Raysect

Python Scientific Ray-tracing Framework

Dr Alex Meakins

1st October 2015

What is Raysect?

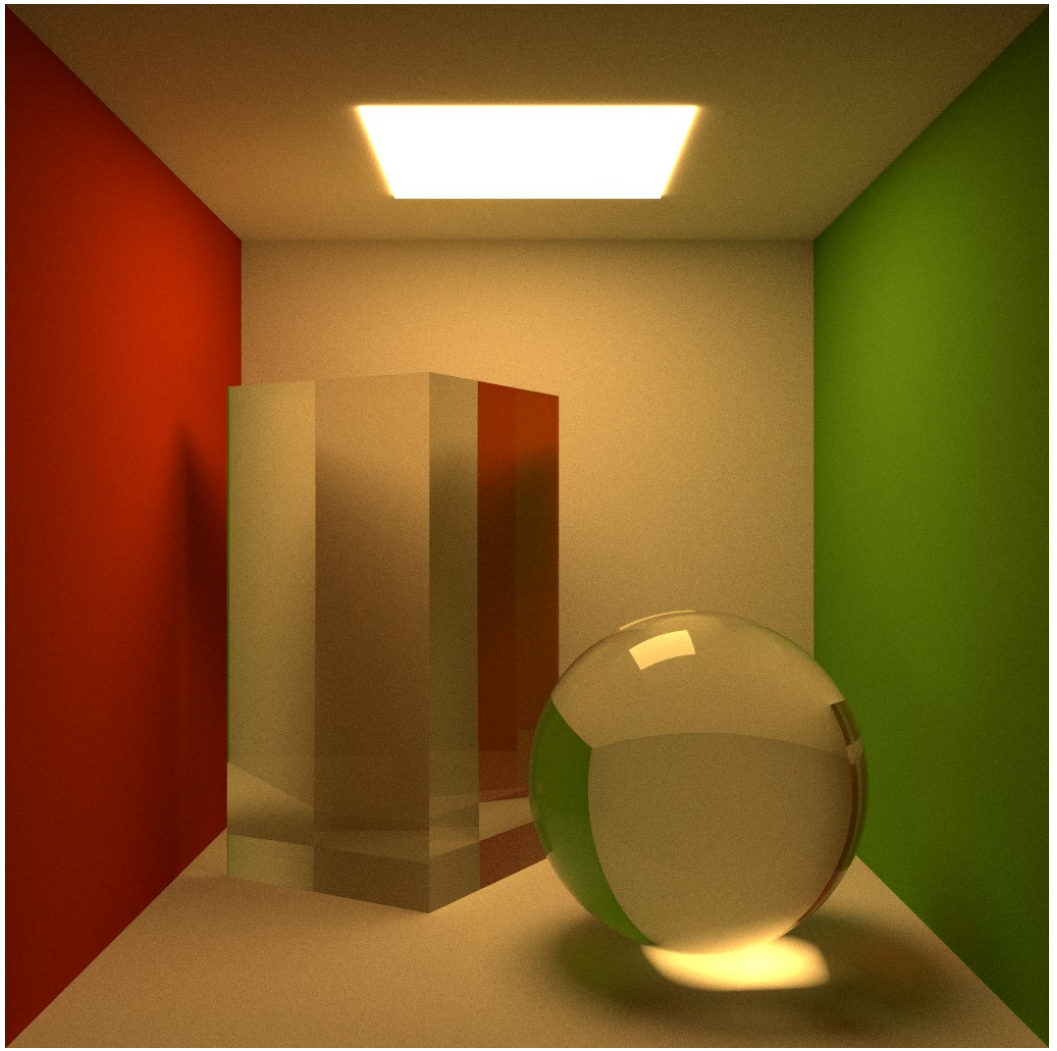
- A ray-tracing framework for Python
  - built for scientific research
    - high precision, fully spectral
  - easy to extend
    - philosophy: ease of use trumps speed, but speed matters
  - fully open license (BSD)
    - can be embedded in your code commercially
    - www.raysect.org / https://github.com/raysect/source
  - developers
    - Dr Alex Meakins and Dr Matthew Carr

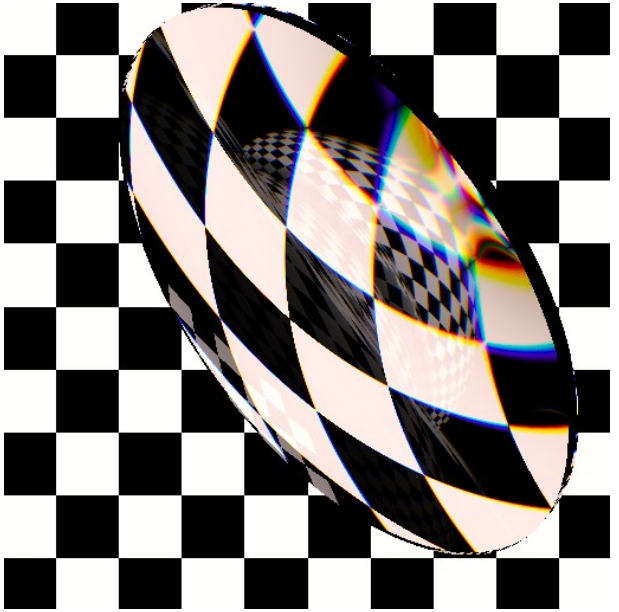
What is a Ray Tracer?

