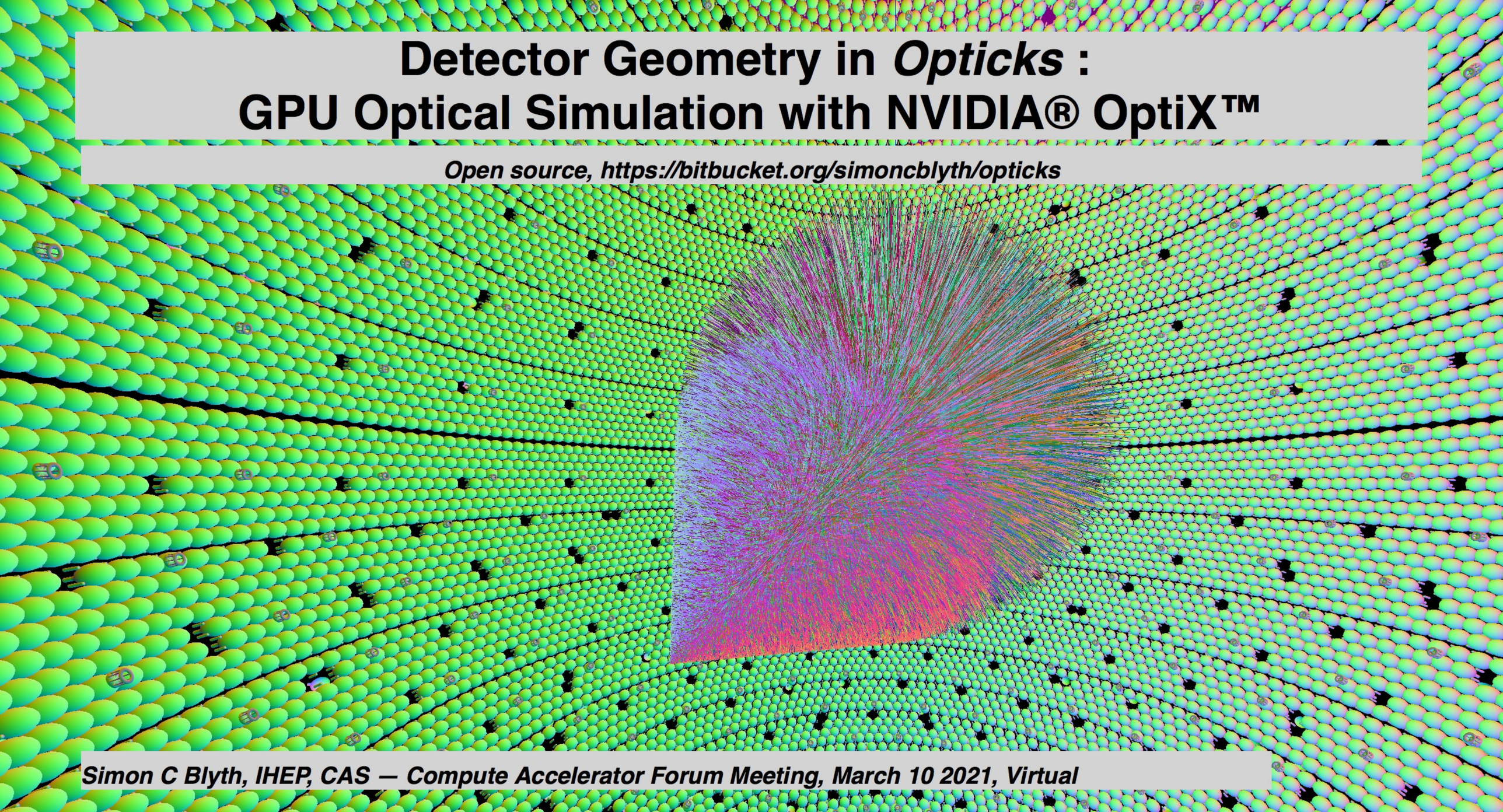
Detector Geometry in *Opticks* : GPU Optical Simulation with NVIDIA® OptiX™

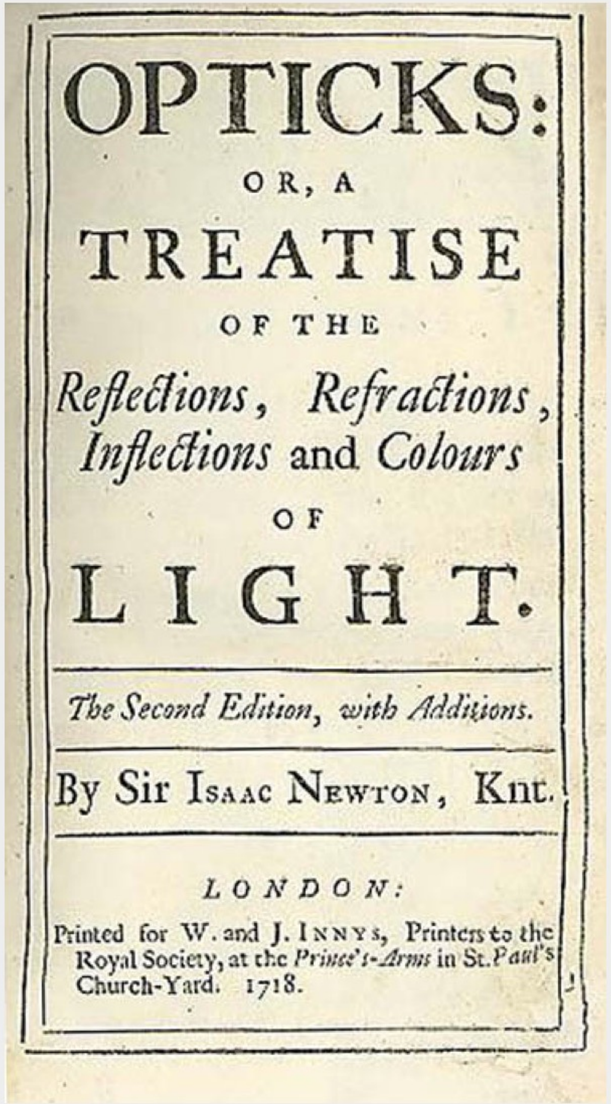
Open source, https://bitbucket.org/simoncblyth/opticks



Simon C Blyth, IHEP, CAS — Compute Accelerator Forum Meeting, March 10 2021, Virtual

Outline

- Introduction
  - JUNO Optical Photon Simulation Problem...
  - NVIDIA Marbles at Night : RTX Demo ; NVIDIA Ampere : 2nd Generation RTX
  - GPU Ray Tracing APIs Converging
  - RTX Execution Pipeline : Common to DirectX RT, VulkanRT, NVIDIA OptiX
  - Spatial Index Acceleration Structure
  - Two-Level Hierarchy : Instance transforms (TLAS) over Geometry (BLAS)
- Opticks : Structural Geometry
  - Geant4 + Opticks Hybrid Workflow
  - NPY Serialization : Fundamental to Opticks Geometry Model ; NumPy Example
  - Translation 1st Step : Geant4 -> Opticks/GGeo : 1->1 conversions
  - Translation 2nd Step : Opticks/GGeo Instancing : "Factorized" Geometry
  - Ray Intersection with Transformed Object -> Geometry Instancing
  - OpenGL Mesh Instancing ; OptiX Ray Traced Instancing
- Opticks Solids : CSG, Constructive Solid Geometry
  - G4VSolid -> CUDA Intersect Functions for ~10 Primitives
  - G4Boolean -> CUDA/OptiX Intersection Program Implementing CSG
  - (CSG details relegated to "Extras")
- Opticks Material/Surface Properties : Boundary Texture
- Opticks vs Geant4 : Extrapolated G4 times compared to Opticks with RTX ON/OFF
- Overview + Links


OPTICKS:

OR, A

TREATISE

OF THE

Reflections, Refractions, Inflections and *Colours*

OF

LIGHT.

The Second Edition, with Additions.

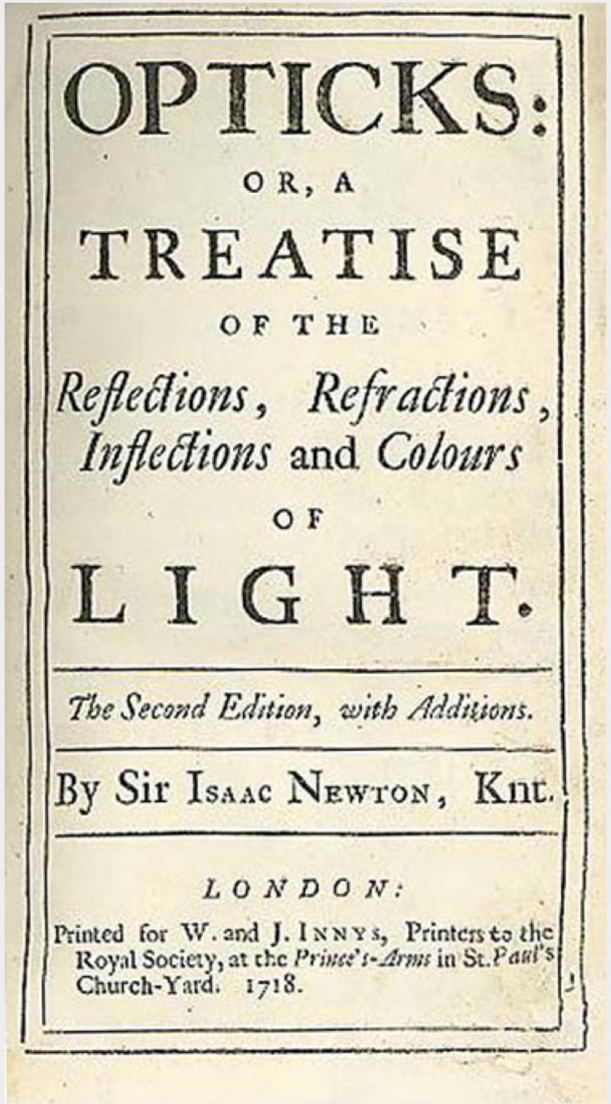
By Sir Isaac Newton, Knt.

LONDON:

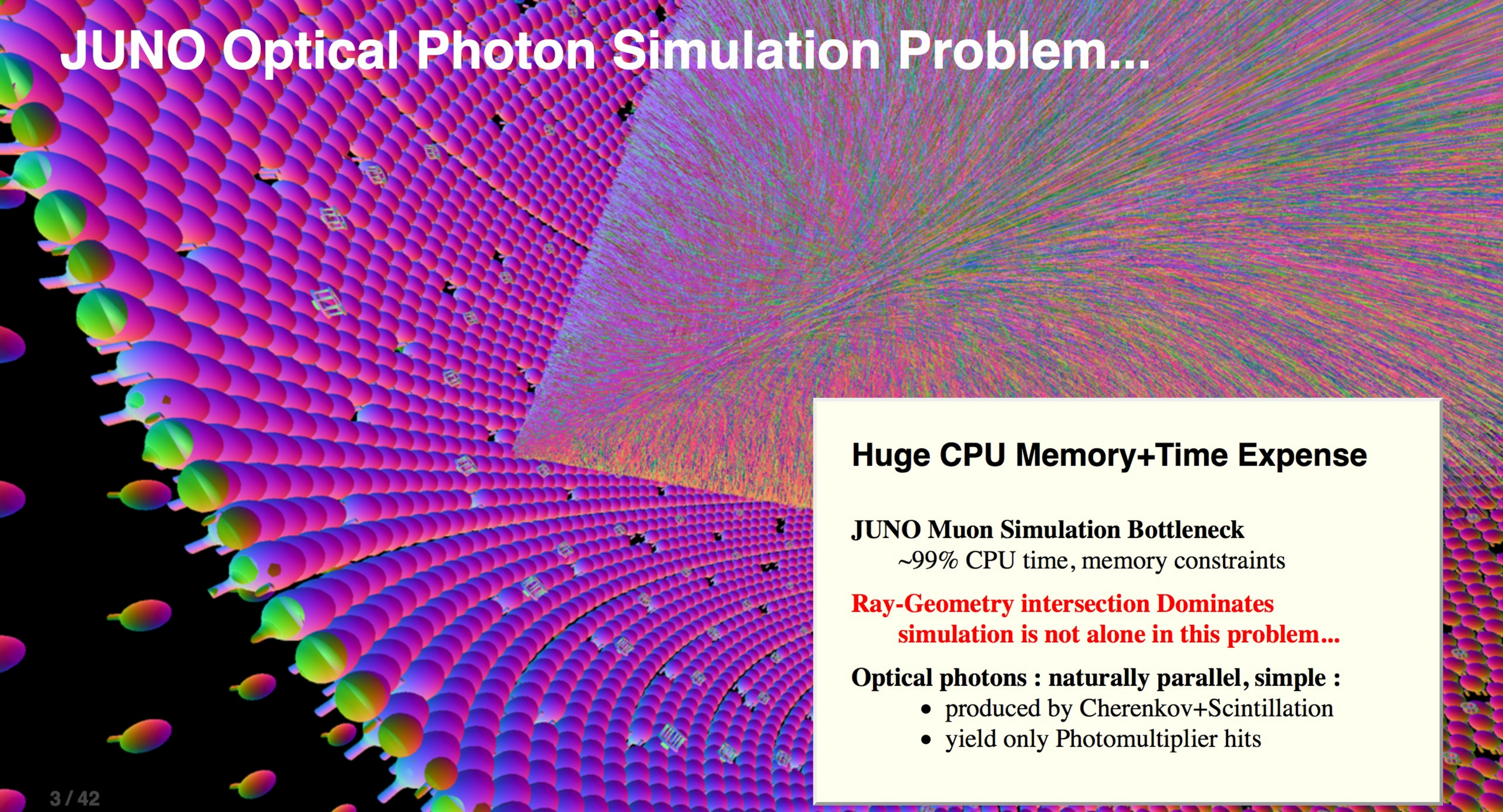
Printed for W. and J. Innys, Printers to the Royal Society, at the *Prince's-Arms* in St. *Paul's* Church-Yard. 1718.

Outline Extras

- Opticks Solids : CSG, Constructive Solid Geometry
  - Constructive Solid Geometry (CSG) : Shapes defined "by construction"
  - CSG : Which primitive intersect to pick ?
  - CSG Complete Binary Tree Serialization -> simplifies GPU side
  - Evaluative CSG intersection Pseudocode : recursion emulated
  - CSG Deep Tree : JUNO "fastener"
  - CSG Deep Tree : height 11 before balancing, too deep for GPU raytrace
  - CSG Deep Tree : Positivize tree using De Morgans laws
  - CSG Deep Tree : height 4 after balancing, OK for GPU raytrace
  - CSG Examples
  - Torus : much more difficult/expensive than other primitives
  - Torus : different artifacts as change implementation/params/viewpoint


OPTICKS:
OR, A
TREATISE
OF THE
Reflections, Refractions, Inflections and *Colours*
OF
LIGHT.
The Second Edition, with Additions.
By Sir Isaac Newton, Knt.
LONDON:
Printed for W. and J. Innys, Printers to the Royal Society, at the *Prince's-Arms* in St. *Paul's* Church-Yard. 1718.

JUNO Optical Photon Simulation Problem...



