Photon Propagation Using Open Source Rendering Software : A Study

Keith Evans[1], Antonin Rat[1], Sacha Barre[1], Yangyang Cui[2], Zahra Montazeri[2], Adam Davis[1], Marco Gersabeck[1],

[1]University of Manchester – Department of Physics and Astronomy
[2]University of Manchester – Department of Computer Science

Keith.evans@manchester.ac.uk

Outline

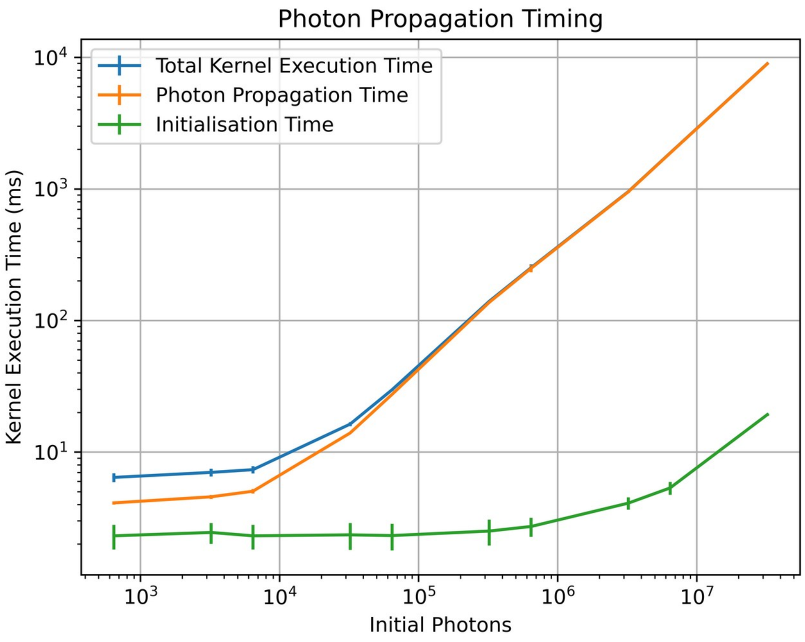
- Motivation
- Propagation vs Rendering
- Mitsuba
  - What
  - How
  - Why
- Initial Questions and Results
- Outline of Pipeline
- Current Progress and results
- Conclusion

Motivation

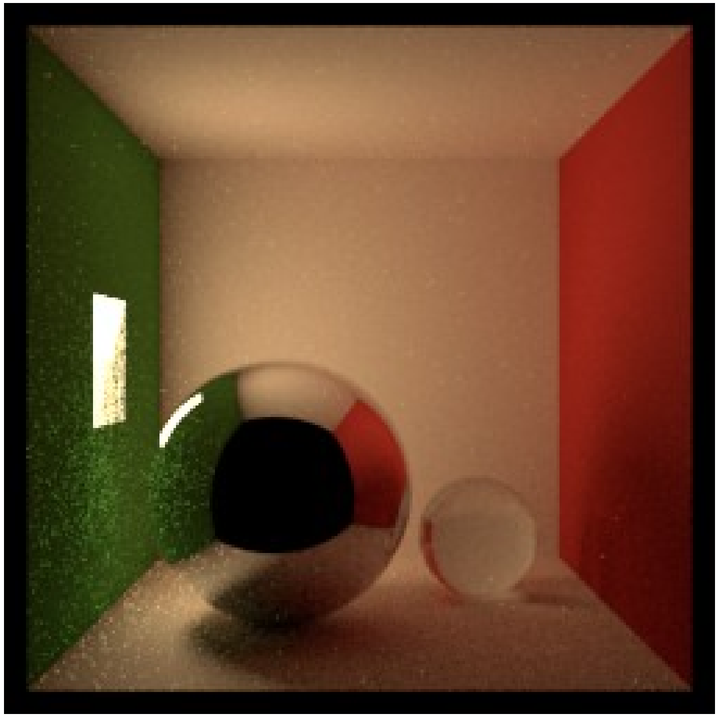
- Computational requirements for LHCb increasing year on year
- RICH contribution signification due to cost of photon propagation
- Photon propagation is a cause of computation expense across HEP, Physics more broadly and in Engineering simulations