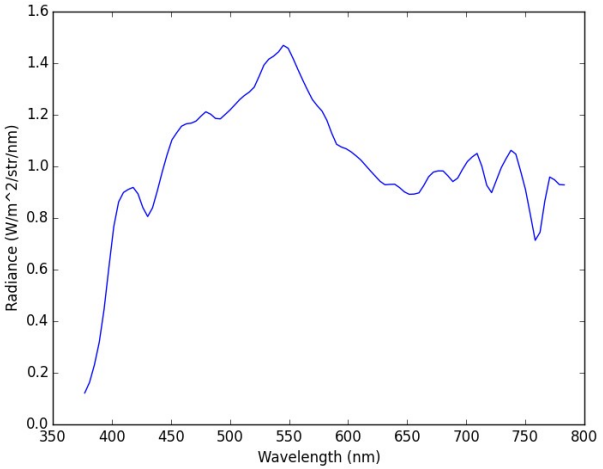
- Algorithm for simulating light propagation
  - geometric optical model of light
    - light modelled as a collection of rays
    - rays follow a straight path unless they interact with a medium
      - e.g. glass, metal, light source

- Used to simulate optical diagnostics
  - forward modelling, diagnostic optimisation

Example





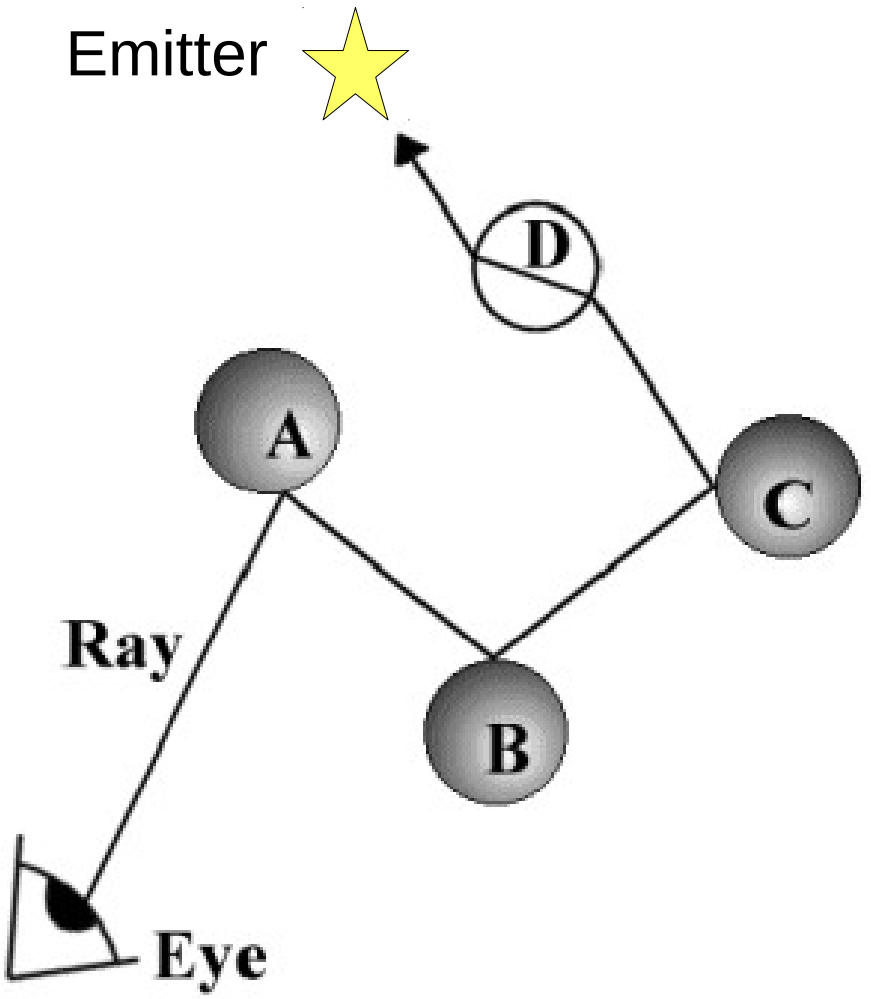

Radiance (W/m^2/str/nm)
Wavelength (nm)
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
350
400
450
500
550
600
650
700
750
800

Ray tracing Algorithm

- Desire intensity/spectrum of light reaching observer
  - e.g. camera pixel
  - sample light (radiance) along paths reaching observer
  - accumulate samples to obtain intensity

- Sampling a path
  - trace ray from observer, through material interactions until it reaches a light source
  - propagate spectrum from source through interactions
  - "Path tracing" algorithm

How Does it Work?