Huge CPU Memory+Time Expense

JUNO Muon Simulation Bottleneck

~99% CPU time, memory constraints

Ray-Geometry intersection Dominates

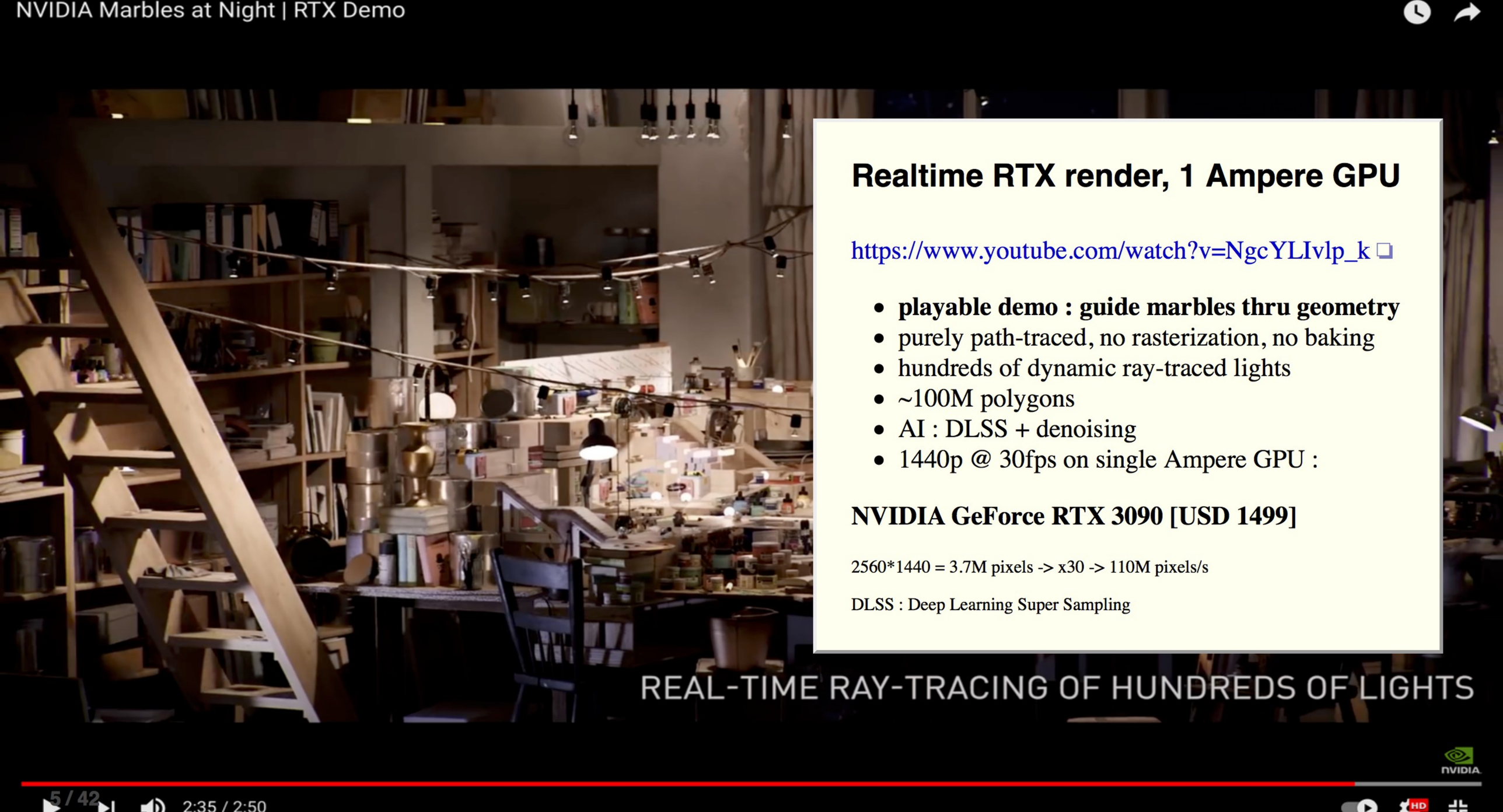
simulation is not alone in this problem...

Optical photons : naturally parallel, simple :

- produced by Cherenkov+Scintillation
- yield only Photomultiplier hits

NVIDIA Marbles at Night | RTX Demo

4 / 42 2:35 / 2:50

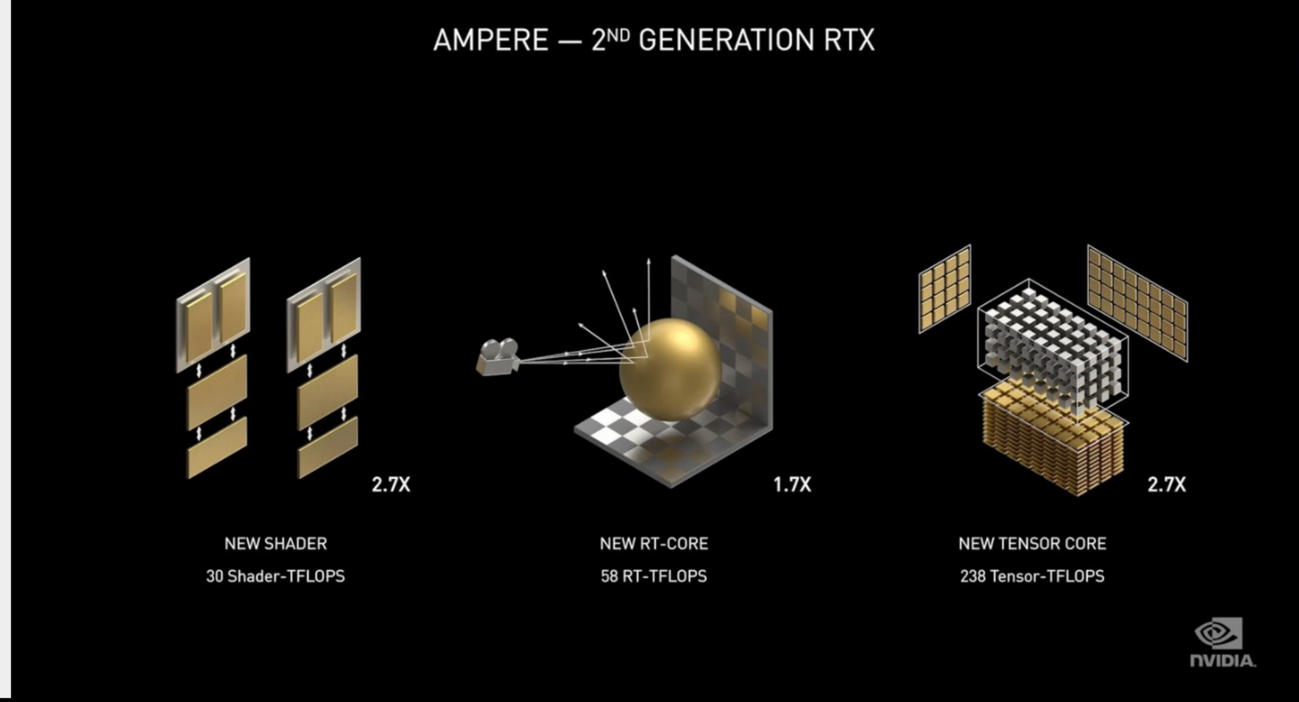

NVIDIA Marbles at Night | RTX Demo
Realtime RTX render, 1 Ampere GPU
https://www.youtube.com/watch?v=NgcYLIvlp_k
playable demo : guide marbles thru geometry
purely path-traced, no rasterization, no baking
hundreds of dynamic ray-traced lights
~100M polygons
AI : DLSS + denoising
1440p @ 30fps on single Ampere GPU :
NVIDIA GeForce RTX 3090 [USD 1499]
2560*1440 = 3.7M pixels -> x30 -> 110M pixels/s
DLSS : Deep Learning Super Sampling
REAL-TIME RAY-TRACING OF HUNDREDS OF LIGHTS
NVIDIA
5 / 42
2:35 / 2:50

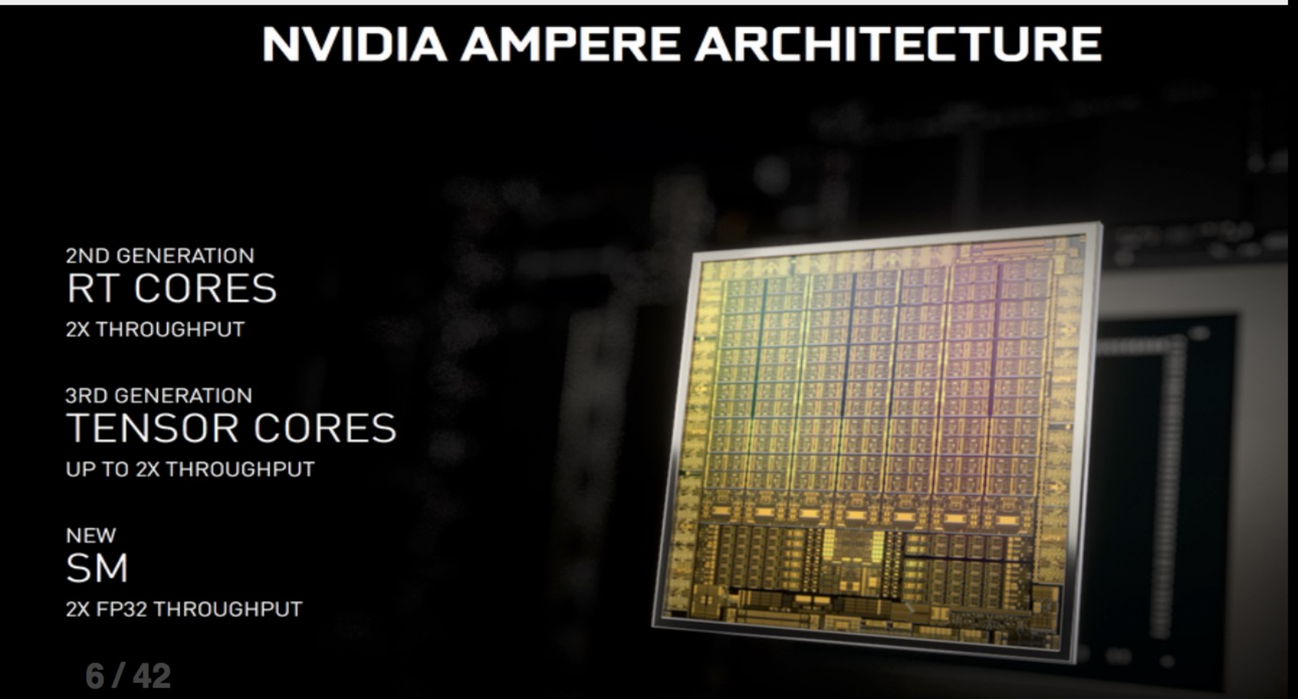
Ampere : 2nd Generation RTX

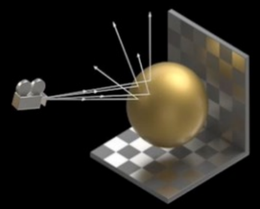
NVIDIA:

"...triple double over Turing..."

- Samsung 8nm (from TSMC 12nm)
- **NVIDIA GeForce RTX 3090**
  - 10,496 CUDA Cores, 28GB VRAM







GPU Ray Tracing (RT) APIs Converging

Three Similar Interfaces over same RTX tech:

NVIDIA OptiX (Linux, Windows) [2009]

- CUDA header only access to Driver functionality

Vulkan RT (Linux, Windows) [final spec 2020]

- cross-vendor cross-platform RT

Microsoft DXR : DirectX 12 Ray Tracing (Windows) [2018]

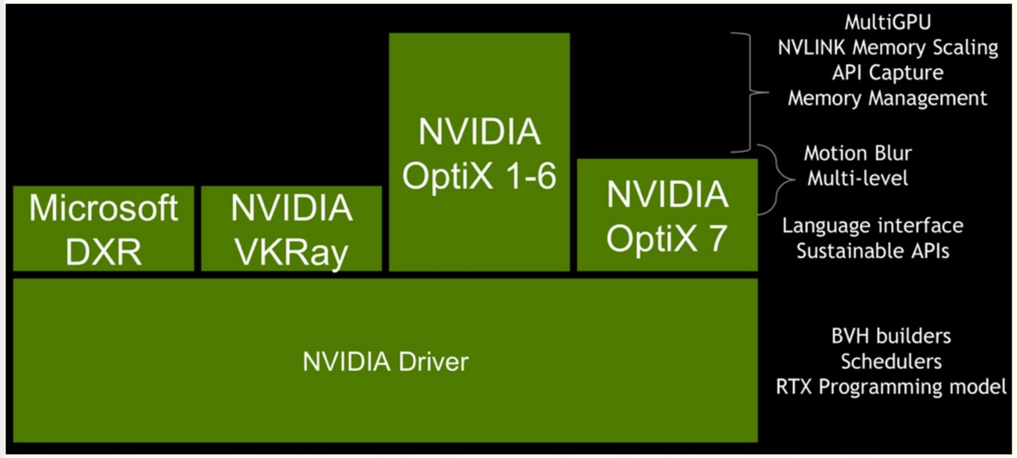
- enhancing visual quality of realtime games

Metal Ray Tracing API (macOS) [introduced 2020[1]]