Propagation and Rendering

- Previously investigated Opticks to propagate photons
- Attended NVIDIA Hackathon Feb 2022
- Learned photon propagation = ray traced rendering



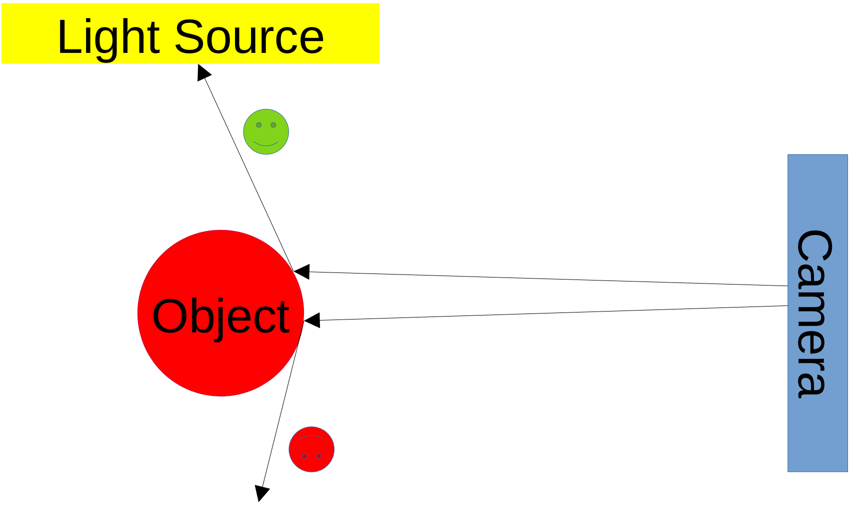
What is Mitsuba?



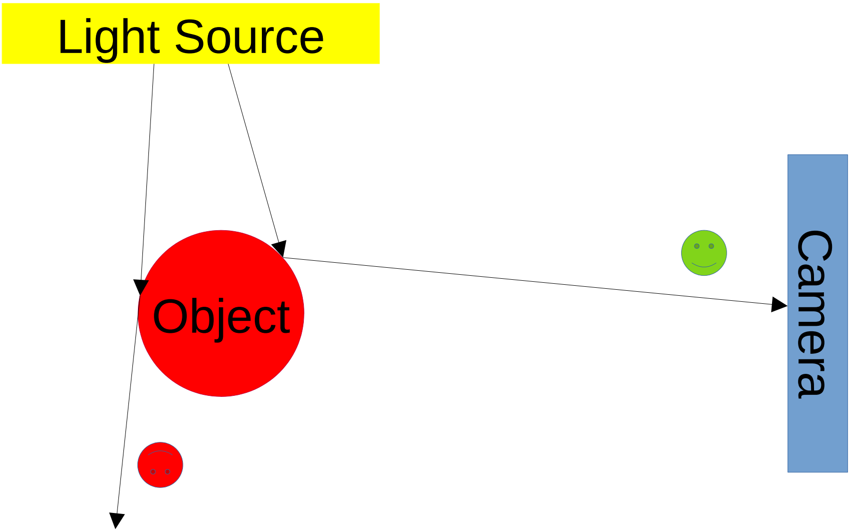
- Industrially recognised “Physically Based Renderer”
- Capable of “Forward”, “Backward” and “Inverse” rendering.
- Uses Monte Carlo sampling to probabilistically determine pixel value/intensity/colour
- Takes scene description in XML of Python format
- Uses “Just In Time” (JIT) compiler to generate optimised and vectorised kernels for CPU/GPU

Mitsuba – How does it work?

Forward Rendering



Backward Rendering



Mitsuba – Why?