Isn't that back-to-front?

- Yes!
  - geometric optics is mathematically reversible
  - emitters generally much larger than observers
  - more computational efficient to trace from observer
    - typically many orders of magnitude

- Some schemes exist that trace from emitter to observer
  - photon tracing
  - bi-directional path-tracing

What does Raysect Do?

- Tracing algorithm is simple to understand, but hard to make efficient
  - requires complex acceleration structures
  - hard to develop and debug
    - £££££££££
- Raysect provides optimised infrastructure for ray-tracing
  - researchers only need to:
    - define a scene containing objects
    - define any material physics required for their work

A Quick Tour

- Raysect provides a number of classes
  - researchers use/build upon these
  - four main classes/concepts:
    - Rays
    - Observers
    - Primitives
    - the scene-graph (World)

- Let's explore these!

Ray Objects

- Represents a ray of light
  - defines a line with an *origin* and *direction*
  - *wavelength range* and number of *spectral samples*
    - centre of range used for refraction calculations

- Implements tracing algorithm
  - `spectrum = ray.trace(world)`
    - causes the ray to start sampling the world
    - returns a Spectrum object
      - samples of spectral radiance: $W/m^2/str/nm$

Observer Objects

- Represents objects that measure light
  - e.g. CCDs, cameras, photo diodes, eyes

- Launch rays and accumulate samples of the scene
  - more convenient that tracing rays manually
    - can place in the world and move around
    - coordinate transforms taken care of
  - observe() method triggers ray-tracing
    - `camera.observe()`

Primitive Objects

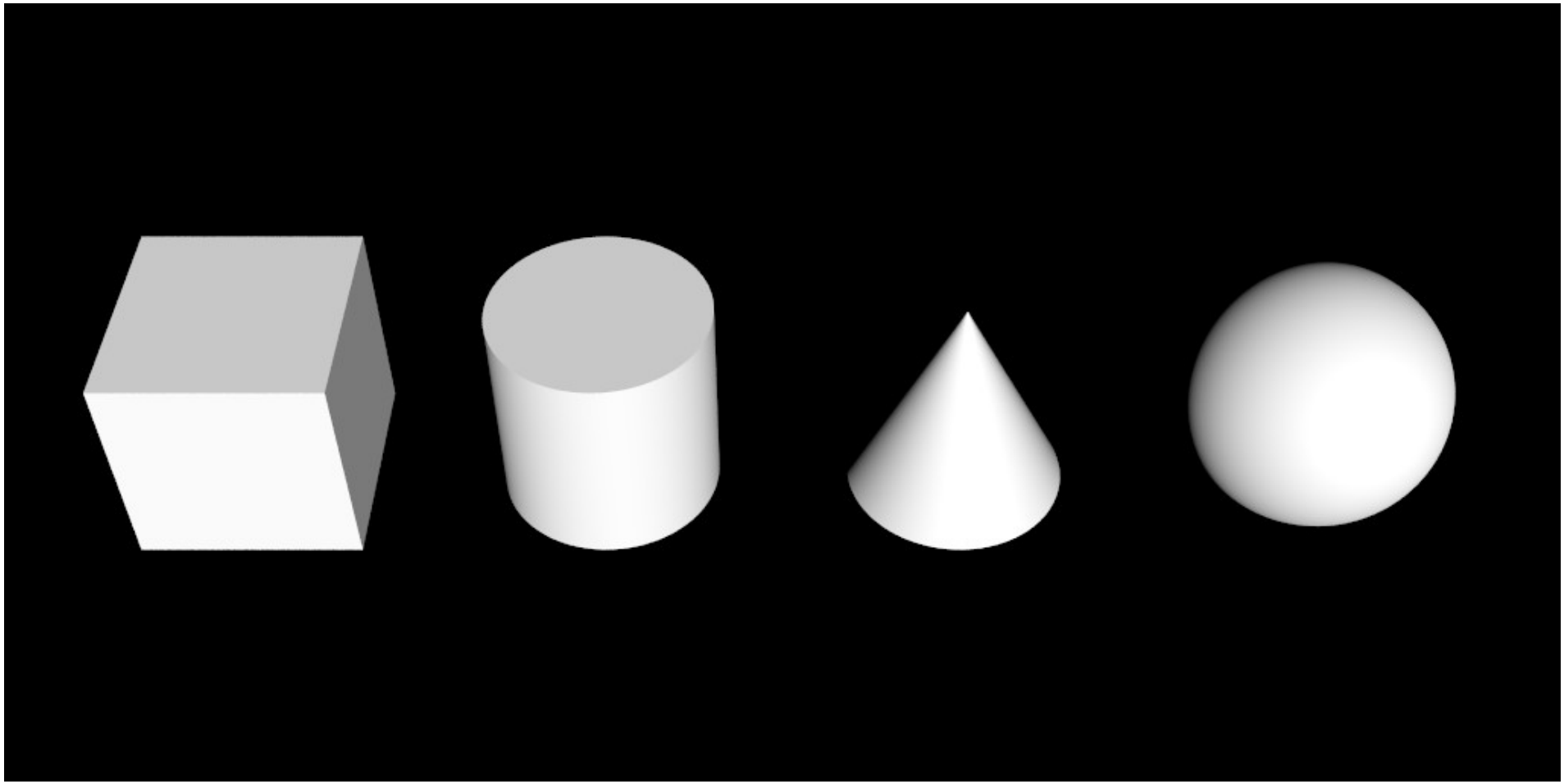
- Primitives are geometric objects that rays interact with
  - e.g light sources, lenses, mirrors, diffuse surfaces