- Very different Integrated GPU : Apple Silicon M1 GPU
- BUT: similar API

[1] https://developer.apple.com/videos/play/wwdc2020/10012/

Interfaces over NVIDIA Driver



Driver Updates : **Independant of Application**

- new GPU support
- performance improvements

RTX Execution Pipeline : Common to DirectX RT, Vulkan NV RT, OptiX

Acceleration Structure (AS) traversal is central to pipeline performance

RG : Ray Generation

IS : Intersect

CH : Closest Hit

AH : Any Hit

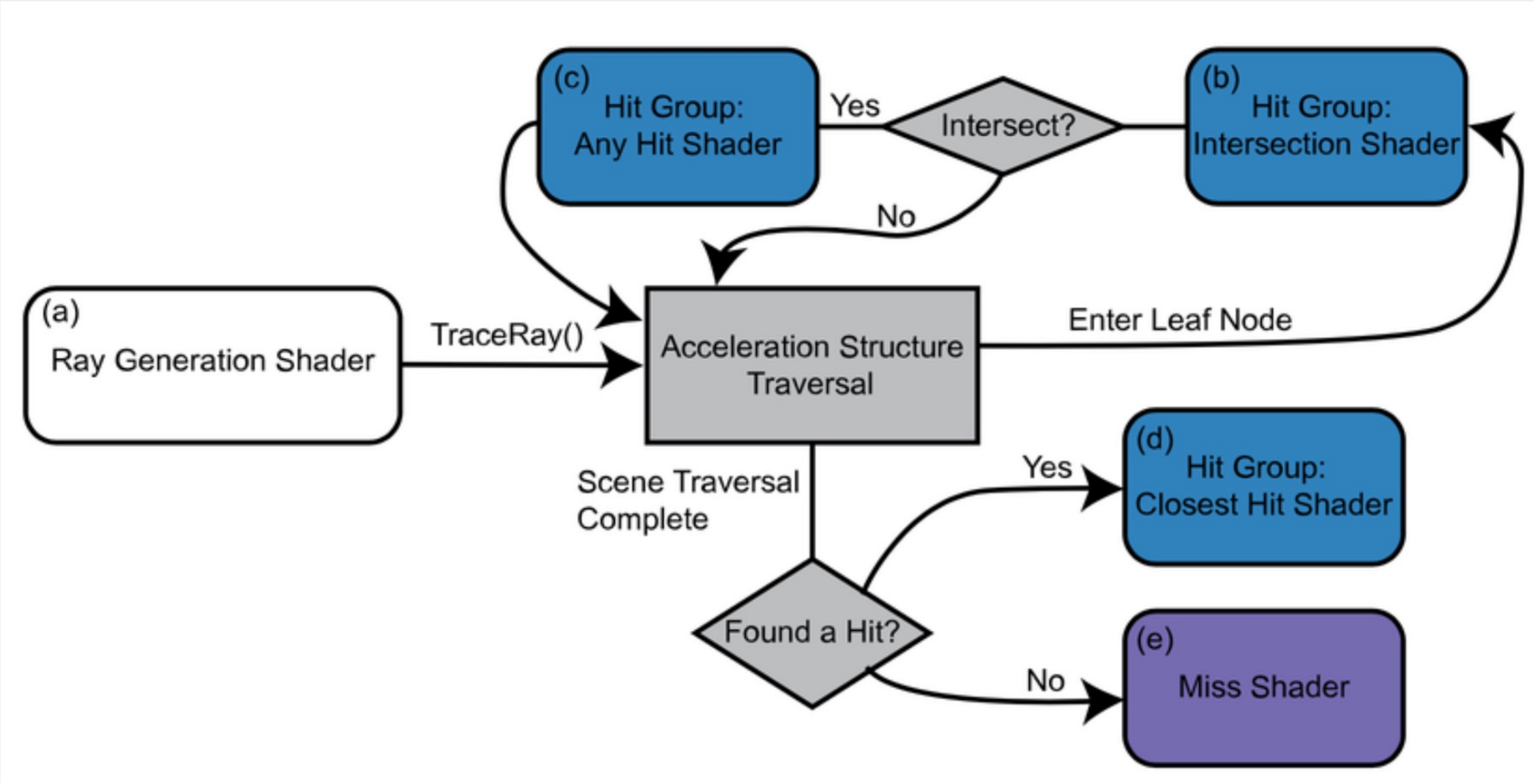
MS : Miss

GPU Opticks

RG
- Cerenkov
- Scintillation
- "bounce" loop

IS
- primitives, CSG

CH
- IS->RG



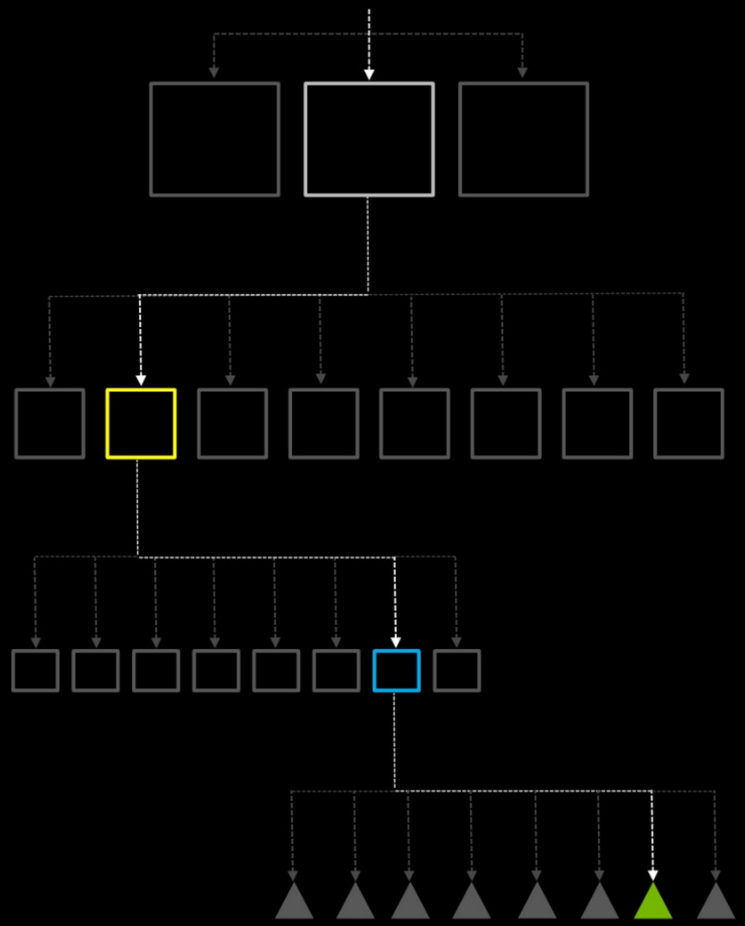
"The RTX Shader Binding Table (SBT) Three Ways", Will Usher

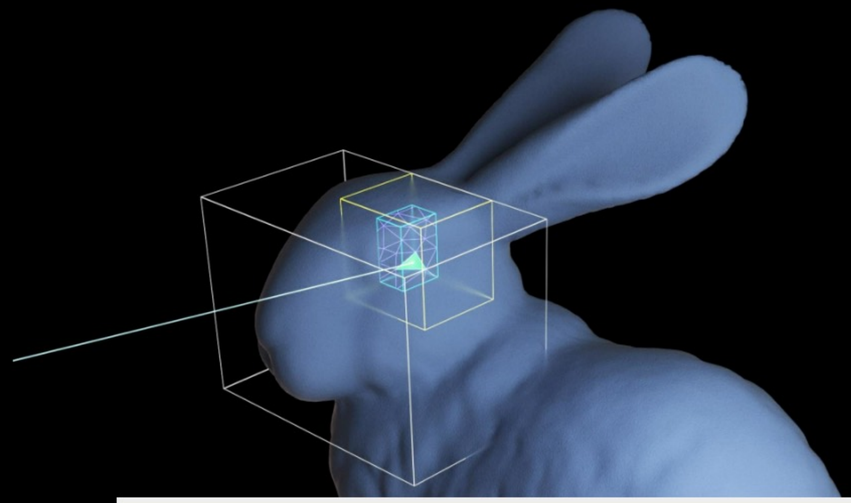
- https://www.willusher.io/graphics/2019/11/20/the-sbt-three-ways ❏

Spatial Index Acceleration Structure

BVH ALGORITHM

Massive Improvement in Search Efficiency





Tree of Bounding Boxes (bbox)

- aims to minimize bbox+primitive intersects
- recursively partitions space

Two-Level Hierarchy : Instance transforms (TLAS) over Geometry (BLAS)

OptiX supports multiple instance levels : IAS->IAS->GAS BUT: **Simple two-level is faster** : works in hardware RT Cores

AS
: Acceleration Structure

TLAS (**IAS**)
: 4x4 transforms, refs to BLAS

BLAS (**GAS**)
: triangles : vertices, indices
: custom primitives : AABB

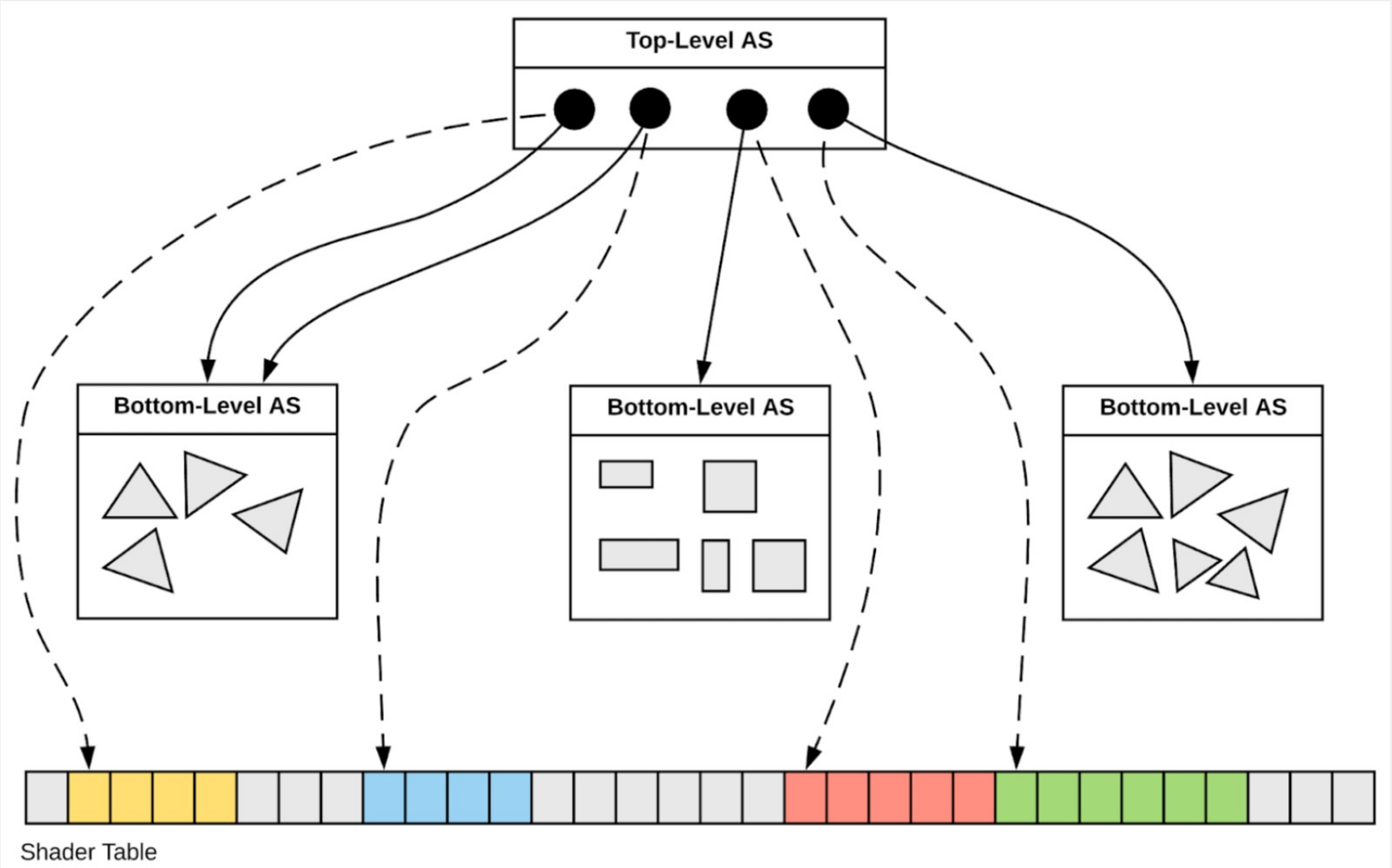
AABB
: axis-aligned bounding box

SBT : Shader Binding Table

Flexibly binds together:

1. geometry objects
2. shader programs
3. data for shader programs

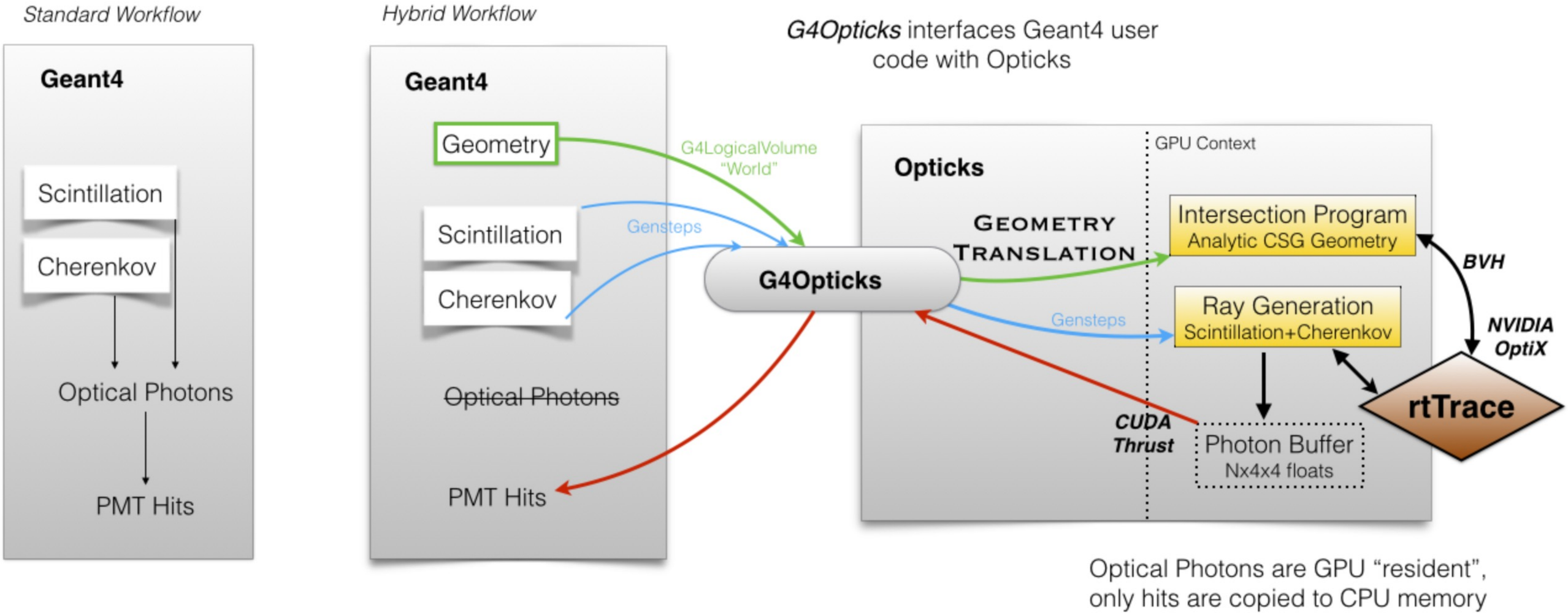
Hidden in OptiX 1-6 APIs



https://developer.nvidia.com/blog/introduction-nvidia-rtx-directx-ray-tracing/ ❑

Geant4 + Opticks Hybrid Workflow : External Optical Photon Simulation

https://bitbucket.org/simoncblyth/opticks



NPY Serialization : Fundamental to Opticks Geometry Model

Separate address space -> cudaMemcpy

upload/download : host(CPU)<->device(GPU)

- Serialize everything -> Arrays
- Thread order undefined -> Arrays
  - (each CUDA thread "owns" slots in the array)

Array-oriented : separate data from compute

- inherent serialization + simplicity
- avoid object serialization/de-serialization
- scales well to millions of element systems

Opticks/NPY pkg : Array Interface Using *glm::mat4 glm::vec4*

- https://bitbucket.org/simoncblyth/opticks/src/master/npy/ ❏

Opticks/GGeo classes implemented with NPY arrays

- all geometry objects persistable -> geocache
- some can be concatenated : *GMesh*, *GParts*
- No dependency on Geant4

[1] http://www.numpy.org/neps/nep-0001-npy-format.html ❏

NPY Serialization Benefits

- simple standard .npy file format[1] : hdr + arr
- flexible load from file and test
- network transport : distributed production
- *NumPy/IPython* debug/analysis eg:

```
a = np.load("transforms.npy")
```

Arrays for Everything -> direct access debug

- (num_photons,4,4) *float32*
- (num_gensteps,6,4) *float32*
- (num_csgnodes,4,4) *float32*
- (num_transforms,3,4,4) *float32*
- (num_planes,4) *float32*
- ...

read/write/stream NumPy arrays from C++

- **NP.hh** : header-only implementation
- https://github.com/simoncblyth/np/ ❏

Persist NumPy Arrays to .npy Files from Python, Examine File Format

IPython persisting NumPy arrays:

```
In [1]: a = np.arange(10)            # array of 10 long (int64)
In [2]: a
Out[2]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [3]: np.save("/tmp/a.npy", a )    # serialize array to file

In [4]: a2 = np.load("/tmp/a.npy")   # load array into memory
In [5]: assert np.all( a == a2 )     # check all elements the same
```

IPython examine NPY file format:

```
In [6]: !xxd /tmp/a.npy              # xxd hexdump file contents
                                     # minimal header : type and shape

00000000: 934e 554d 5059 0100 7600 7b27 6465 7363  .NUMPY..v.{'desc
00000010: 7227 3a20 273c 6938 272c 2027 666f 7274  r': '<i8', 'fort
00000020: 7261 6e5f 6f72 6465 7227 3a20 4661 6c73  ran_order': Fals
00000030: 652c 2027 7368 6170 6527 3a20 2831 302c  e, 'shape': (10,
00000040: 292c 207d 2020 2020 2020 2020 2020 2020  ), }
00000050: 2020 2020 2020 2020 2020 2020 2020 2020
00000060: 2020 2020 2020 2020 2020 2020 2020 2020
00000070: 2020 2020 2020 2020 2020 2020 2020 200a                .
00000080: 0000 0000 0000 0000 0100 0000 0000 0000  ................
00000090: 0200 0000 0000 0000 0300 0000 0000 0000  ................
..
```

Load NumPy array into C/C++

Straightforward to parse NPY files

http://github.com/simoncblyth/np/ ❏

NP::Load

- parses file header, array shape + type
- reads array data into std::vector

```
// gcc NPMinimal.cc -lstdc++ && ./a.out /tmp/a.npy
#include "NP.hh"
int main(int argc, char** argv)
{
    assert( argc > 1 && argv[1] ) ;
    NP* a = NP::Load(argv[1]) ;
    a->dump();
    return 0 ;
}
```

Translation 1st Step : Geant4 -> Opticks/GGeo : 1->1 conversions

Structural volumes : G4PVPlacement ->

GVolume
JUNO: tree of ~300,000 *GVolume*

Solid shapes : G4VSolid ->

GMesh (collected into **GMeshLib**)
arrays: vertices, indices
ref to *NCSG*

NCSG
tree of *NNode* (CSG constituents)

Material/surface properties as function of wavelength

- *G4Material -> GMaterial*
- *G4Logical(Border/Skin)Surface -> GSurface*
- adopts standard wavelength domain
- collected into **GMaterialLib GSurfaceLib**

Translation steered by X4 package

https://bitbucket.org/simoncblyth/opticks/src/master/extg4/X4PhysicalVolume.hh ❑

Geant4 -> Opticks/GGeo -> OptiX

Multi-stage translation

Geant4
G4PVPlacement, G4VSolid, G4Material, ...

Opticks/GGeo : *(No Geant4,OptiX dependency)*
GGeo, GVolume, GParts, GMesh, GMaterial, ...

