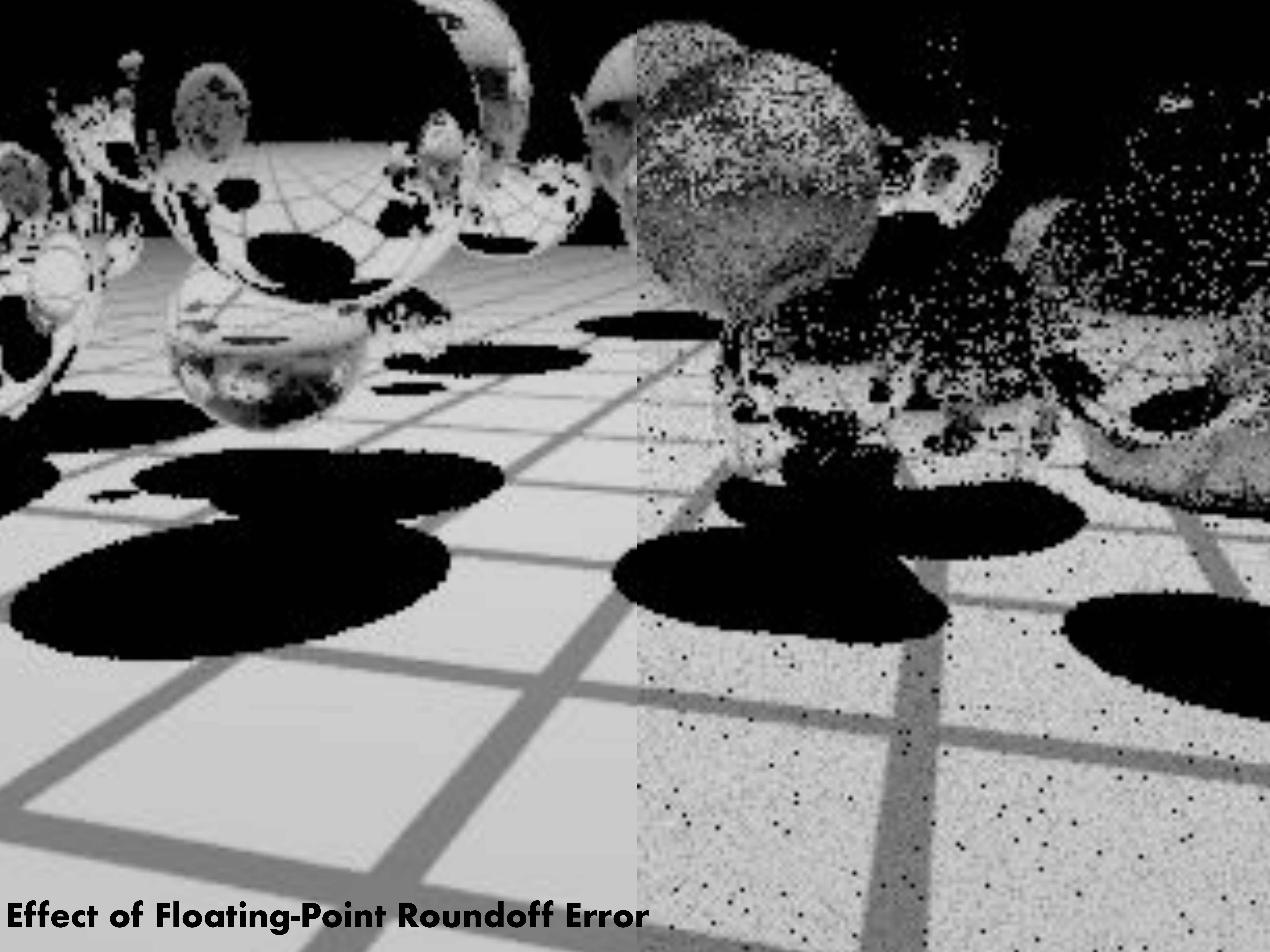


Roundoff Error and Ray Tracing



Roundoff Error and Ray Tracing



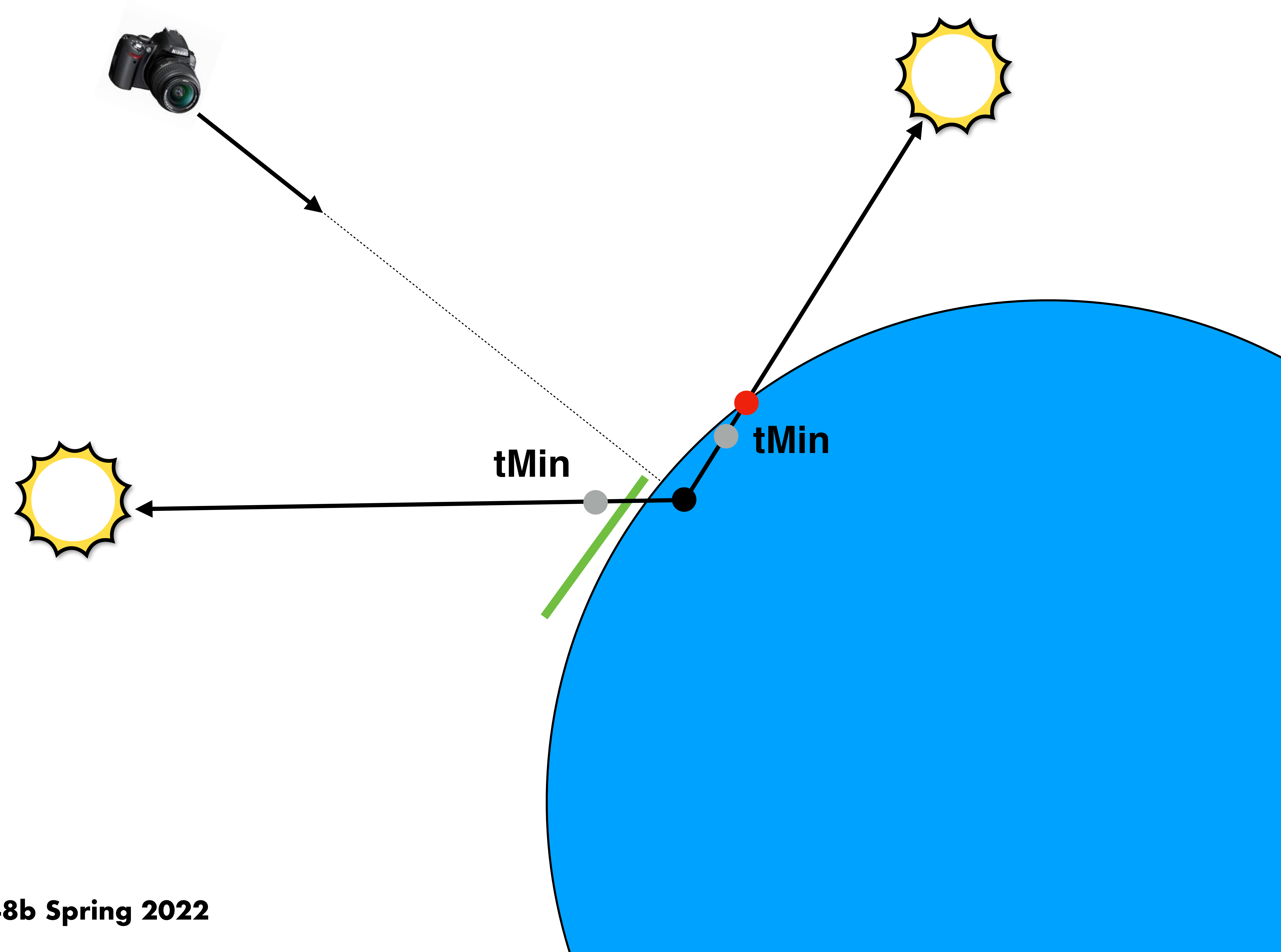