- Mathematically defined surfaces
  - open or closed
    - closed surfaces define a volume
  - assigned materials
    - e.g. glass, metal, emitter
    - surface and volume materials

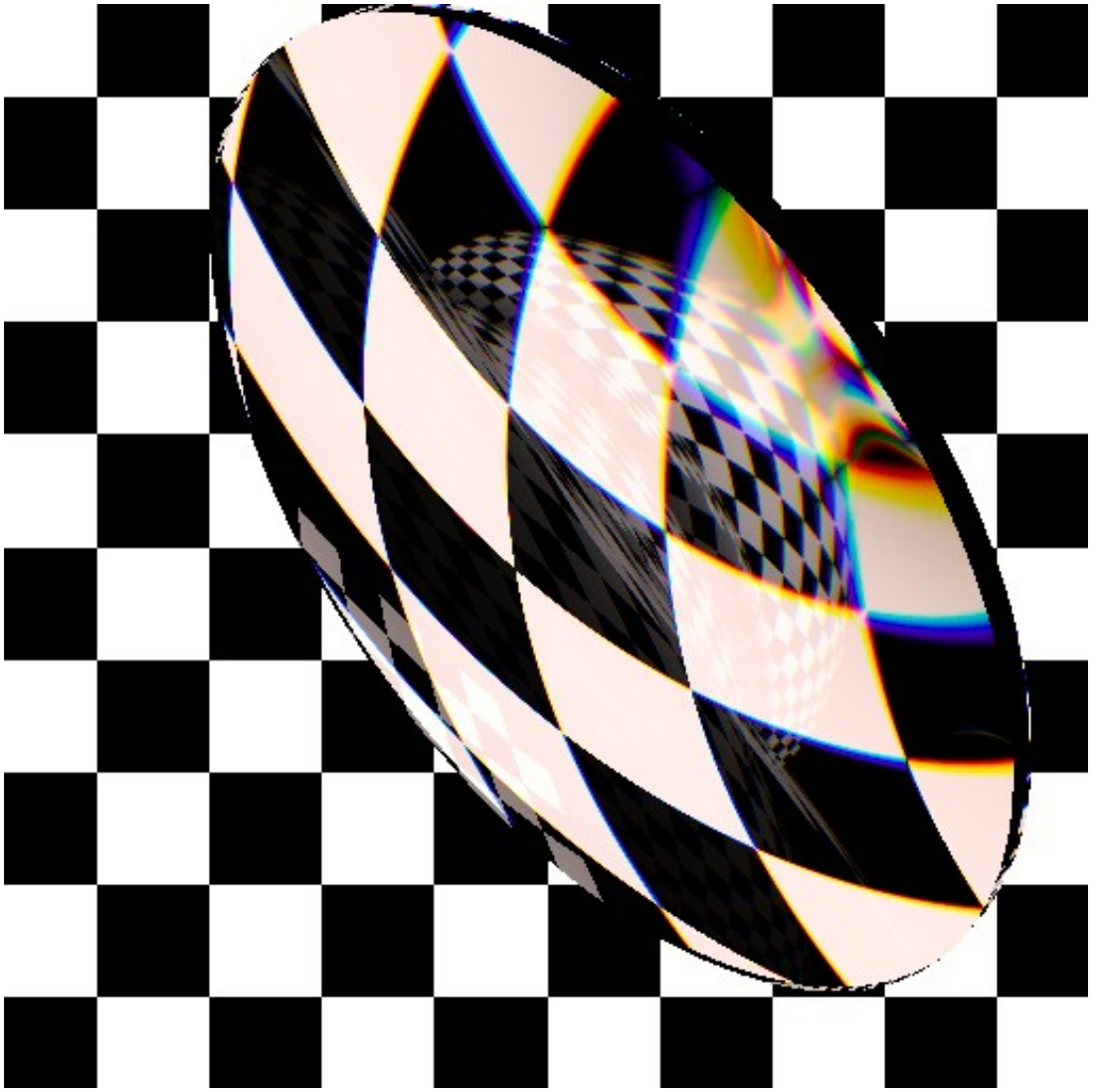
Raysect Primitives

- Basic solids
  - sphere, box, cylinder, cone
- Constructive Solid Geometry Operators
  - union, intersect, subtract
- Meshes
  - tri-poly meshes
  - importers for STL and OBJ