- Industrially recognised
- Uses ray traced rendering
  - Ray tracing more accurately represents how light propagates
- Open source
  - Under active development
- Accelerated with LLVM, CUDA and OptiX 7.x
  - Uses Intel Embree for CPU ray tracing
  - Uses NVIDIA OptiX 7.x for CUDA

Links

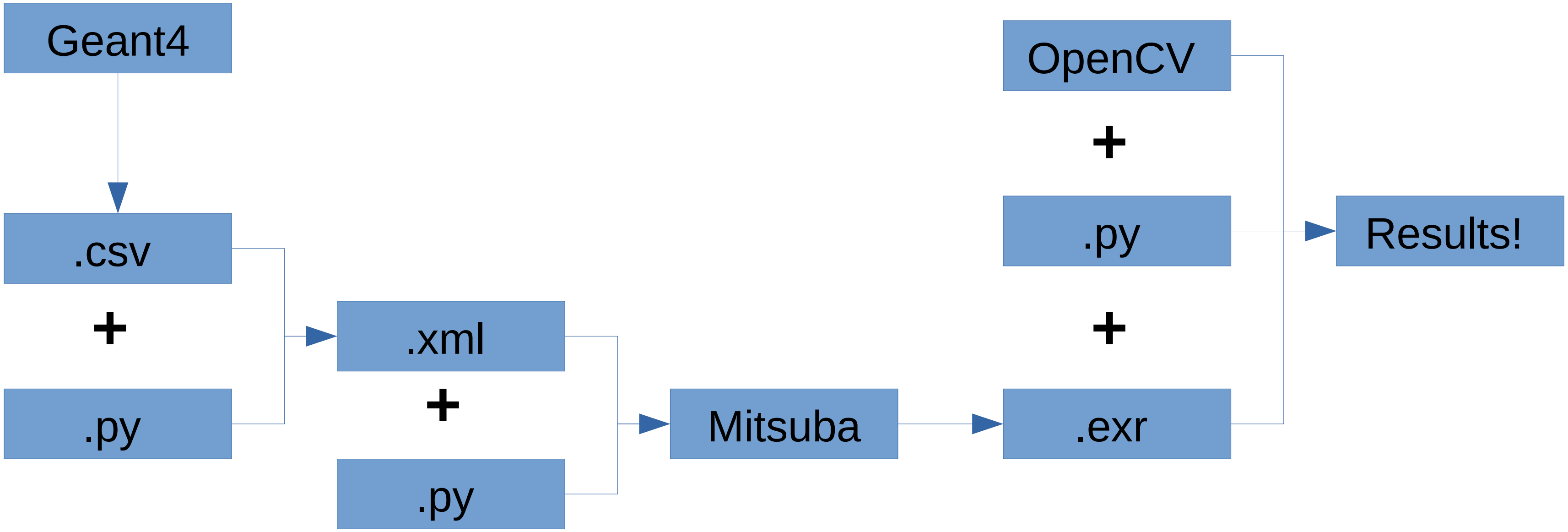
- https://www.mitsuba-renderer.org/
- https://github.com/mitsuba-renderer/mitsuba3
- https://github.com/mitsuba-renderer/drjit

Can we use Mitsuba to propagate photons?

Mitsuba – Can we?

- Can we construct a detector in Mitsuba?
- Can we represent photons in Mitsuba?
- Can we get results out of Mitsuba?