Effect of Floating-Point Roundoff Error

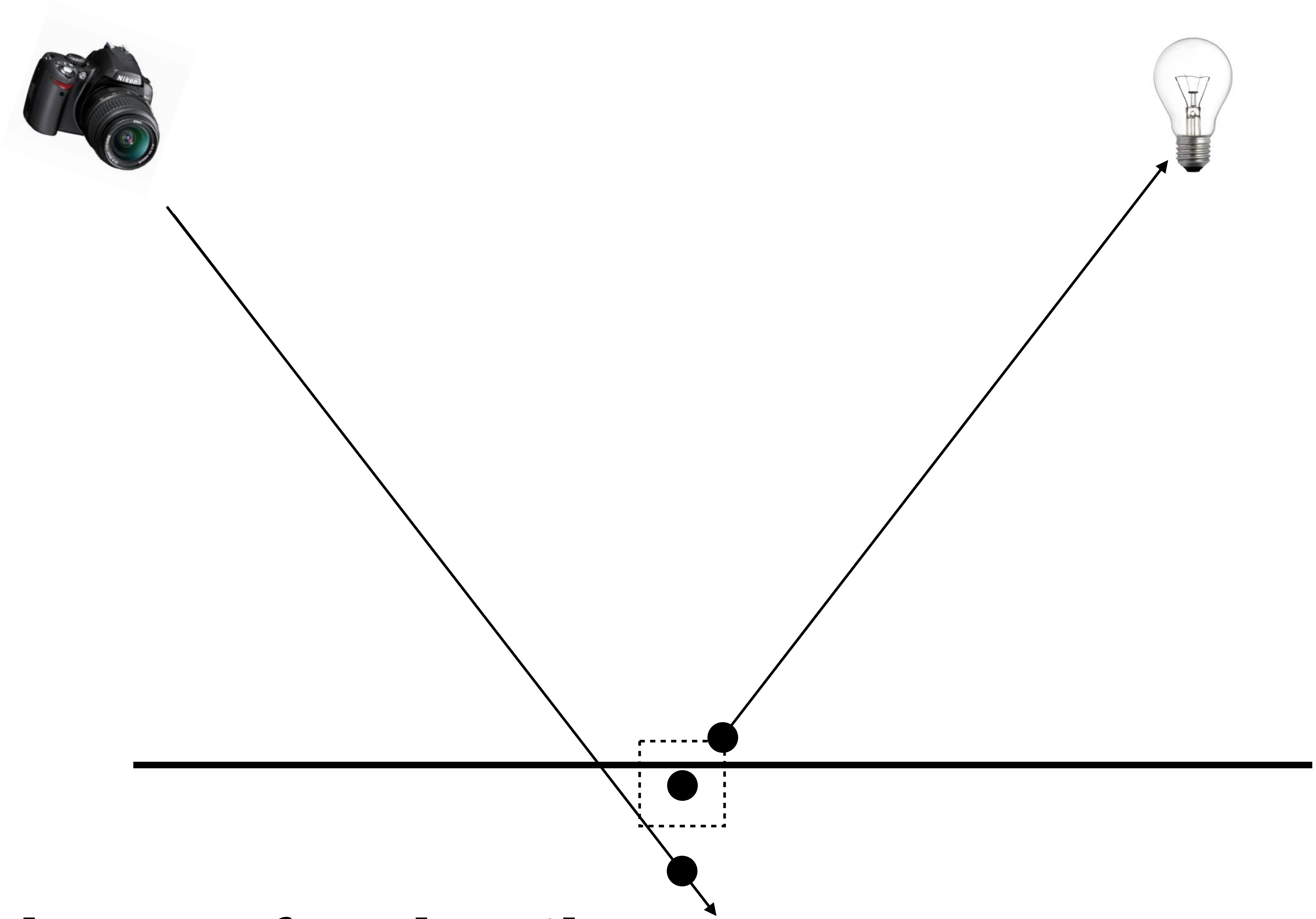
Effect of Roundoff Error

Round-off Error Remedies

Problems With a Fixed Epsilon

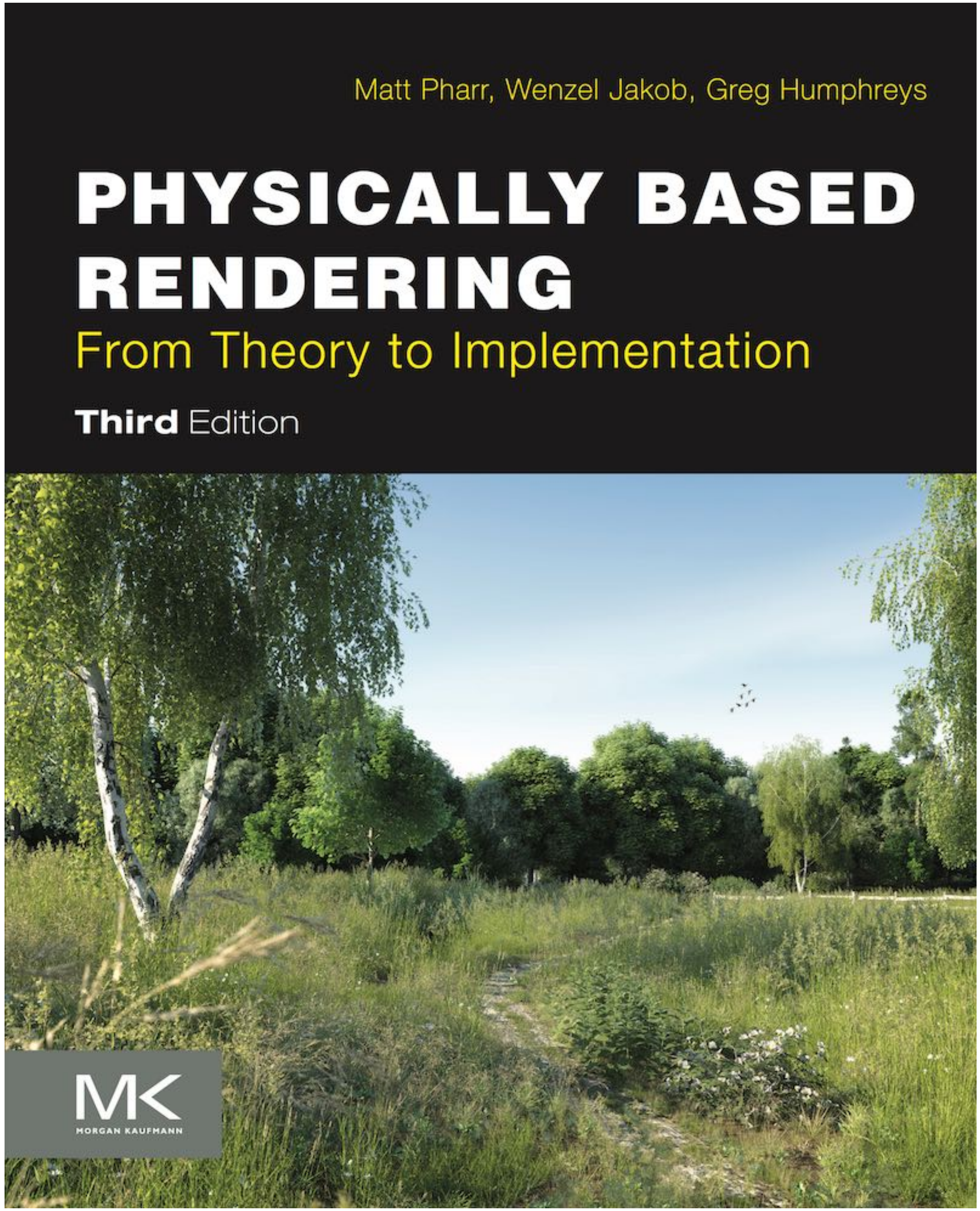


Better: Refine Intersection, Bound Error



See pbrt 3.9 for details

PBRT Overview



http://www.pbr-book.org/

Table 1.1: Main Interface Types. Most of pbrt is implemented in terms of 10 key abstract base classes, listed here. Implementations of each of these can easily be added to the system to extend its functionality.