  - optimised for millions of polygons
  - support instancing

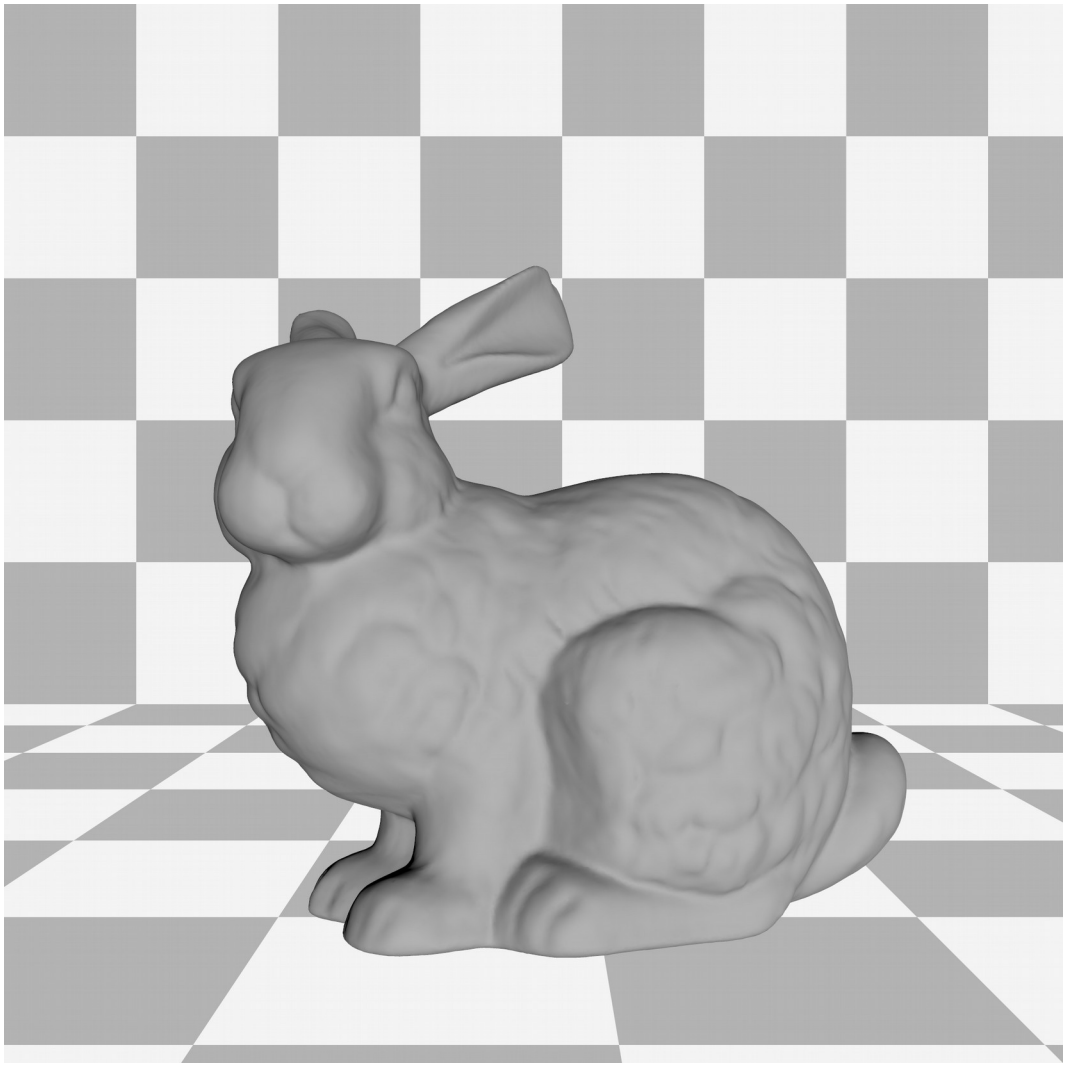
Basic Primitives



Constructive Solid Geometry



Meshes



Ray-Primitive Interaction

- Ray encounters primitive
  - point of closest intersection calculated
    - hit position and surface normal
  - surface material calculation performed

```
spectrum = primitive.material.evaluate_surface(...)
```