Mitsuba Pipeline



Geant4

.csv

+

.py

.xml

+

.py

Mitsuba

.exr

+

.py

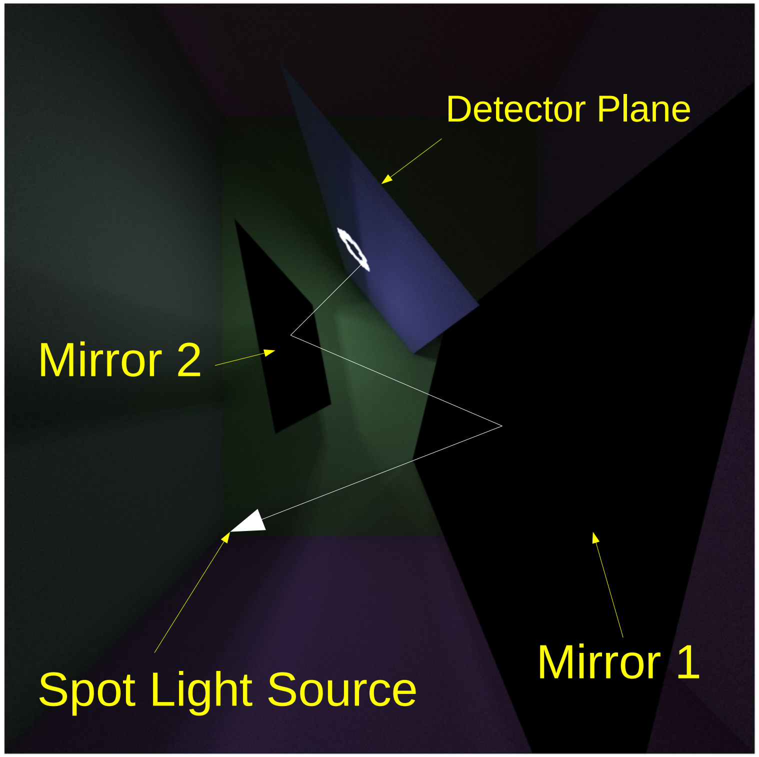
+

OpenCV

Results!

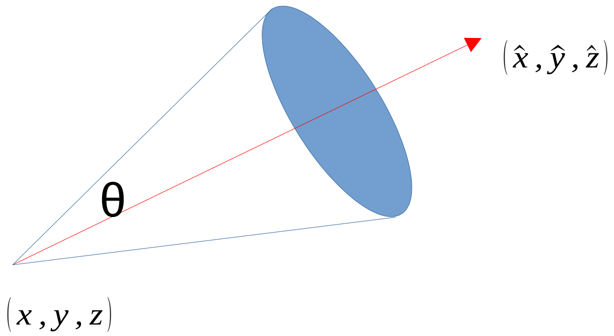
Can we construct a geometry?

- Answer : YES!
- Mitsuba can take either XML or Python geometry input
  - “Variants” are used to specify spectra, integration and parallelisation method
- We built a generic RICH detector
  - Shapes are used to define mirrors and detector planes
  - Bidirectional Scattering Distribution Functions (BSDFs) are used to handle surface scattering
  - Light sources can be defined using “Emitters”



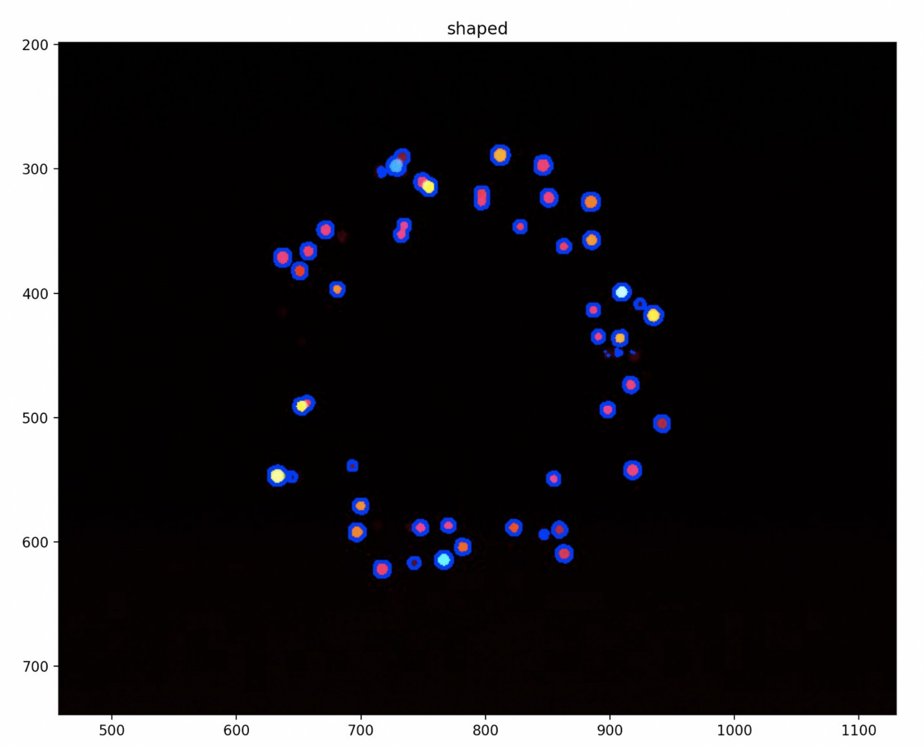
• Can we represent photons in Mitsuba?