Base class	Directory	Section
Shape	shapes/	3.1
Aggregate	accelerators/	4.2
Camera	cameras/	6.1
Sampler	samplers/	7.2
Filter	filters/	7.8
Material	materials/	9.2
Texture	textures/	10.3
Medium	media/	11.3
Light	lights/	12.2
Integrator	integrators/	1.3.3

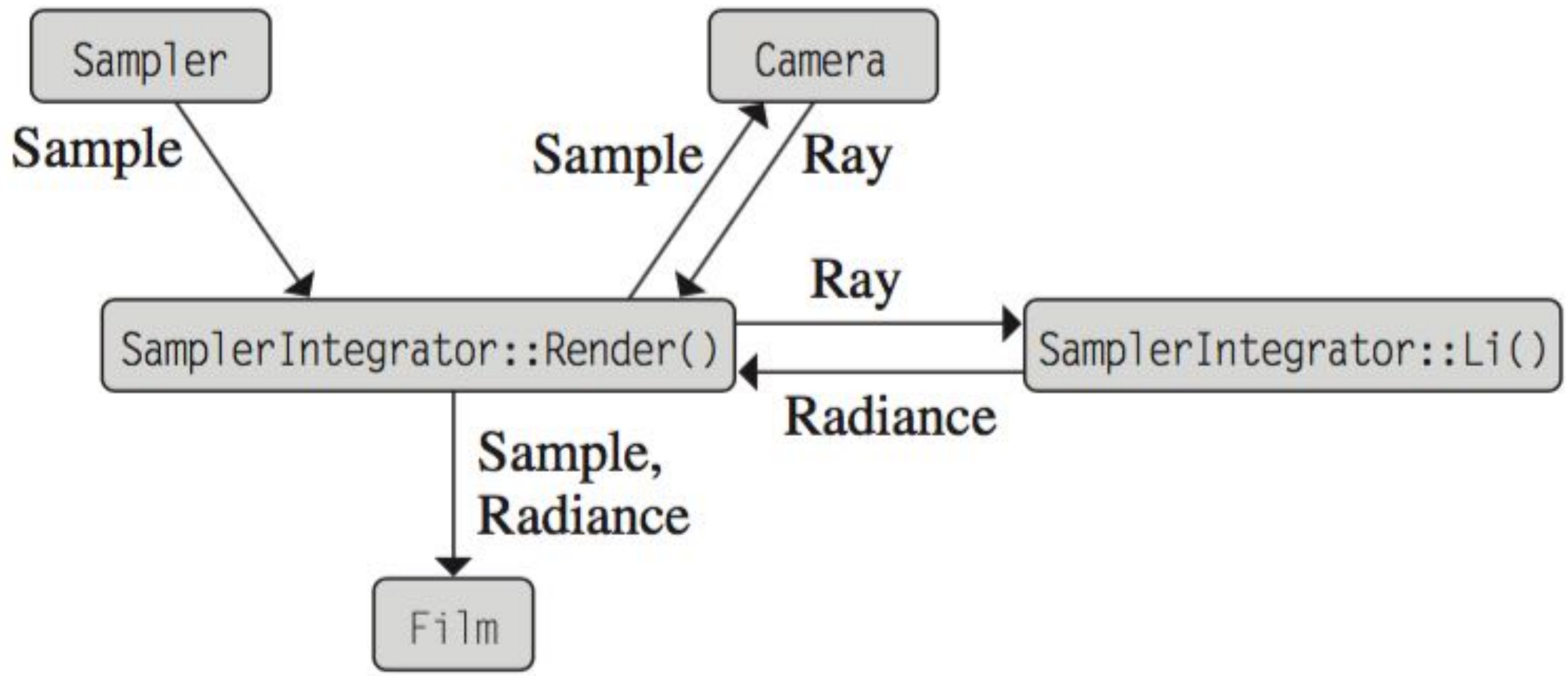


Figure 1.17: Class Relationships for the Main Rendering Loop in the `SamplerIntegrator::Render()` Method in `core/integrator.cpp`. The `Sampler` provides a sequence of sample values, one for each image sample to be taken. The `Camera` turns a sample into a corresponding ray from the film plane, and the `Li()` method implementation computes the radiance along that ray arriving at the film. The sample and its radiance are given to the `Film`, which stores their contribution in an image. This process repeats until the `Sampler` has provided as many samples as are necessary to generate the final image.