    - how spectrum is obtained up to material
  - if ray travelling inside volume
    - volume contribution to path calculated

```
spectrum = primitive.material.evaluate_volume(spectrum, ...)
```

      - emission, attenuation, scattering

Materials

- Raysect provides a growing material catalogue
  - Glasses, Metals, Diffuse, Emitters, Modifiers

- To ease development, material base-classes provided:
  - `NullSurface, NullVolume`
  - `SurfaceEmitter, UniformVolumeEmitter, VolumeEmitterHomogeneous, VolumeEmitterInHomogeneous`
  - generic BRDF base classes in development

Emitters and Glasses

Metals and Diffuse Surfaces

Scene Graph

- Primitives and Observers defined in their own local coord system
  - need to be placed into the “world”
  - need a system to keep track of the locations/co-ordinate transforms

- Scene-graph
  - tree structure consisting of nodes
    - nodes are primitives, observers
  - each node has an associated coordinate space
    - translated/rotated relative to it's parent

The World

- World is the root node of the scene-graph
  - all primitives and observers must be attached to World

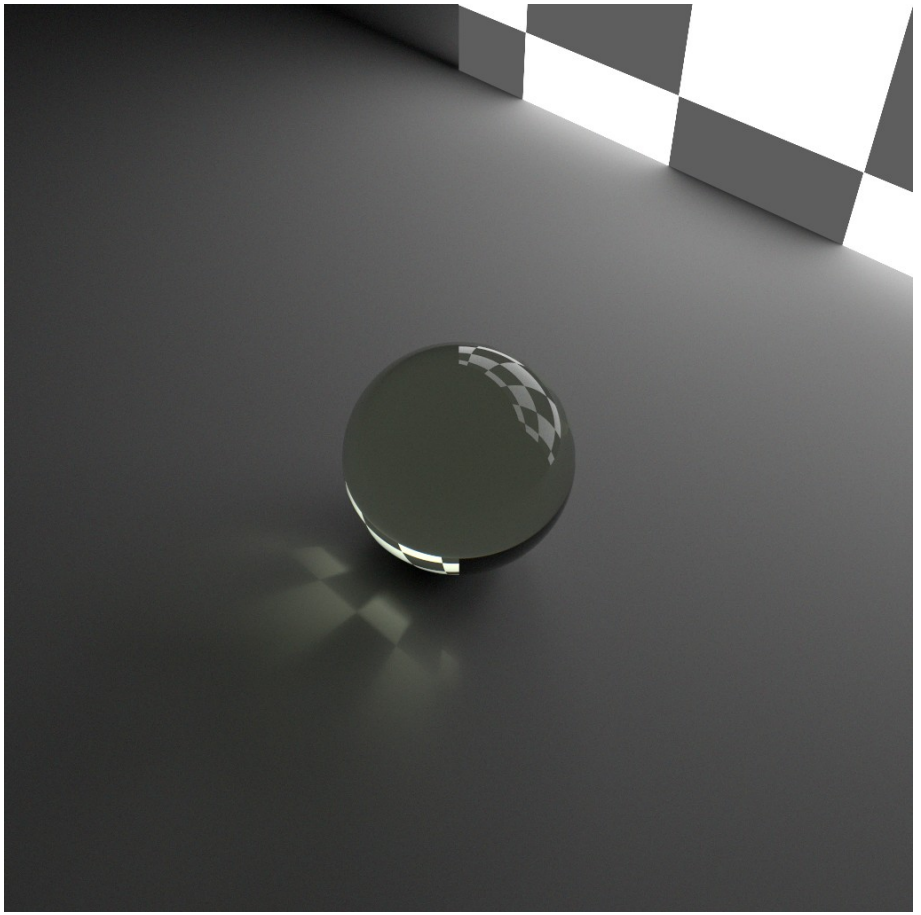
- Adding nodes to the world
  - nodes are parented to another node (e.g. World)
  - nodes are given a transform relative to parent
    - e.g. a translation and/or rotation
  - Can build hierarchies of objects
    - manipulate whole group with one transform

Typical Work-flow

1) Set-up your primitives and observers
   - define materials, set-up camera etc...
2) Assemble the scene-graph
   - link primitives and observers to the World
   - set transforms
3) Call observe() on an Observer or trace a ray manually

Example

- This is what we are going to produce:



The Scene

- Primitives:
  - Ground
  - Checker-board emitter
  - Glass sphere
- Observers:
  - Pinhole camera
- Will skip some finer details for brevity

Add primitives

```
# primitives
sphere = Sphere(radius=1.5,
                material=schott("N-BK7"))

ground = Box(lower=Point(-50, -0.01, -50),
             upper=Point(50, 0, 50),
             material=Lambert())

emitter = Box(lower=Point(-10, -10, 0),
              upper=Point(10, 10, 0.1),
              material=Checkerboard(4, d65_white, d65_white, 0.1, 2.0))
```