- Answer : Kind of
- Mitsuba provides a series of “Emitter” types
- “Spot” is the closest match to actual photon behaviour
- Single position and target direction
- Cut off angle defining ray acceptance
- 1 emitter / photon
- Custom emitter (in progress) possible thanks to open source code



Can we get results out of Mitsuba?

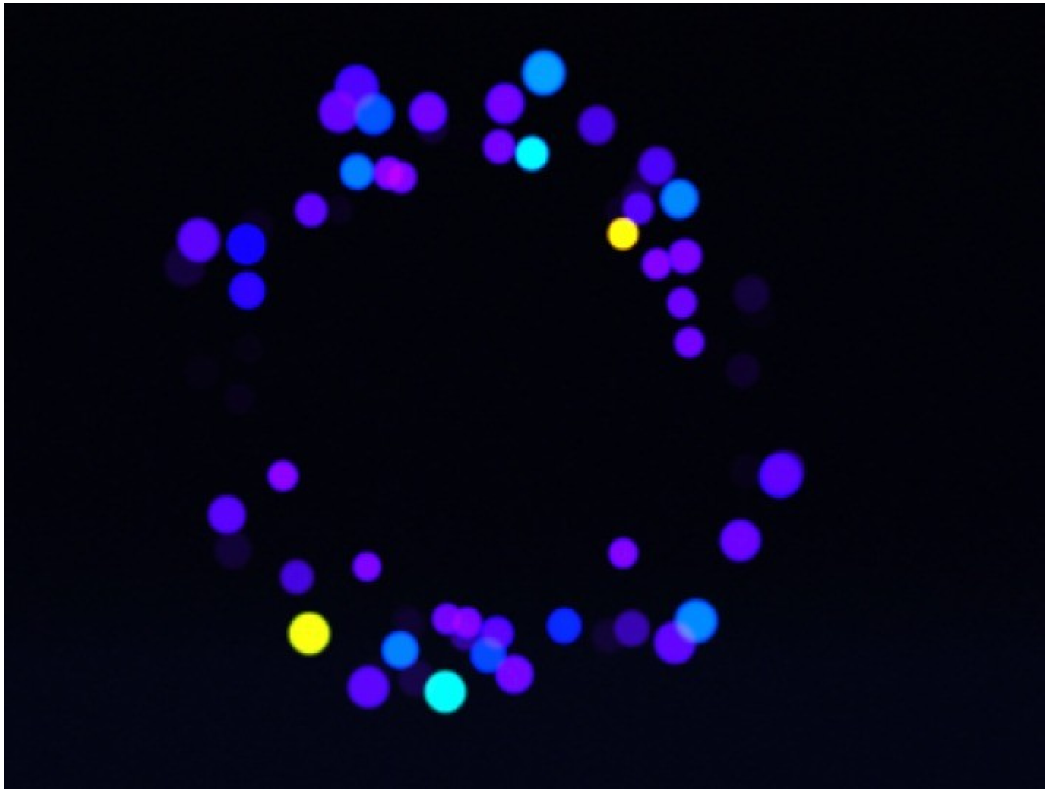
- Answer : YES!
- Mitsuba outputs EXR format as standard
  - EXR is a standard HDR image format
- We can use OpenCV to fit to the images and return hits