- persists as binary **.npy** file based geocache

Opticks/OptiXRap
OGeo, OGeometry, OBndLib, ...

- instanciation populates GPU OptiX context

OptiX
"IAS", "GAS", ...

Translation 2nd Step : Opticks/GGeo Instancing : "Factorizes" Geometry

Structural volumes vs solid shapes
distinction for convenience only, **distinction is movable**

JUNO: ~300,000 GVolume : **mostly small repeated groups** (PMTs)

GGeo/GInstancer

0. GVolume **progeny digest** : shapes+transforms -> subtree ident.
1. find repeated **digests**, disqualifying repeats inside others
2. label all nodes with repeat index, non-repeated remainder : 0

For each repeat+remainder create **GMergedMesh**:

- collecting transforms, identity -> instance arrays
- merged volumes+solids
  - **GMesh**: concatenated arrays: triangles, indices
  - **GParts**: concatenated arrays: CSG nodes + transforms
  - transforms applied -> gets into instance frame
  - **Consolidation : structural volumes -> compound solid**

GMergedMesh -> IAS+GAS

- OptiX6 : ~10(IAS + GAS) OptiX7 Aim: 1 IAS + ~10 GAS

https://bitbucket.org/simoncblyth/opticks/src/master/ggeo/GInstancer.hh

Form of GPU Detector Geometry

GAS

- intersection + bbox CUDA programs
- buffers: CSG nodes, transforms, planes

IAS

- transforms and references to **GAS**
- identity info, boundary tex. refs

Boundary GPU Texture, interleaving:

- material props
- surface props

JUNO: ~300,000 GVolume -> ~10 GMergedMesh

repeated GMergedMesh
thousands of instance transforms
consolidates < 10 *GVolume*

remainder GMergedMesh
one identity instance transform
consolidates ~ few hundred *GVolume*

Optimizing Geometry : Split BLAS to avoid overlapping bbox

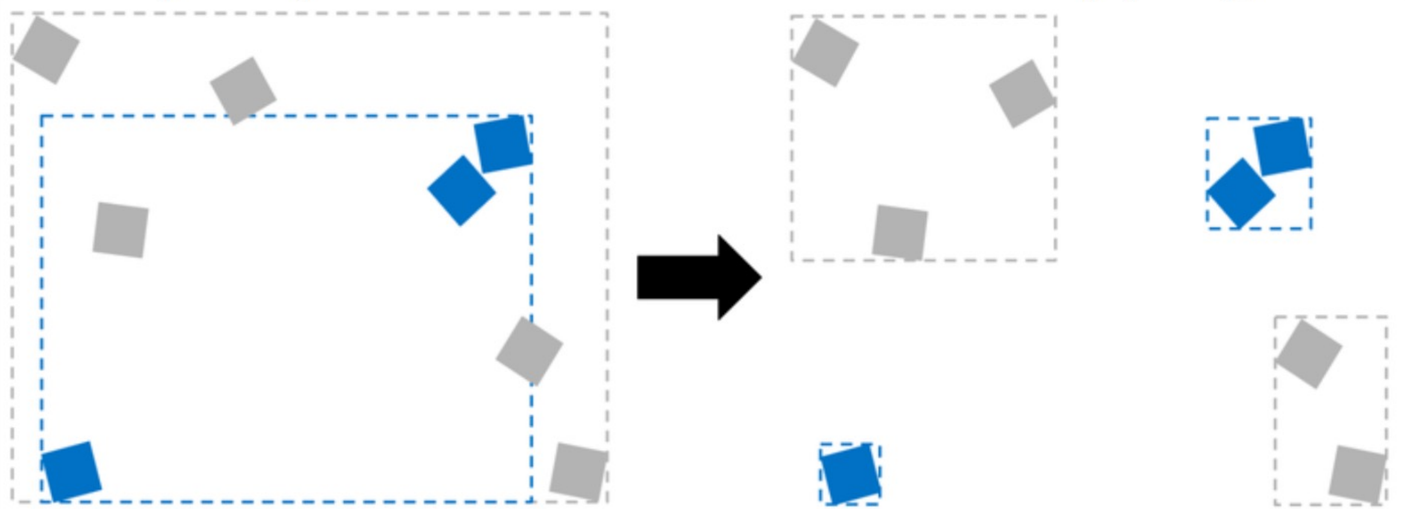


Figure 1. Geometries in two BLASes with overlapping AABBs with a lot of empty space. After splitting the BLASes to four independent ones, the AABBs don't overlap anymore.

Optimization : **deciding where to draw lines between:**

1. structure and solid (**IAS** and **GAS**)
2. solids within **GAS** (bbox choice to minimize traversal intersection tests)

Where those lines are drawn **defines the AS**

https://developer.nvidia.com/blog/best-practices-using-nvidia-rtx-ray-tracing/ ❑

Optimizing Geometry : Merge BLAS when lots of overlaps

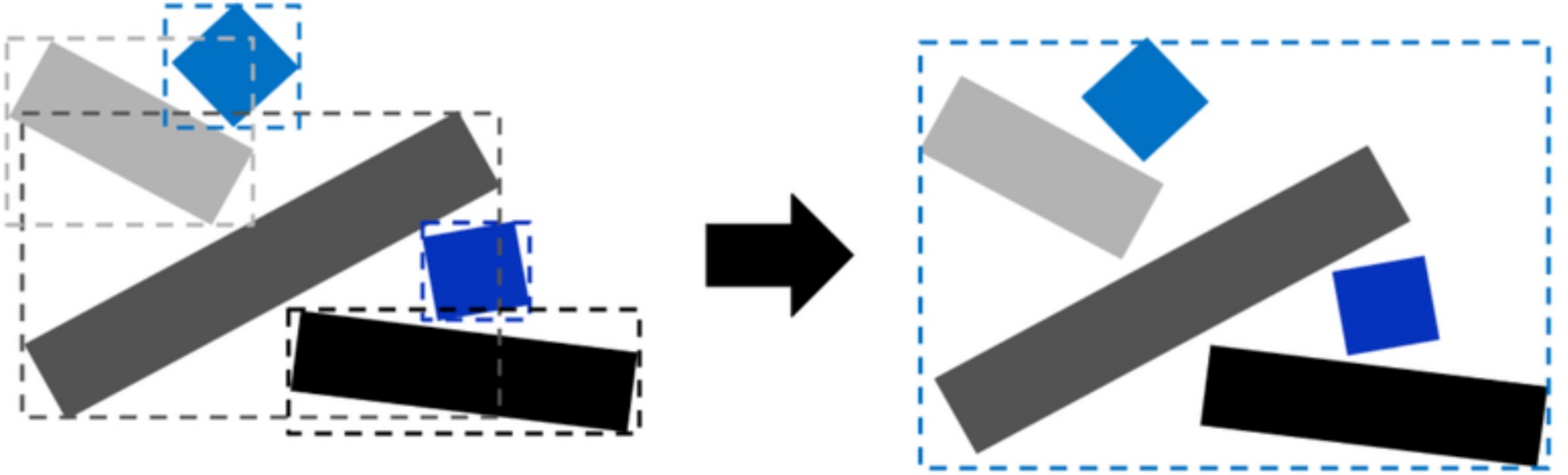


Figure 2. Independent instances with overlapping AABBs. Merging them into one BLAS would be efficient.

- lots of overlapping forces lots of intersections to find closest
- but too few bbox means the **AS** cannot help to avoid intersect tests
- balance required : **needs experimentation and measurement to optimize**

https://developer.nvidia.com/blog/best-practices-using-nvidia-rtx-ray-tracing/ ❑

Ray Intersection with Transformed Object -> Geometry Instancing

Equivalent Intersects -> same *t*

1. **ray** with *ellipsoid* : **M*p**
2. M^{-1} **ray** with *sphere* : **p**

Local Frame Advantages

1. simpler intersect (sphere vs ellipsoid)
2. closer to origin -> better precision

Geometry Instancing Advantages

- many objects share local geometry
  - orient+position with 4x4 **M**
- huge VRAM saving, less to copy

Requirements

- must **not** normalize ray direction
- normals transform differently
  - $N' = N * M^{-1T}$
  - (due to non-uniform scaling)

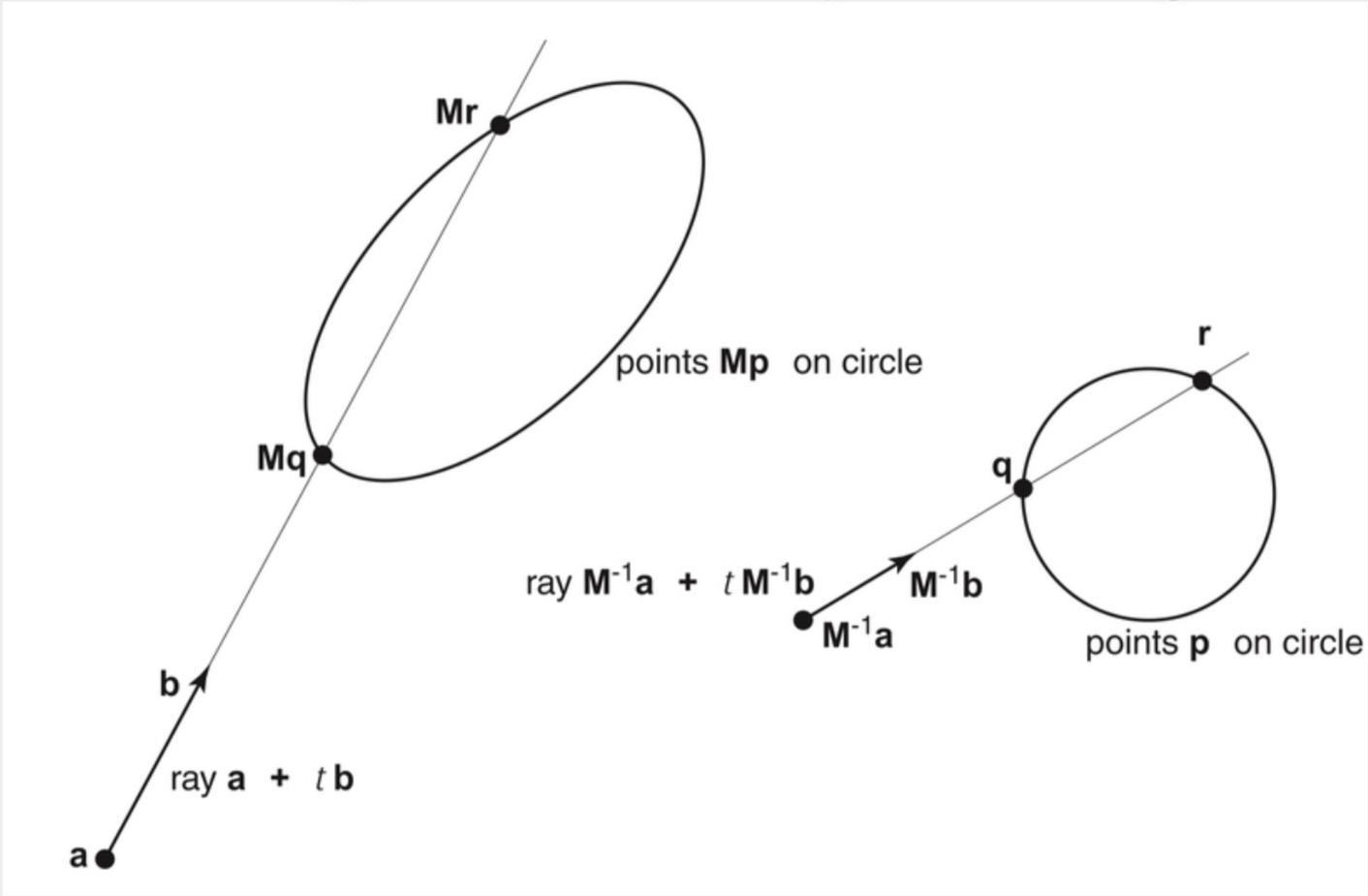
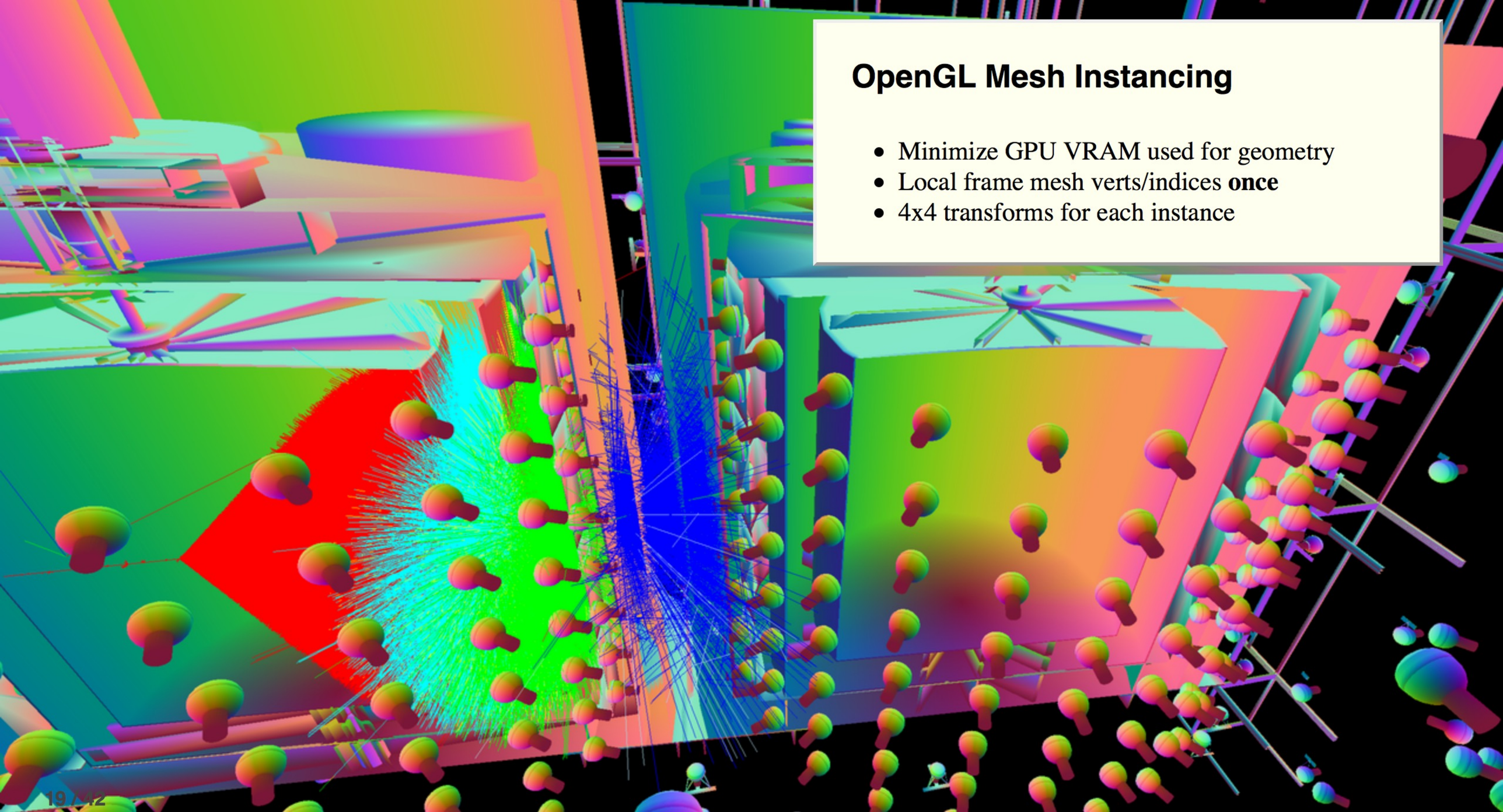