Add Observer

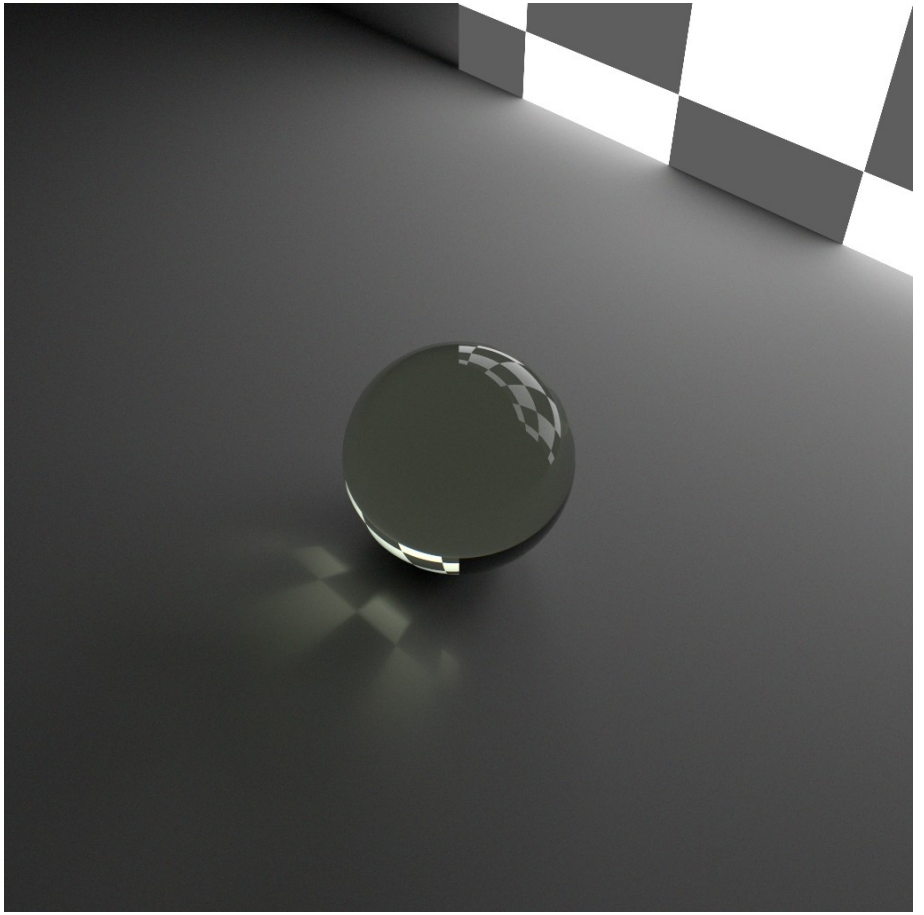
```
# camera
camera = PinholeCamera()

camera.fov = 45
camera.pixels = (512, 512)
camera.pixel_samples = 250
camera.sub_sample = True

camera.rays = 1
camera.spectral_samples = 20
camera.ray_min_depth = 3
camera.ray_max_depth = 100
camera.ray_extinction_prob = 0.1

camera.display_progress = True
camera.display_update_time = 10
```

Build Scene-graph

```
# scenegraph
world = World()

sphere.parent = world
sphere.transform = translate(0, 0.0001, 0) * rotate(20, 0, 0)

ground.parent = world
ground.transform = translate(0, -1.5, 0)

emitter.parent = world
emitter.transform = rotate(45, 0, 0) * translate(0, 0, 10)

camera.parent = world
camera.transform = translate(0, 10, -10) * rotate(0, -45, 0)
```