100 G4 Photons – Scalar RGB – Detector Plane

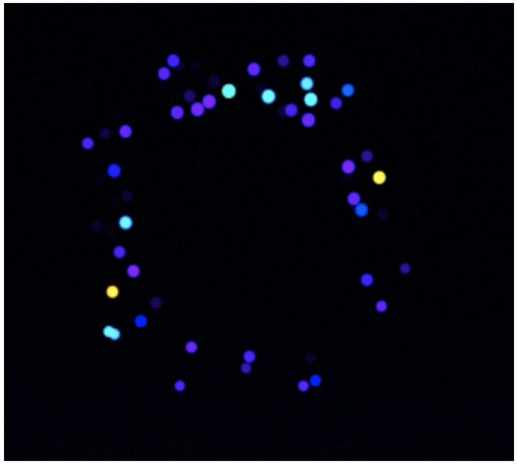
Results

- G4 Photons propagated through generic RICH
- Image contains UV photons
  - Mitsuba cannot assign RGB values to UV photons
  - UV photons ARE propagated but not visualised
- Photon “spots” are different sizes due to differing path lengths
  - Solved with custom emitter



100 G4 Photons – Scalar RGB – Detector Plane

Road to Custom Emitter



- Cut off now inversely proportional to path length
- Detector quantum efficiency applying prior propagation to reduce computation expense

100 Photons - Scaler RGB - Cut-off Angle Adjusted

- Mono variants “ignore” wavelength instead registering binary hits
- Variants can be passed at runtime avoiding expensive recompilation

100 Photons - Scaler Mono - Cut-off Angle Adjusted

Proof of Concept : Lessons Learned

- Spot emitter needs modifying to remove cut off angle
    - Also sampling needs to be turned off
    - Vector input to custom emitter also essential
- RGB Spectrum not suitable due to UV photons
    - Solved with mono variants
- Pass G4 data “directly” to Mitsuba
    - Already demonstrated by Mitsuba binary