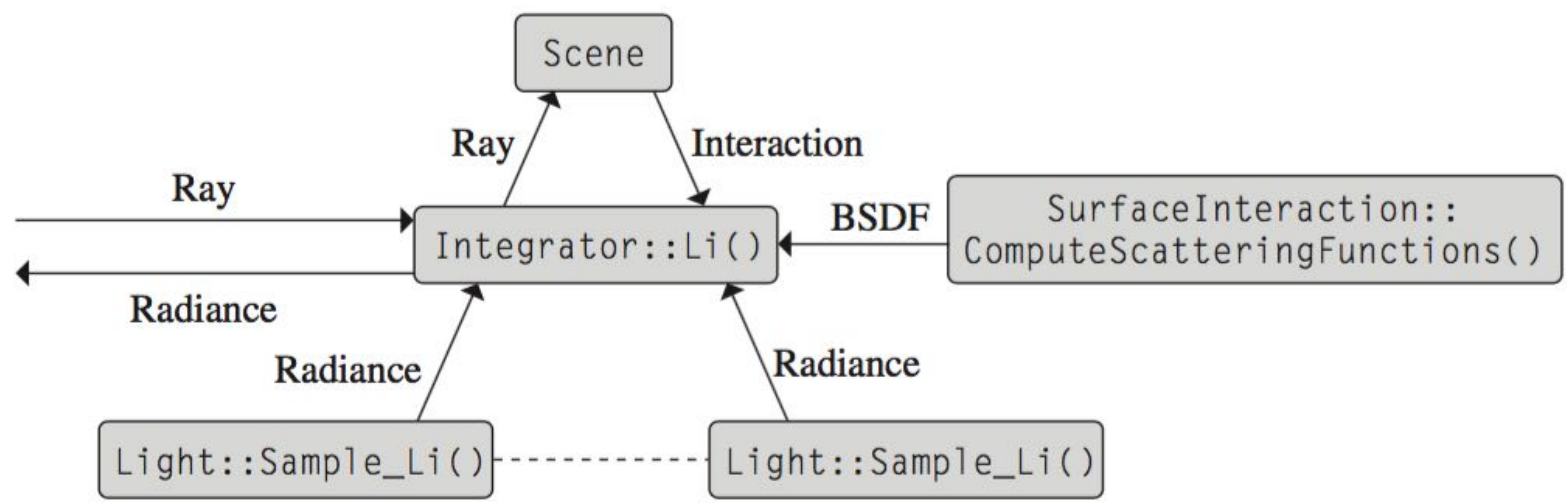


Figure 1.19: Class Relationships for Surface Integration. The main rendering loop in the `SamplerIntegrator` computes a camera ray and passes it to the `Li()` method, which returns the radiance along that ray arriving at the ray's origin. After finding the closest intersection, it computes the material properties at the intersection point, representing them in the form of a BSDF. It then uses the Lights in the Scene to determine the illumination there. Together, these give the information needed to compute the radiance reflected back along the ray at the intersection point.

Shape Interface (Simplified)

```
class Shape {
  public:
    Bounds3f ObjectBound() const;
    Bounds3f WorldBound() const;
    bool Intersect(const Ray &ray, Float *tHit,
                   SurfaceInteraction *isect,
                   bool testAlphaTexture) const;
    bool IntersectP(const Ray &ray,
                    bool testAlphaTexture);
    Float Area() const;
    // …
};
```

Surface Interaction (Simplified)