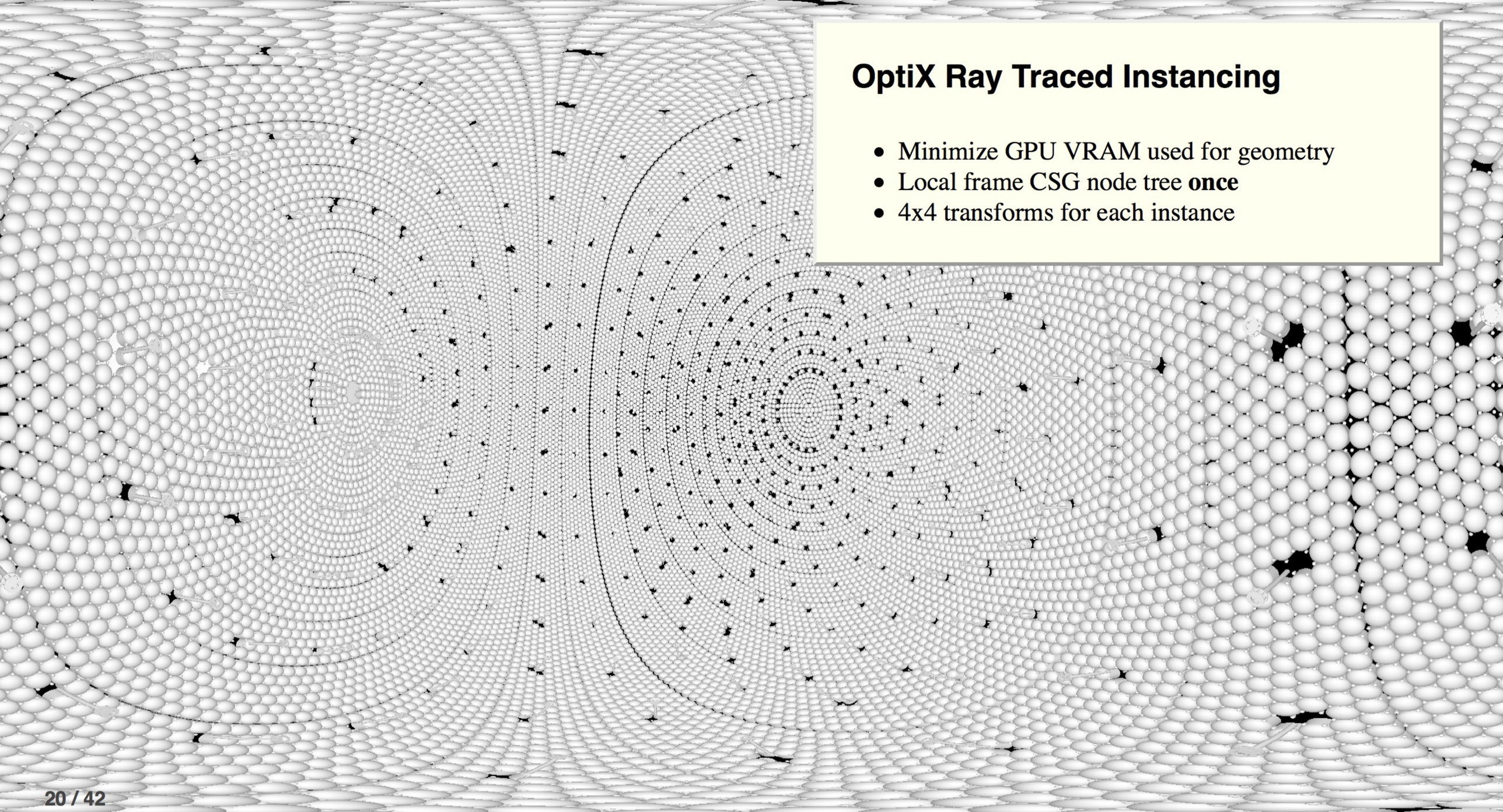
Fig 13.5 "Realistic Ray Tracing", Peter Shirley

Advantages apply equally to acceleration structures



OpenGL Mesh Instancing

- Minimize GPU VRAM used for geometry
- Local frame mesh verts/indices **once**
- 4x4 transforms for each instance

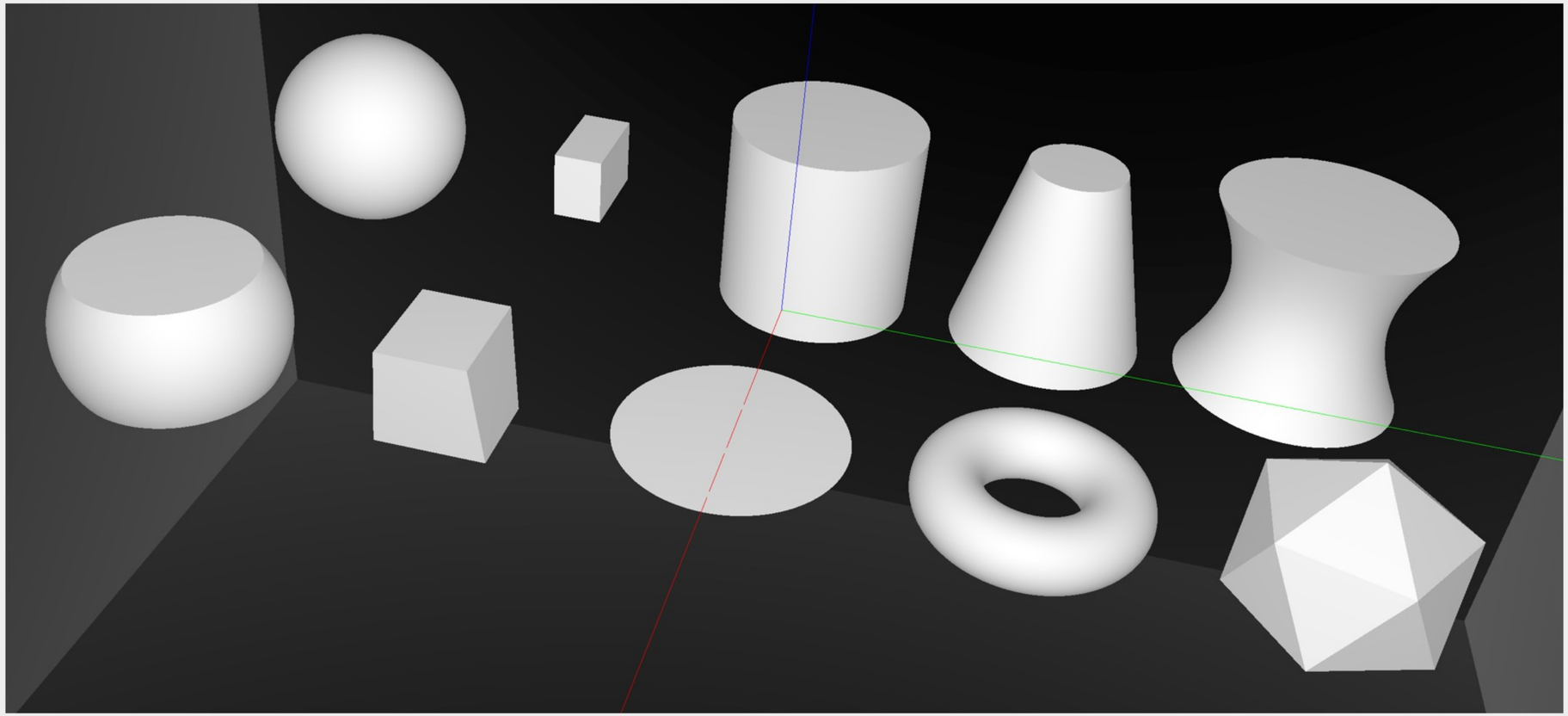


OptiX Ray Traced Instancing

- Minimize GPU VRAM used for geometry
- Local frame CSG node tree **once**
- 4x4 transforms for each instance

G4VSolid -> CUDA Intersect Functions for ~10 Primitives

- 3D parametric ray : **ray(x,y,z;t) = rayOrigin + t * rayDirection**
- implicit equation of primitive : **f(x,y,z) = 0**
- -> polynomial in **t** , roots: **t > t_min** -> intersection positions + surface normals



Sphere, Cylinder, Disc, Cone, Convex Polyhedron, Hyperboloid, Torus, ...

intersect_analytic.cu : for all shapes/concatenated-shapes/primitives

OptiXRap/cu/intersect_analytic.cu

- concatenated shapes, called for each **primIdx**

```
425 RT_PROGRAM void intersect(int primIdx)
426 {
427     const Prim& prim    = primBuffer[primIdx];
428
429     unsigned partOffset  = prim.partOffset() ;
430     unsigned numParts    = prim.numParts() ;
431     unsigned primFlag    = prim.primFlag() ;
432
433     if(primFlag == CSG_FLAGNODETREE)
434     {
435         evaluative_csg( prim, primIdx );
436     }
474 }
```

```
229 RT_PROGRAM void bounds (int primIdx, float result[6])
230 {
251     optix::Aabb* aabb = (optix::Aabb*)result;
252     *aabb = optix::Aabb();
253
254     uint4 identity =
        identityBuffer[instance_index*primitive_count+primIdx] ;
...
271     const Prim& prim    = primBuffer[primIdx];
...
294     if(primFlag == CSG_FLAGNODETREE || primFlag == CSG_FLAGINVISIBLE )
295     {
301         csg_bounds_prim(primIdx, prim, aabb);
318     }
385 }
```

Shapes from buffer content:

primBuffer
: offsets into *part*, *tran*, *plan* buffers

partBuffer
: nodes of the CSG tree (operators+primitives)

tranBuffer
: transforms of CSG nodes (instance local frame)

planBuffer
: planes

identityBuffer
: volume identity *uint4* (including **boundary**)

Populated from *GMergedMesh*

https://bitbucket.org/simoncblyth/opticks/src/master/optixrap/cu/intersect_analytic.cu

G4Boolean -> CUDA/OptiX Intersection Program Implementing CSG

Complete Binary Tree, pick between pairs of nearest intersects:

UNION tA < tB	Enter B	Exit B	Miss B
Enter A	ReturnA	LoopA	ReturnA
Exit A	ReturnA	ReturnB	ReturnA
Miss A	ReturnB	ReturnB	ReturnMiss

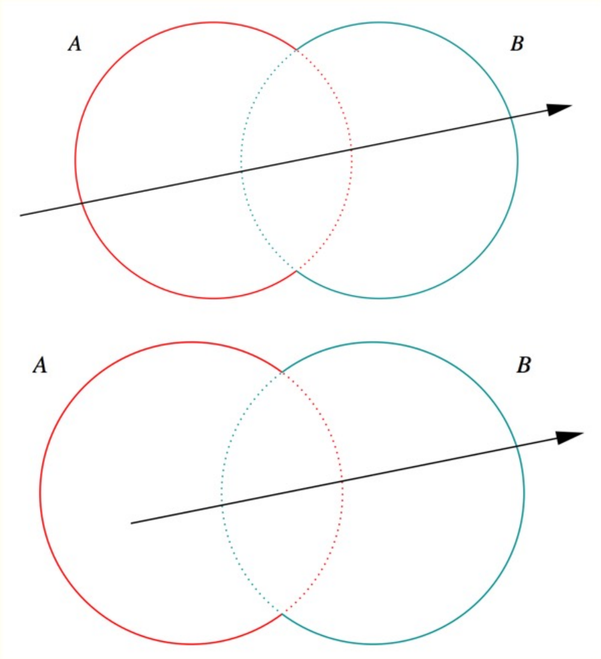
- *Nearest hit intersect algorithm* [1] avoids state
  - sometimes Loop : advance **t_min** , re-intersect both
  - classification shows if inside/outside
- *Evaluative* [2] implementation emulates recursion:
  - recursion not allowed in OptiX intersect programs
  - bit twiddle traversal of complete binary tree
  - stacks of postorder slices and intersects
- Identical geometry to Geant4
  - solving the same polynomials
  - near perfect intersection match

[1] Ray Tracing CSG Objects Using Single Hit Intersections, Andrew Kensler (2006)
with corrections by author of XRT Raytracer http://xrt.wikidot.com/doc:csg

[2] https://bitbucket.org/simoncblyth/opticks/src/tip/optixrap/cu/csg_intersect_boolean.h
Similar to binary expression tree evaluation using postorder traverse.

Outside/Inside Unions

dot(normal,rayDir) -> Enter/Exit



- **A + B** boundary not inside other
- **A * B** boundary inside other

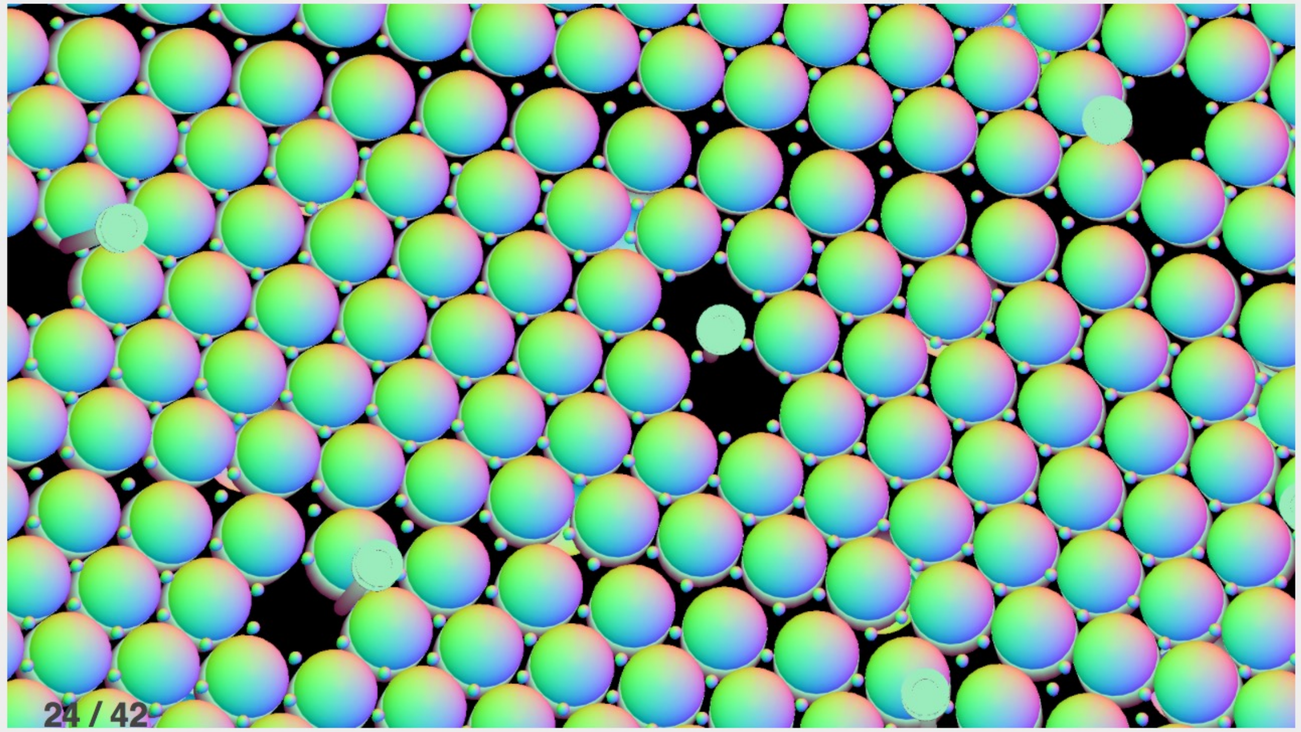
Opticks : Translates G4 Geometry to GPU, Without Approximation

G4 Structure Tree -> Instance+Global Arrays -> OptiX

Group structure into repeated instances + global remainder:

- auto-identify repeated geometry with "progeny digests"
  - JUNO : 5 distinct instances + 1 global
- instance transforms used in OptiX/OpenGL geometry

instancing -> huge memory savings for JUNO PMTs



Materials/Surfaces -> GPU Texture

Material/Surface/Scintillator properties

- interpolated to standard wavelength domain
- interleaved into "boundary" texture
- "reemission" texture for wavelength generation