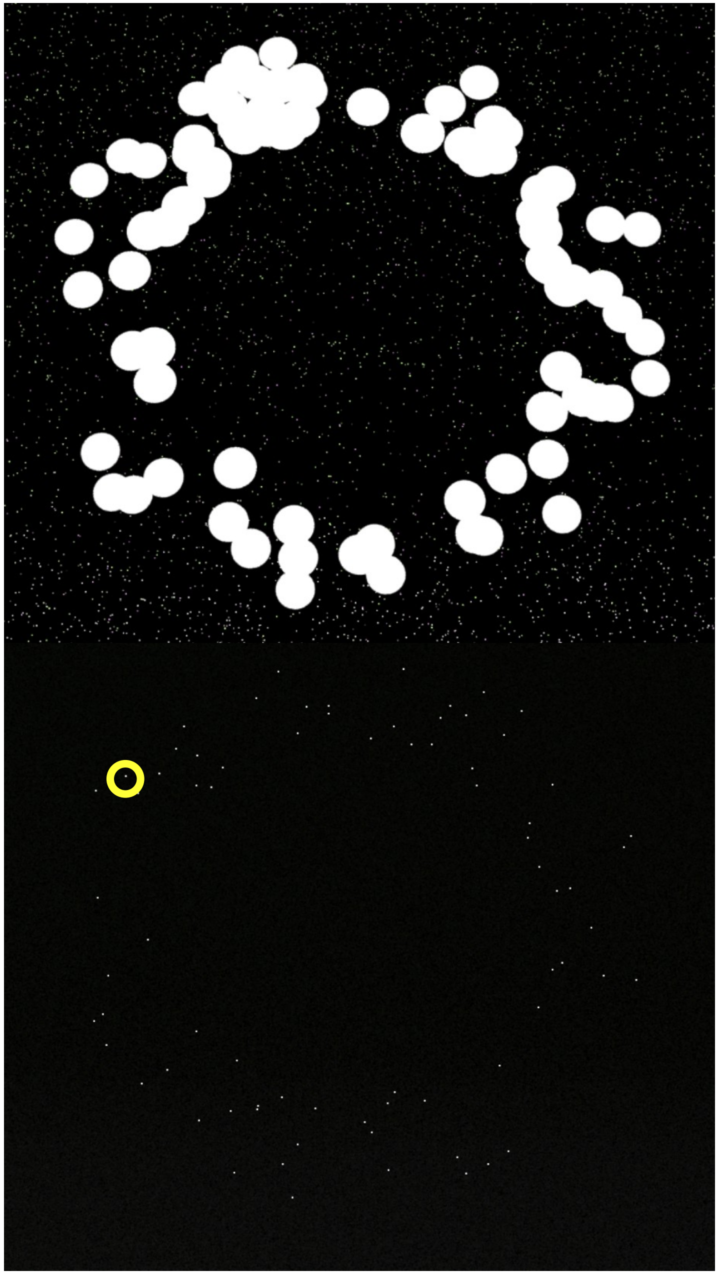
Prototype : Questions

- Can we create a “photon” emitter class?
- How does Mitsuba perform?
- Can we validate Mitsuba results against G4?

MANCHESTER
1824
The University of Manchester

Can we create a “photon” emitter?

- Answer : YES!
- Modified version of the spot emitter
  - The “photon_emitter” fires a single ray with a fixed direction
  - No cut off angle
- Still requires 1 instantiation / photon
  - Working towards vector input emitter



How does Mitsuba perform?

CPU = Intel Xeon 4210R @ 2.40Ghz and 20 Cores

- G4
  ~ 4,000 $\mathrm{Pps}^{-1}\mathrm{t}^{-1}$
- Mitsuba (llvm_mono & 16 samples and suboptimal optimisation)
  ~ 2,000 $\mathrm{Pps}^{-1}\mathrm{t}^{-1}$

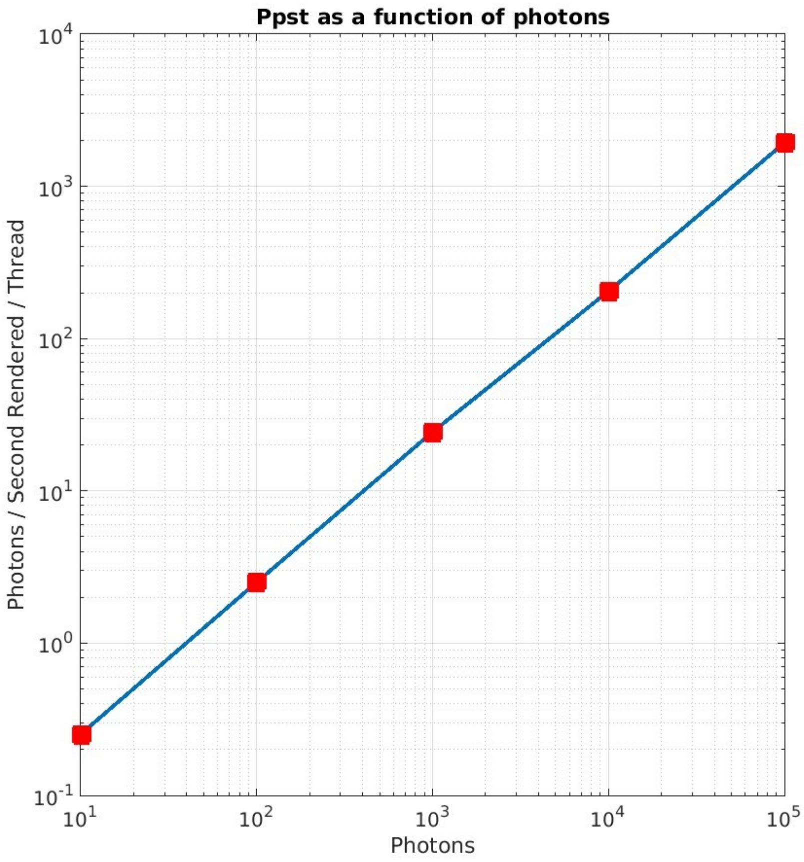
GPU(s) = 2 x Nvidia Tesla T4 @ 1.590Ghz, 2560 CUDA cores, 40 RT cores