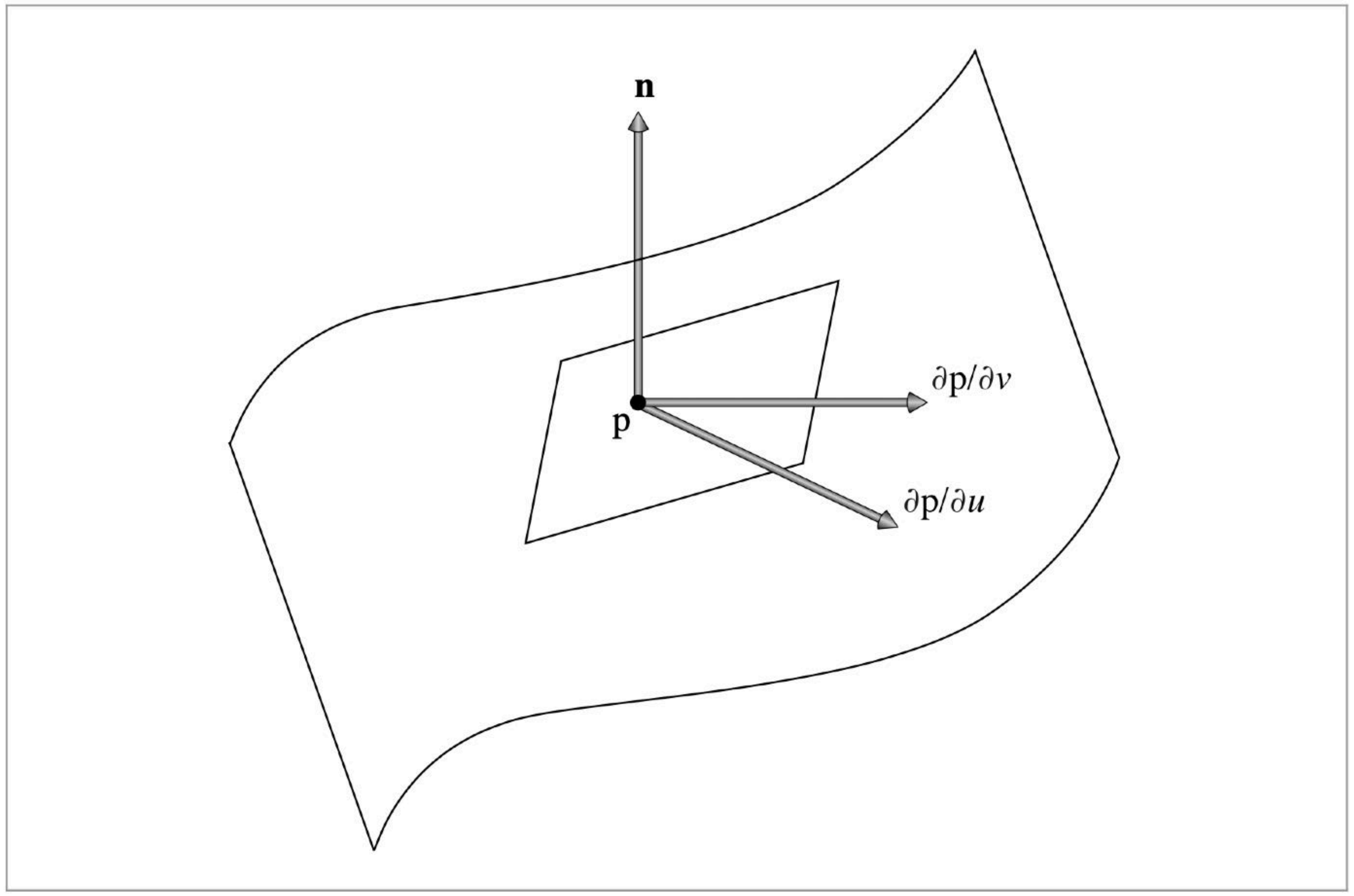
Information about the surface point hit by a ray.

```
class SurfaceInteraction {
    Point3f p;
    Normal3f n;

    Point2f uv;
    Vector3f dpdu, dpdv;
    Normal3f dndu, dndv;

    struct {
        Normal3f n;
        Vector3f dpdu, dpdv;
        Normal3f dndu, dndv;
    } shading;

    // …
};
```



Primitives in PBRT

pbrt `Primitive` base class

- `Shape`
- `Material` (for a later class)

```
class Primitive {
  public:
    virtual Bounds3f WorldBound() const = 0;
    virtual bool Intersect(const Ray &r,
                           SurfaceInteraction *) const = 0;
    virtual bool IntersectP(const Ray &r) const = 0;
    virtual const AreaLight *GetAreaLight() const = 0;
    virtual const Material *GetMaterial() const = 0;
    virtual void ComputeScatteringFunctions(…) const = 0;
};
```

Primitives

Collections

- `TransfomedPrimitive`: Transformation + primitive
- `Aggregate`
  - Treat acceleration data structures as primitives
  - Two types of accelerators: `kdtree.cpp`, and `bvh.cpp`
  - May nest accelerators of different types

```
class Scene {
    // …
    bool Intersect(const Ray &ray,
                   SurfaceInteraction *isect) const {
        return aggregate->Intersect(ray, isect);
    }
    std::shared_ptr<Primitive> aggregate;
};
```