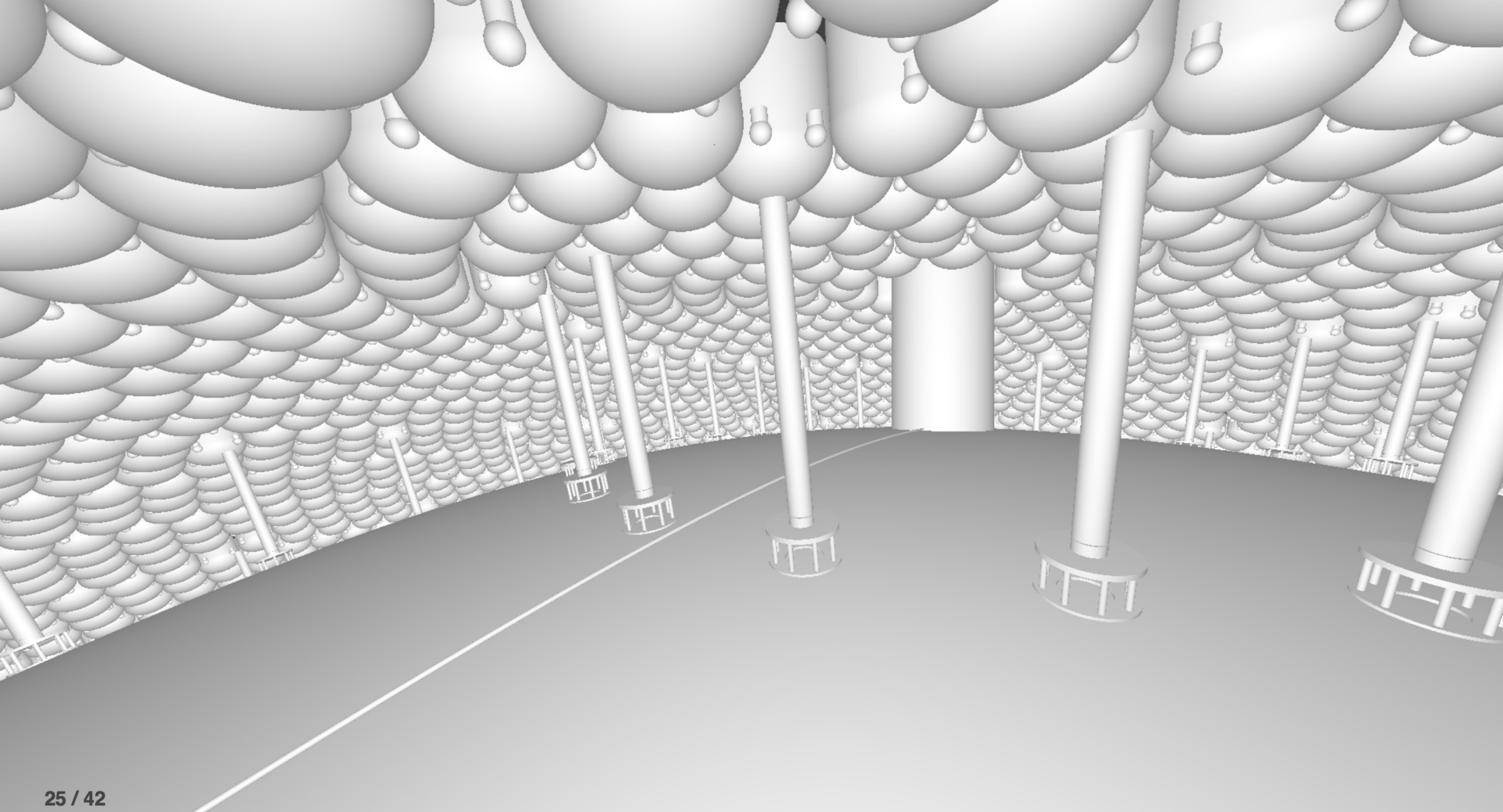
Material/surface boundary : 4 indices

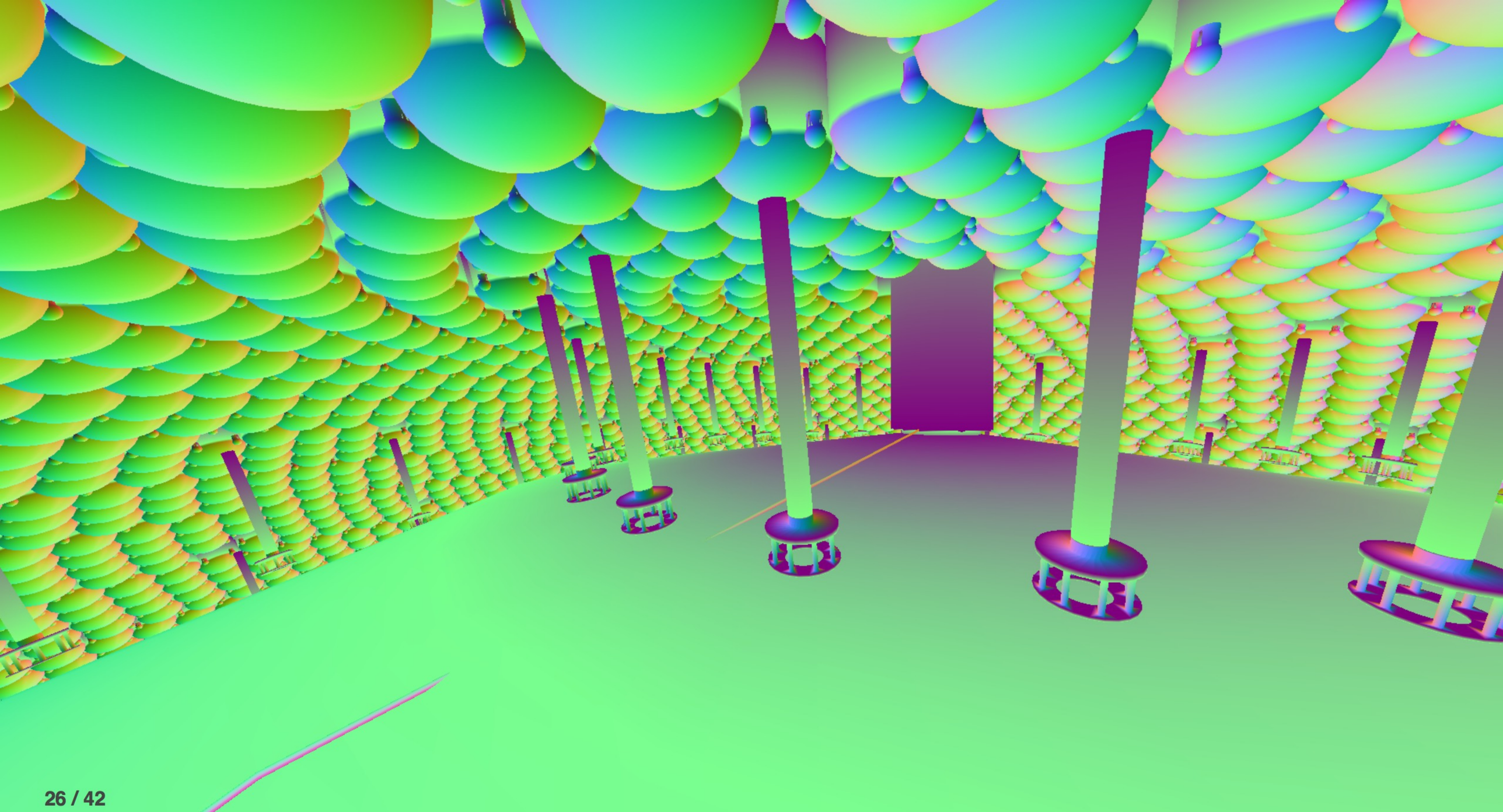
- outer material (parent)
- outer surface (inward photons, parent -> self)
- inner surface (outward photons, self -> parent)
- inner material (self)

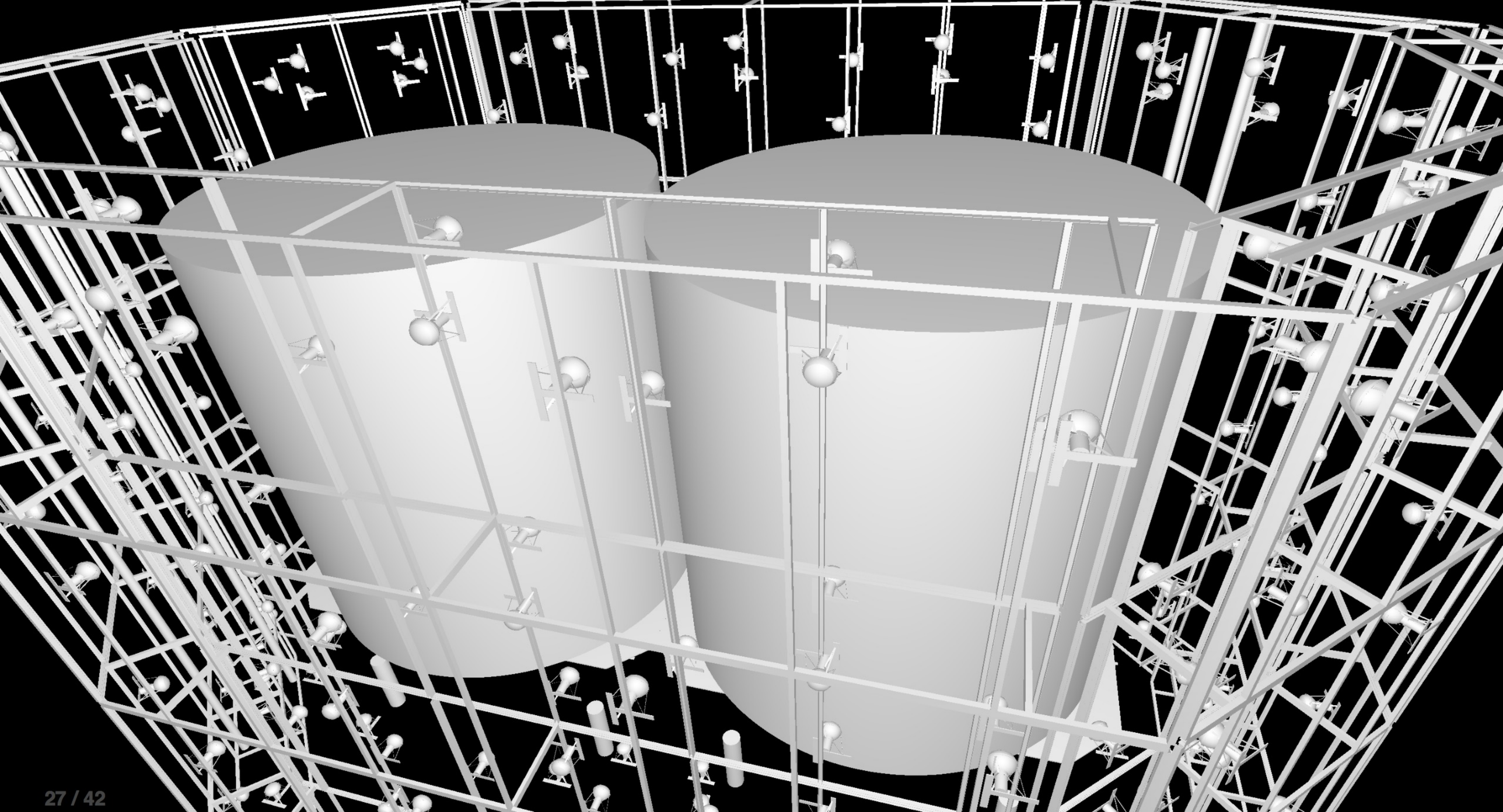
Primitives labelled with unique boundary index

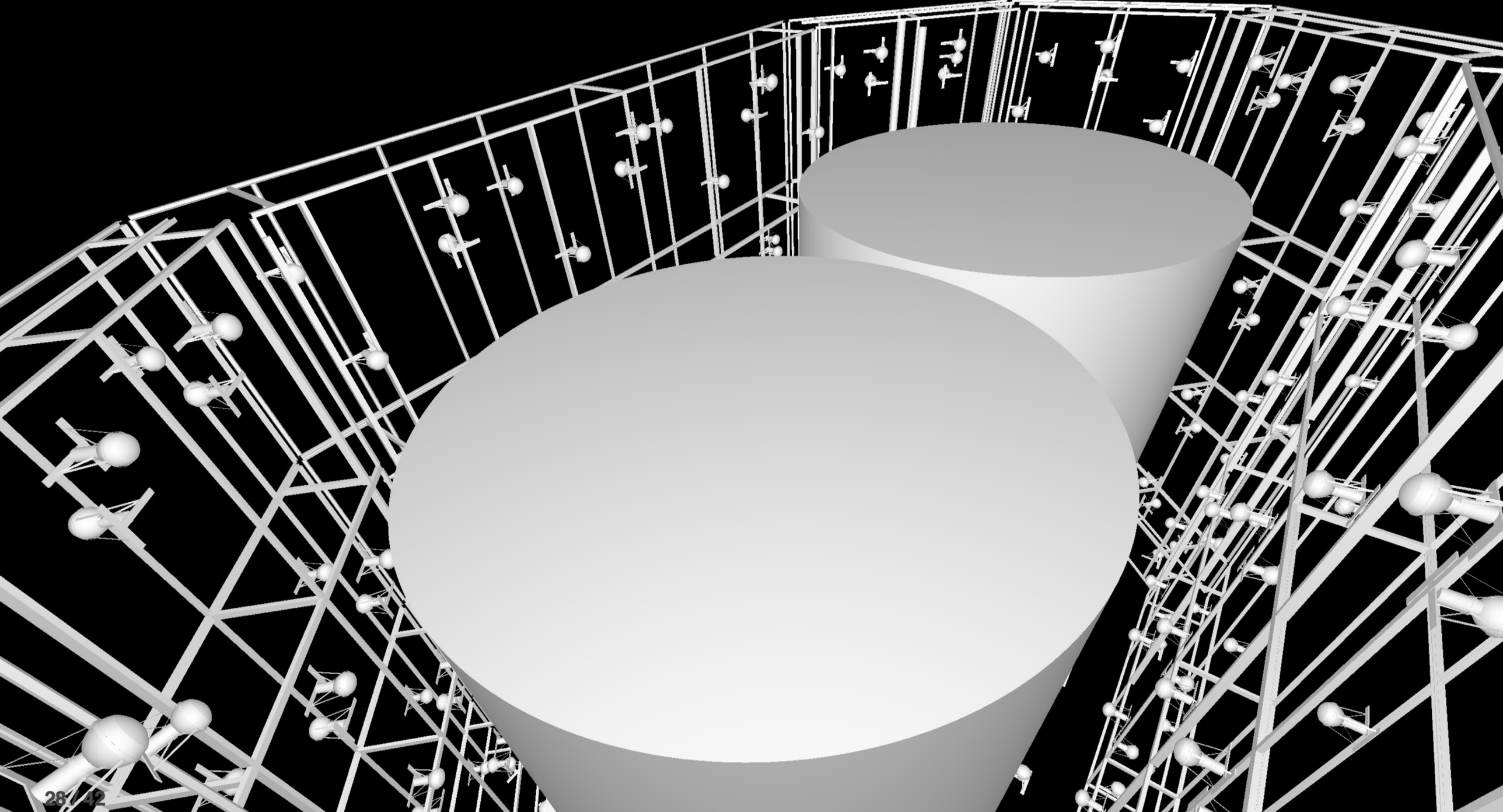
- ray primitive intersection -> boundary index
- texture lookup -> material/surface properties