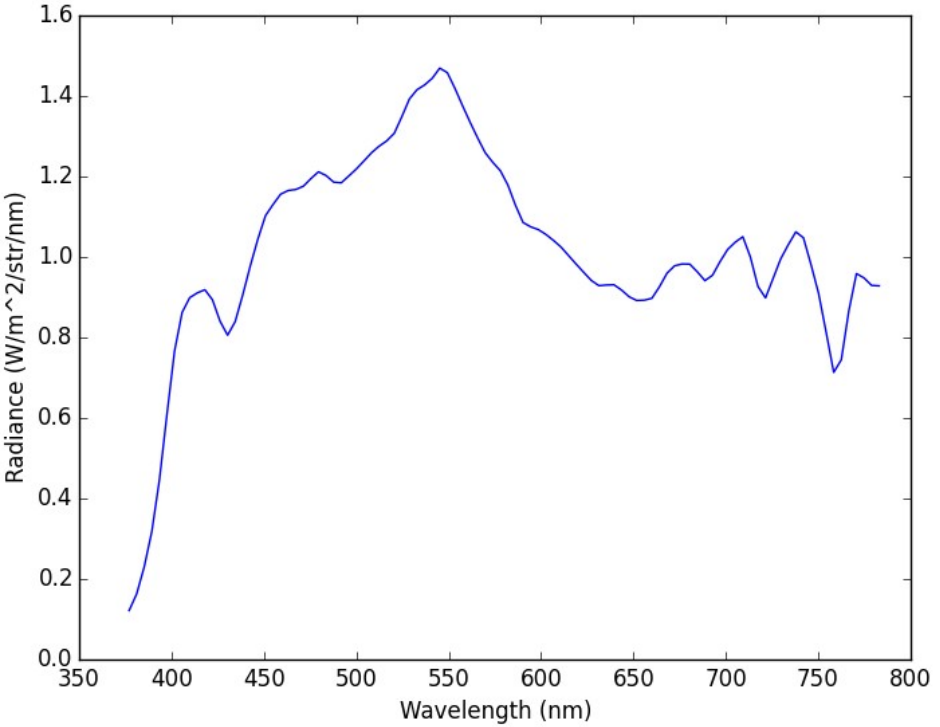
Observe!

```
# observe
ion()
camera.observe()

ioff()
camera.save("render.png")
camera.display()
```

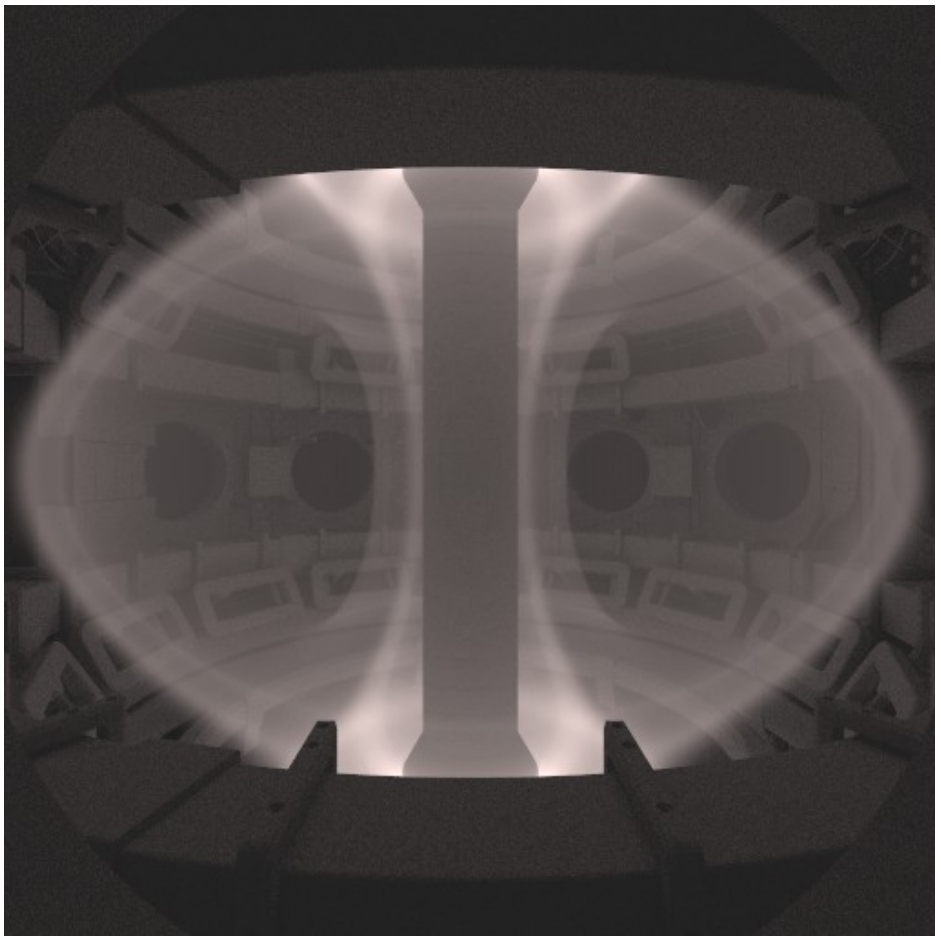


Looking at Spectrum

```
ray = Ray(origin=Point(0, 0, -5),
          direction=Vector(0, 0, 1),
          min_wavelength=375,
          max_wavelength=785,
          num_samples=100)

spectrum = ray.trace(world)
```



Fusion Example: Wall Reflections



- Notes about image:
  - emission from SOLPS modelling code
  - technology test, no realistic materials
    - mesh is perfect Lambertian material
    - SOLPS artifact causes inner “ring” inside plasma
    - mesh incomplete

Summary

- Raysect
  - A ray-tracing framework for Python
  - Provides optimised classes for researchers
  - open source
    - BSD licensed
    - contributions very welcome

- Any questions?