- G4 + Opticks
  ~ 500,000 $\mathrm{Pps}^{-1}\mathrm{g}^{-1}$ (195 $\mathrm{Pps}^{-1}\mathrm{t}^{-1}$)
- Mitsuba (cuda_mono & 16 samples and suboptimal optimisation)
  ~ 500,000 $\mathrm{Pps}^{-1}\mathrm{g}^{-1}$ (195 $\mathrm{Pps}^{-1}\mathrm{t}^{-1}$)

*Photons Propagated / Second / Thread ($\mathrm{Pps}^{-1}\mathrm{t}^{-1}$)
*Photons Propagated / Second / GPU ($\mathrm{Pps}^{-1}\mathrm{g}^{-1}$)

Discussion on Performance

- Mitsuba performance data assumes all rendering time is propagation
  - This is false
  - Rendering includes
    - Integration
    - Image Generation
- Tracing scales linearly with samples
  - Spot emitter emits further rays
- Conclusion :
  - Mitsuba is over-propagating
  - Mitsuba includes necessary steps
- Developers have advised us to call the ptracer directly



Can we validate Mitsuba against G4?

- Validation requires a G4 Simulation with Mitsuba offloading
  - This is under construction
- We’ve developed a pipeline to convert between GDML to OBJ
  - Enables experiment independency
- Hoping to complete this process within the next weeks

Summary

- Photon propagation is a general [performance] problem in physics
  - Photon Propagation ≈ Ray Traced Rendering
- Built a working G4 + Mitsuba proof of concept Pipeline
- Working towards an integrated prototype pipeline
- Geometry conversion pipeline in place
- Promising performance results
- Validation is in progress

Future Work

- G4 Integration needs completing
  - G4 – Mitsuba interface needs to be developed
  - Potentially include Mitsuba offloading as an advanced example
- Mitsuba “render” function can be [mostly] skipped
  - Lots of unnecessary functionality
  - Developers have suggested calling “ptracer” functions directly

Thank you!

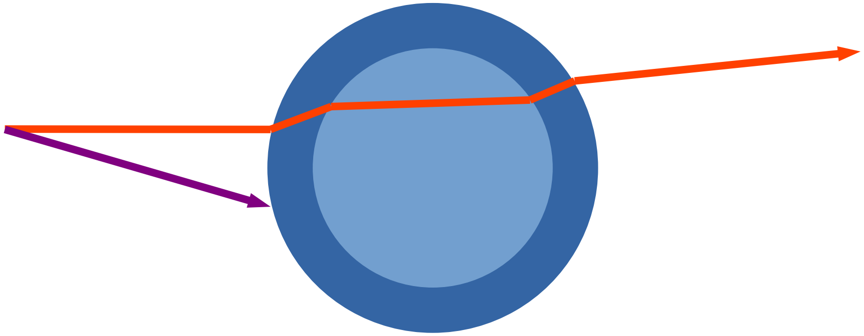
Scene Optimisations

- Mitsuba Workflow
  - XML Parsed, C++ objects instantiated
  - Rendering process is traced
  - Kernel compiled
- Scenes map 1-2-1 emitters to photons
  - Inefficient
    - 10^6 Photons = 10^6 Emitters = 10^6 C++ Objects
  - Solved by Vector input
    - 10^6 Photons = 1 Emitter = 1 C++ Object

Ray Intersection Parallelisation

- Rays only exist for 1 intersection
  - Load Balanced
  - Warp Synchronisation
- Absorbed Rays are killed
- Reflected Rays are “re-emitted”

Track Based Parallelism



Intersection Based Parallelism