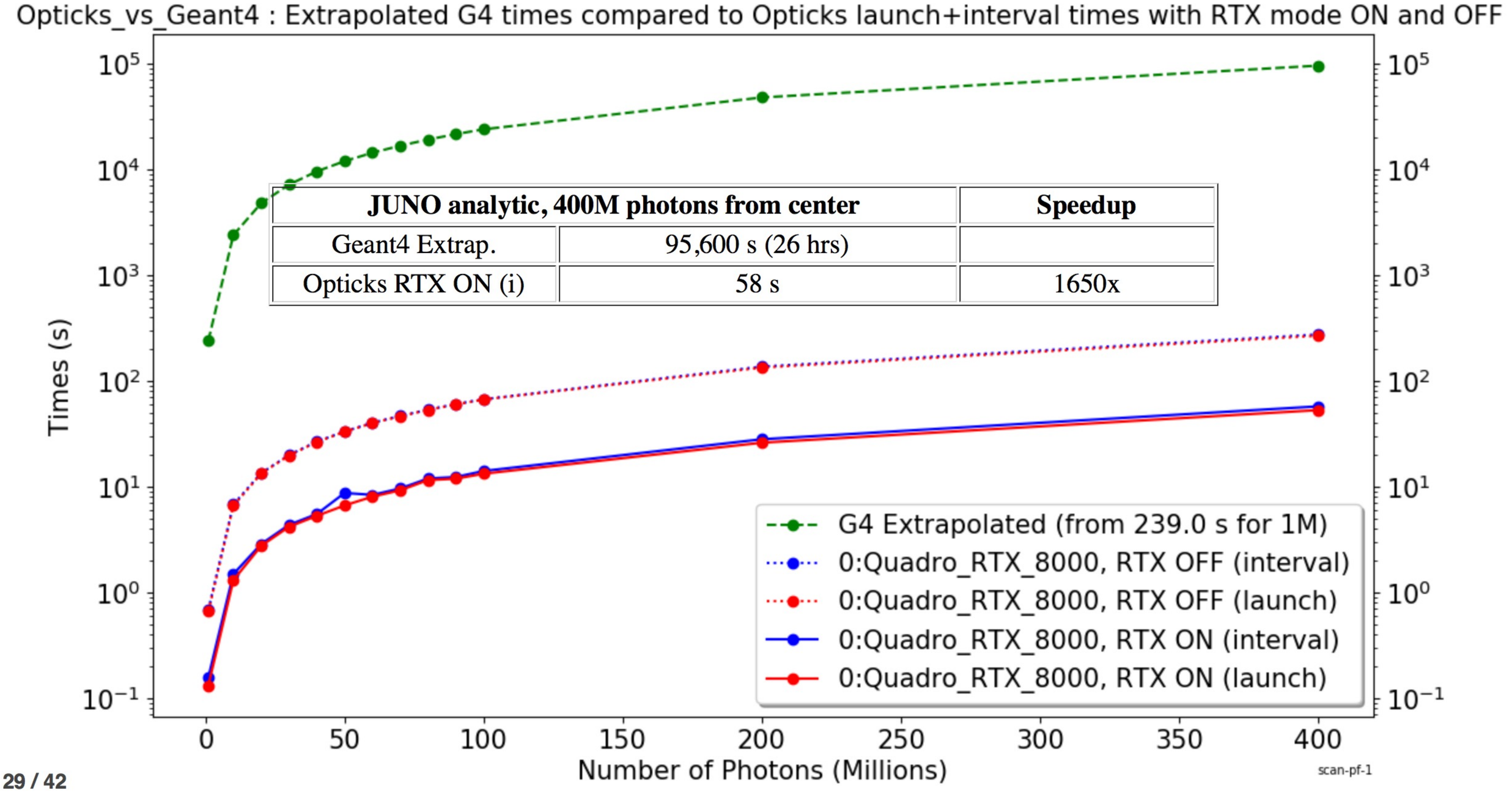
simple/fast properties + reemission wavelength





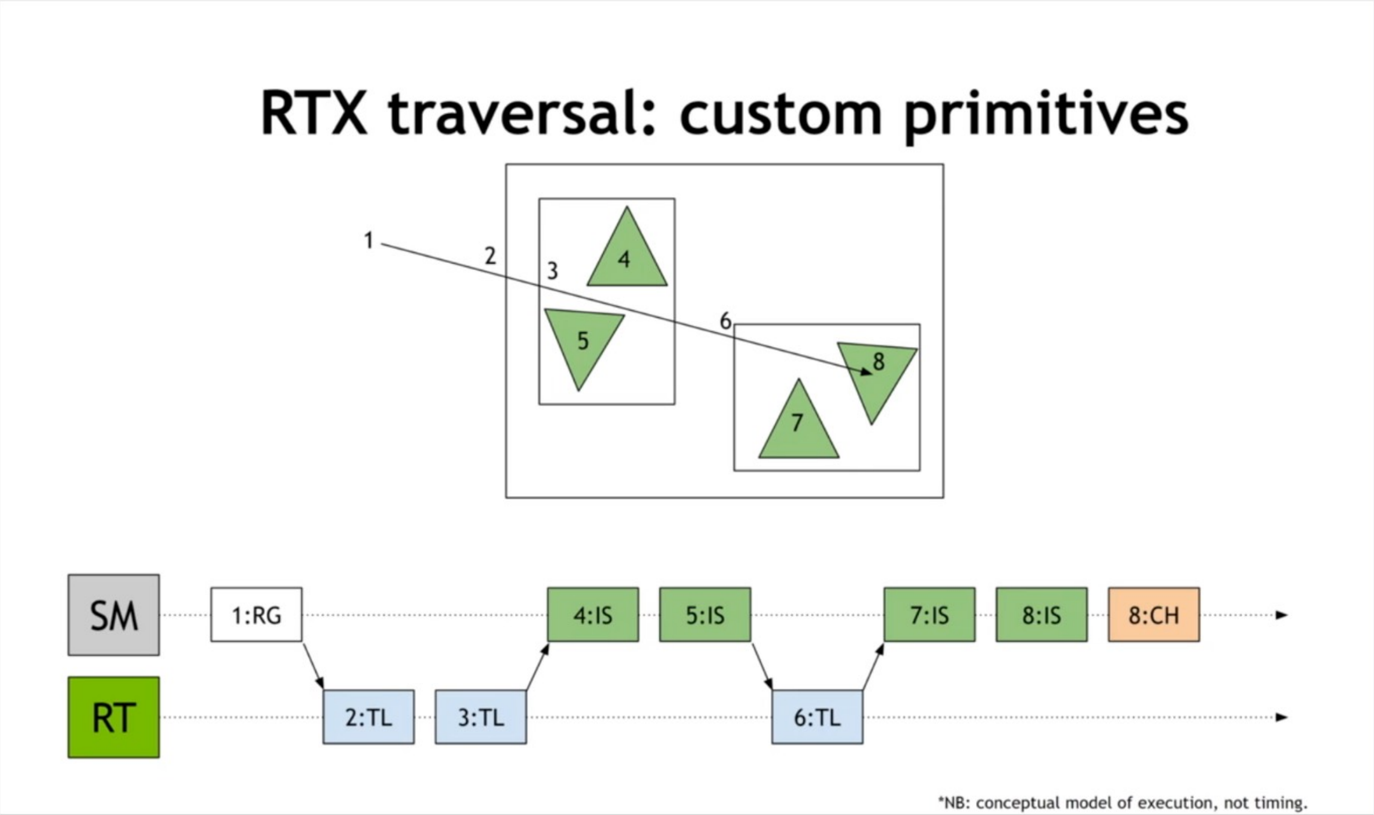





Opticks_vs_Geant4 : Extrapolated G4 times compared to Opticks launch+interval times with RTX mode ON and OFF
JUNO analytic, 400M photons from center
Speedup
Geant4 Extrap.
95,600 s (26 hrs)
Opticks RTX ON (i)
58 s
1650x
Times (s)
Number of Photons (Millions)
G4 Extrapolated (from 239.0 s for 1M)
0:Quadro_RTX_8000, RTX OFF (interval)
0:Quadro_RTX_8000, RTX OFF (launch)
0:Quadro_RTX_8000, RTX ON (interval)
0:Quadro_RTX_8000, RTX ON (launch)
scan-pf-1

Useful Speedup > 1500x : But Why Not Giga Rays/s ? (1 Photon ~10 Rays)

- NVIDIA claim : 10 Giga Rays/s with RT Core
- **-> 1 Billion photons per second**
- **RT cores : built-in triangle intersect + 1-level of instancing**
- flatten scene model to avoid SM<->RT roundtrips ?



OptiX Performance Tools and Tricks, David Hart, NVIDIA
https://developer.nvidia.com/siggraph/2019/video/sig915-vid

100M photon RTX times, avg of 10

Launch times for various geometries			
Geometry	**Launch (s)**	**Giga Rays/s**	**Relative to ana**
JUNO ana	13.2	0.07	
JUNO tri.sw	6.9	0.14	1.9x
JUNO tri.hw	2.2	0.45	6.0x
Boxtest ana	0.59	1.7	
Boxtest tri.sw	0.62	1.6	
Boxtest tri.hw	0.30	3.3	1.9x

- ana : Opticks analytic CSG (SM)
- tri.sw : software triangle intersect (SM)
- tri.hw : hardware triangle intersect (RT)

JUNO 15k triangles, 132M without instancing

Simple Boxtest geometry gets into ballpark

Summary : Opticks Detector Geometry

Opticks : state-of-the-art GPU ray tracing applied to optical photon simulation and integrated with *Geant4*, giving a leap in performance that eliminates memory and time bottlenecks.

GPU : Geometry Rethink Mandatory

- ray tracing APIs -> simple 2-level hierarchy
- rethink amply rewarded

Opticks > 1500x Geant4 (one Turing GPU)

- Drastic speedup -> better detector understanding -> greater precision
  - **any simulation limited by optical photons can benefit**
  - more photon limited -> more overall speedup (99% -> 100x)

https://bitbucket.org/simoncblyth/opticks ❏	code repository
https://simoncblyth.bitbucket.io ❏	presentations and videos
https://groups.io/g/opticks ❏	forum/mailing list archive
email:opticks+subscribe@groups.io	subscribe to mailing list

Constructive Solid Geometry (CSG) : Shapes defined "by construction"

Simple *by construction* definition, implicit geometry.

- **A, B** implicit primitive **solids**
- **A + B** : union (OR)
- **A * B** : intersection (AND)
- **A - B** : difference (AND NOT)
- **!B** : complement (NOT) (inside <-> outside)

CSG expressions

- non-unique: **A - B == A * !B**
- represented by binary tree, primitives at leaves

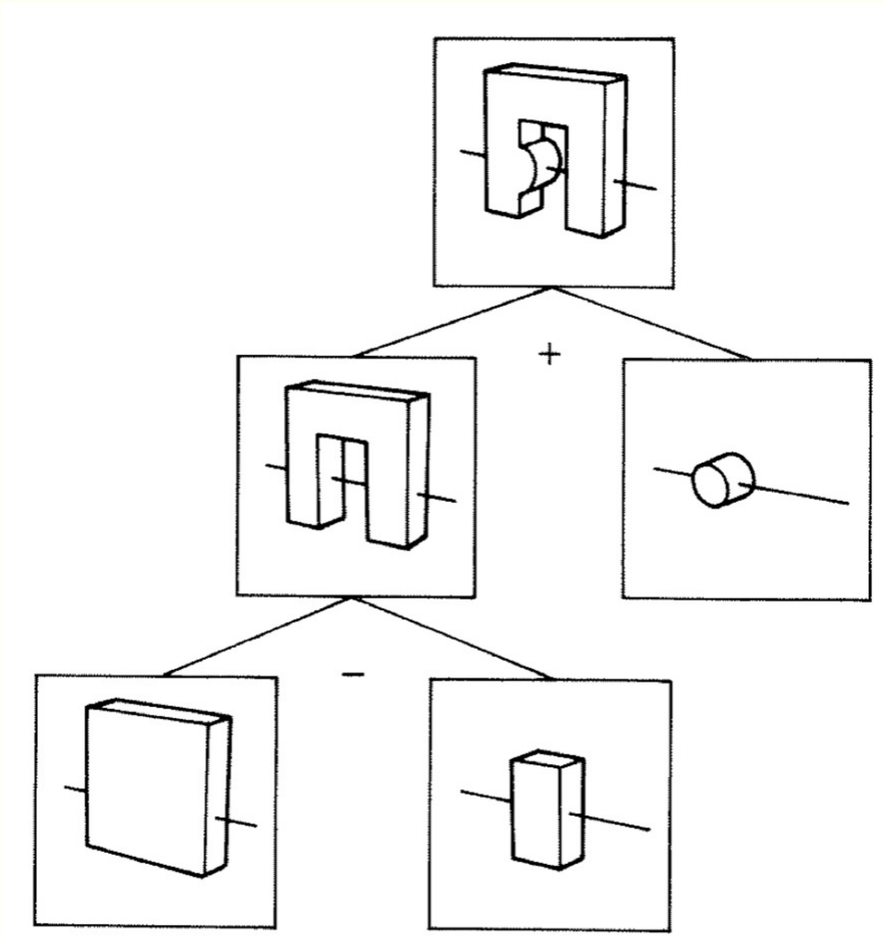
3D Parametric Ray : **ray(t) = r0 + t rDir**

Ray Geometry Intersection

- primitive : find *t* roots of implicit eqn
- composite : pick primitive intersect, depending on CSG tree

How to pick exactly ?

CSG Binary Tree



Primitives combined via binary operators

CSG : Which primitive intersect to pick ?

Classical Roth diagram approach

- find all ray/primitive intersects
- recursively combine inside intervals using CSG operator
- works from leaves upwards

Computational requirements:

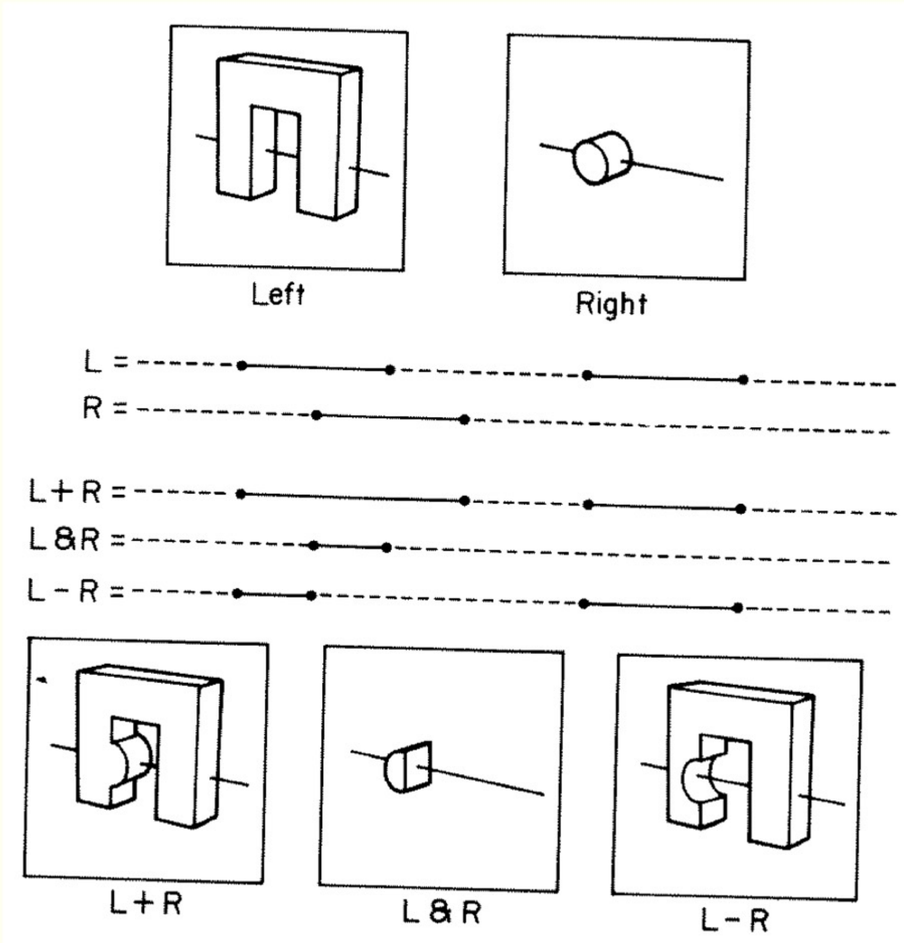
- find all intersects, store them, order them
- recursive traverse

BUT : **High performance on GPU requires**:

- massive parallelism -> more the merrier
- low register usage -> keep it simple
- small stack size -> **avoid recursion**

Classical approach not appropriate on GPU

In/On/Out transitions



CSG Complete Binary Tree Serialization -> simplifies GPU side

Geant4 solid -> CSG binary tree (leaf primitives, non-leaf operators, 4x4 transforms on any node)

Serialize to **complete binary tree** buffer:

- no need to deserialize, no child/parent pointers
- bit twiddling navigation avoids recursion
- simple approach profits from small size of binary trees
- BUT: very inefficient when unbalanced

Bit Twiddling Navigation

- parent(i) = i/2 = i >> 1
- leftchild(i) = 2*i = i << 1
- rightchild(i) = 2*i + 1 = (i << 1) + 1
- leftmost(height) = 1 << height

Height 3 complete binary tree with level order indices:

```
                                                             depth     elevation

                     1                                         0           3

            10                      11                         1           2

       100         101          110         111                2           1

   1000  1001   1010  1011   1100  1101   1110   1111          3           0
```

---

postorder_next(i,elevation) = i & 1 ? i >> 1 : (i << elevation) + (1 << elevation) ; // from pattern of bits

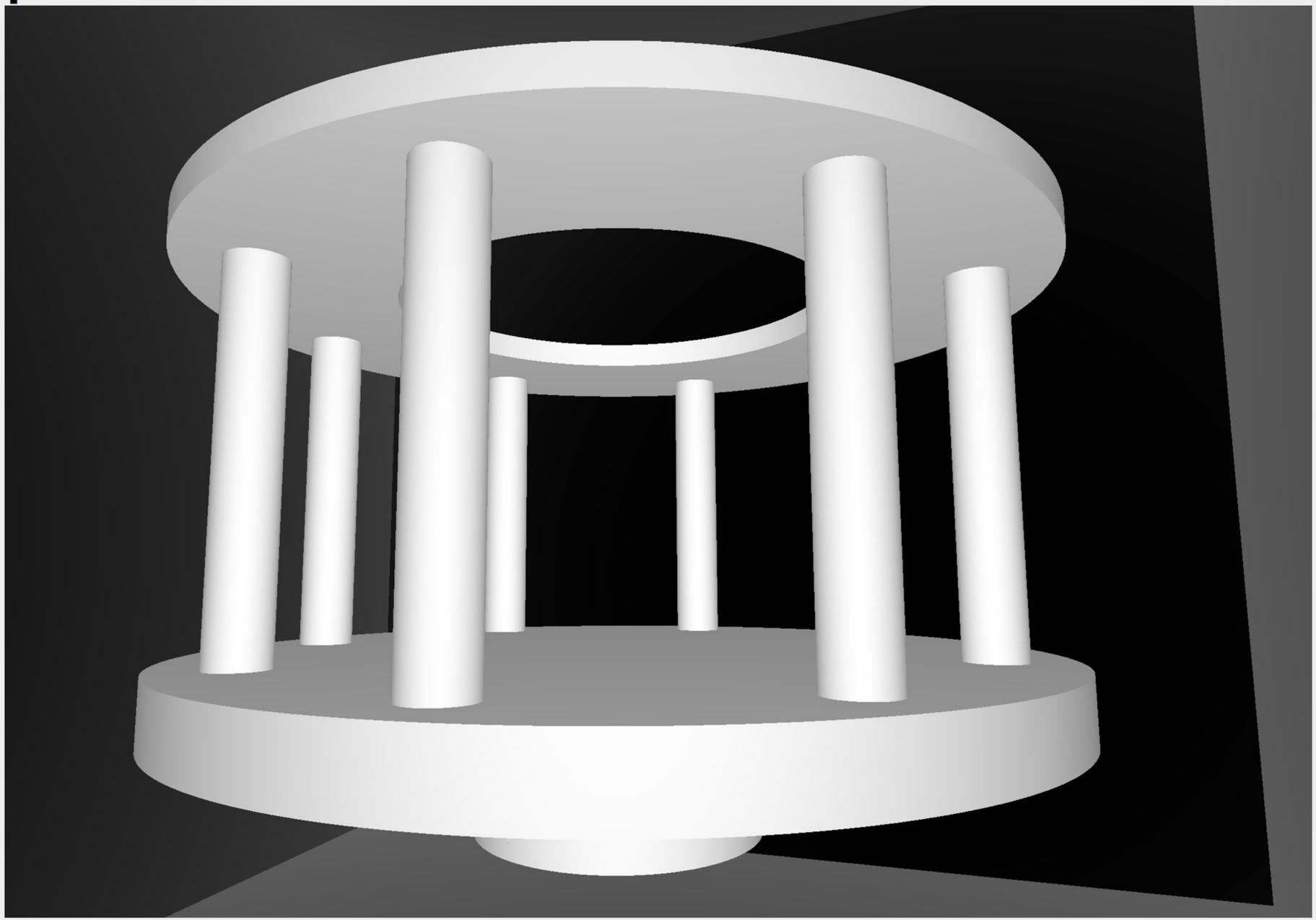
Postorder tree traverse visits all nodes, starting from leftmost, such that children are visited prior to their parents.

Evaluative CSG intersection Pseudocode : recursion emulated

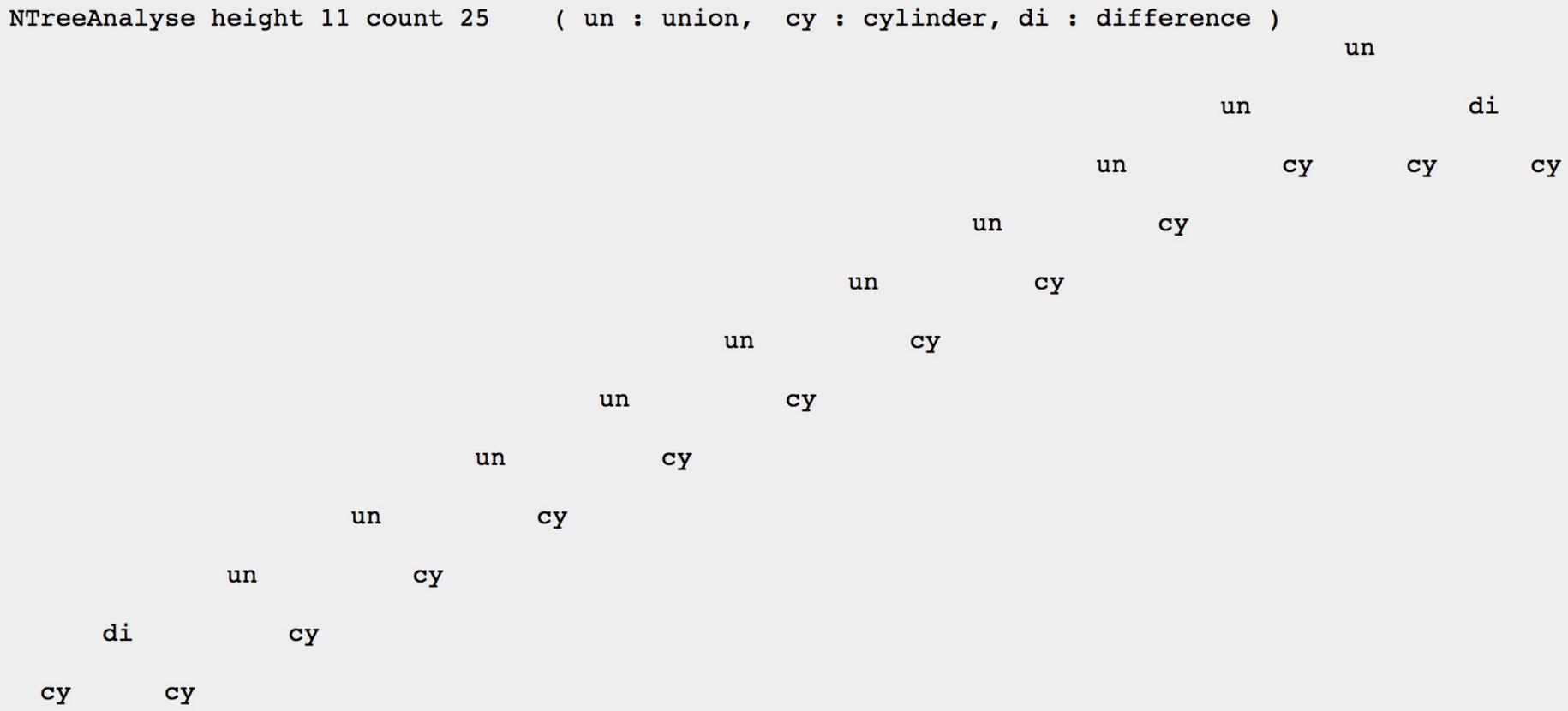
```
fullTree = PACK( 1 << height, 1 >> 1 )  // leftmost, parent_of_root(=0)
tranche.push(fullTree, ray.tmin)

while (!tranche.empty)         // stack of begin/end indices
{
    begin, end, tmin <- tranche.pop  ; node <- begin ;
    while( node != end )                    // over tranche of postorder traversal
    {
        elevation = height - TREE_DEPTH(node) ;
        if(is_primitive(node)){ isect <- intersect_primitive(node, tmin) ;  csg.push(isect) }
        else{
            i_left, i_right = csg.pop, csg.pop            // csg stack of intersect normals, t
            l_state = CLASSIFY(i_left, ray.direction, tmin)
            r_state = CLASSIFY(i_right, ray.direction, tmin)
            action = LUT(operator(node), leftIsCloser)(l_state, r_state)

            if(      action is ReturnLeft/Right)     csg.push(i_left or i_right)
            else if( action is LoopLeft/Right)
            {
                left = 2*node ; right = 2*node + 1 ;
                endTranche = PACK( node,  end );
                leftTranche = PACK(  left << (elevation-1), right << (elevation-1) )
                rightTranche = PACK(  right << (elevation-1),  node  )
                loopTranche = action ? leftTranche : rightTranche

                tranche.push(endTranche, tmin)
                tranche.push(loopTranche, tminAdvanced )  // subtree re-traversal with changed tmin
                break ; // to next tranche
            }
        }
        node <- postorder_next(node, elevation)         // bit twiddling postorder
    }
}
isect = csg.pop();          // winning intersect
```

https://bitbucket.org/simoncblyth/opticks/src/tip/optixrap/cu/csg_intersect_boolean.h

CSG Deep Tree : JUNO "fastener"



CSG Deep Tree : height 11 before balancing, too deep for GPU raytrace



CSG trees are non-unique

- many possible expressions of same shape
- some much more efficiently represented as complete binary trees

CSG Deep Tree : Positivize tree using De Morgan's laws

1st step to allow balancing : **Positivize** : remove CSG difference **di** operators

Positive form CSG Trees

Apply deMorgan pushing negations down tree

- **A - B -> A * !B**
- **!(A*B) -> !A + !B**
- **!(A+B) -> !A * !B**
- **!(A - B) -> !(A*!B) -> !A + B**

End with only UNION, INTERSECT operators, and some complemented leaves.

COMMUTATIVE -> easily rearranged

CSG Deep Tree : height 4 after balancing, OK for GPU raytrace

```
NTreeAnalyse height 4 count 25
                                                  un
                                un                                                  un
                    un                                  un                                  in
          un                  un                  un                  cy          in          cy      !cy
    cy        cy        cy        cy        cy        cy        cy        cy              cy      !cy
```

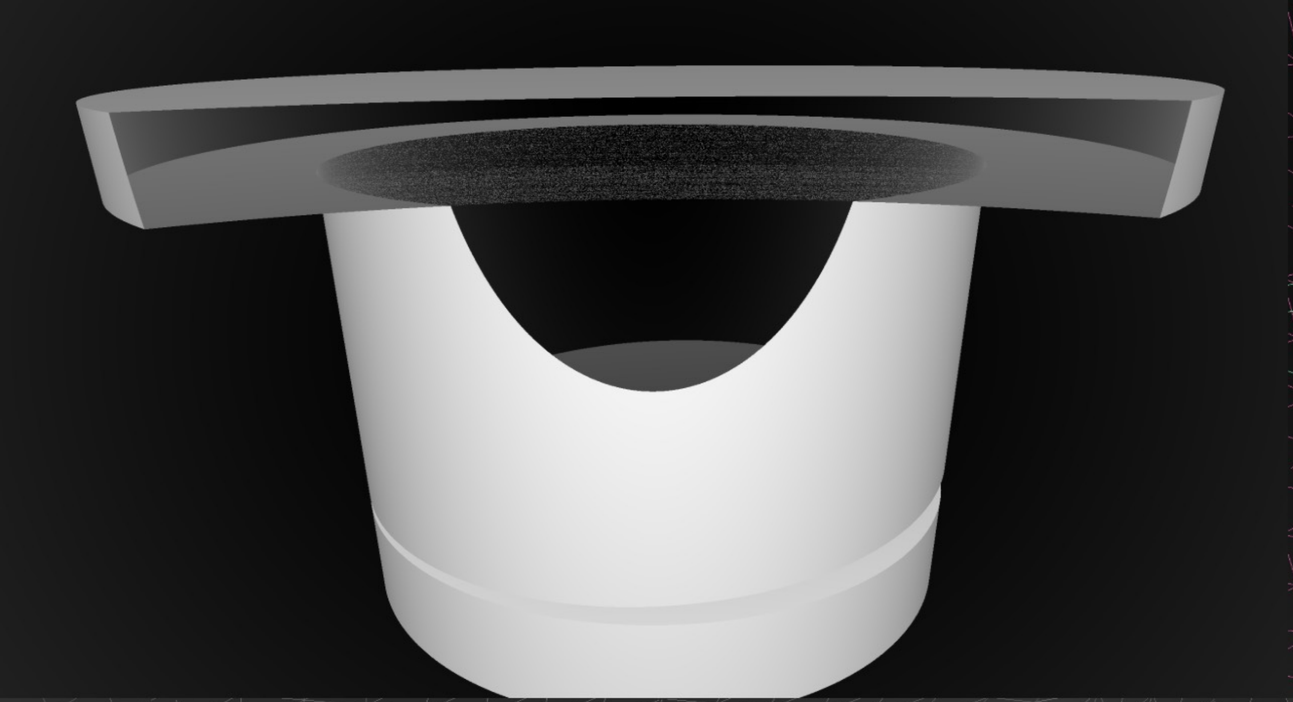
un : union, in : intersect, cy : cylinder, !cy : complemented cylinder

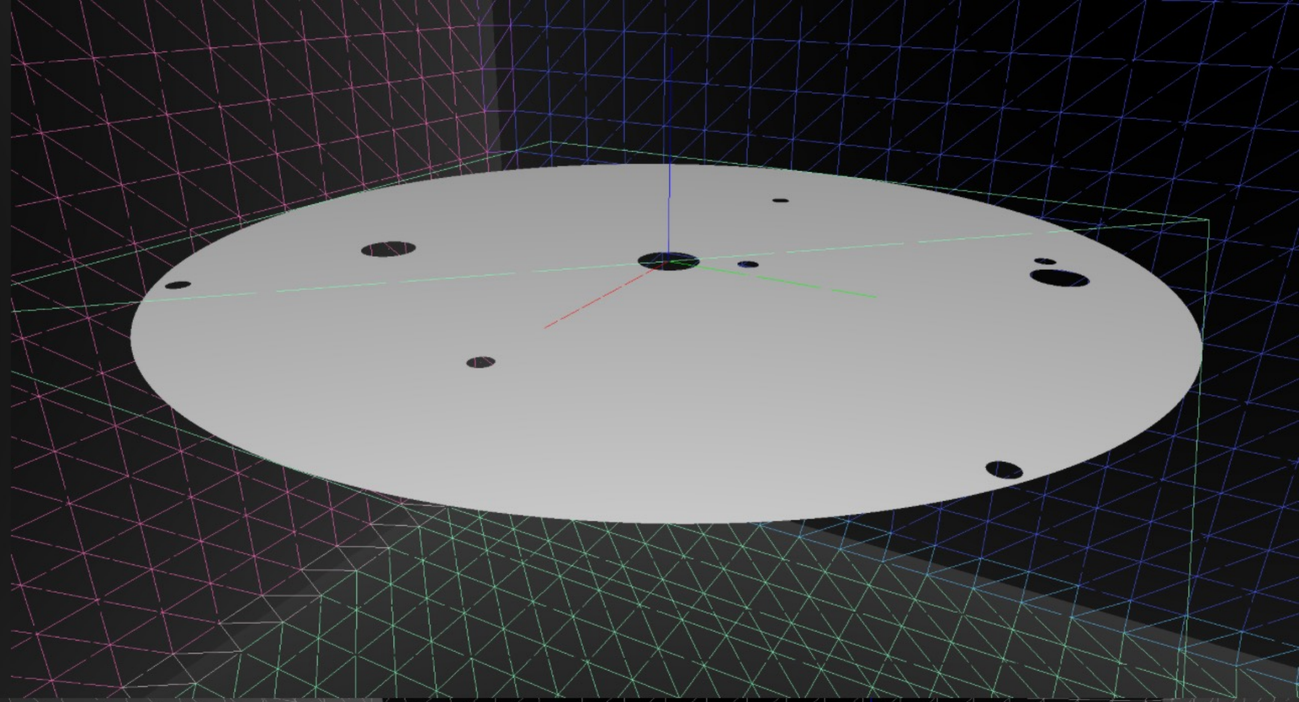
Balancing positive tree:

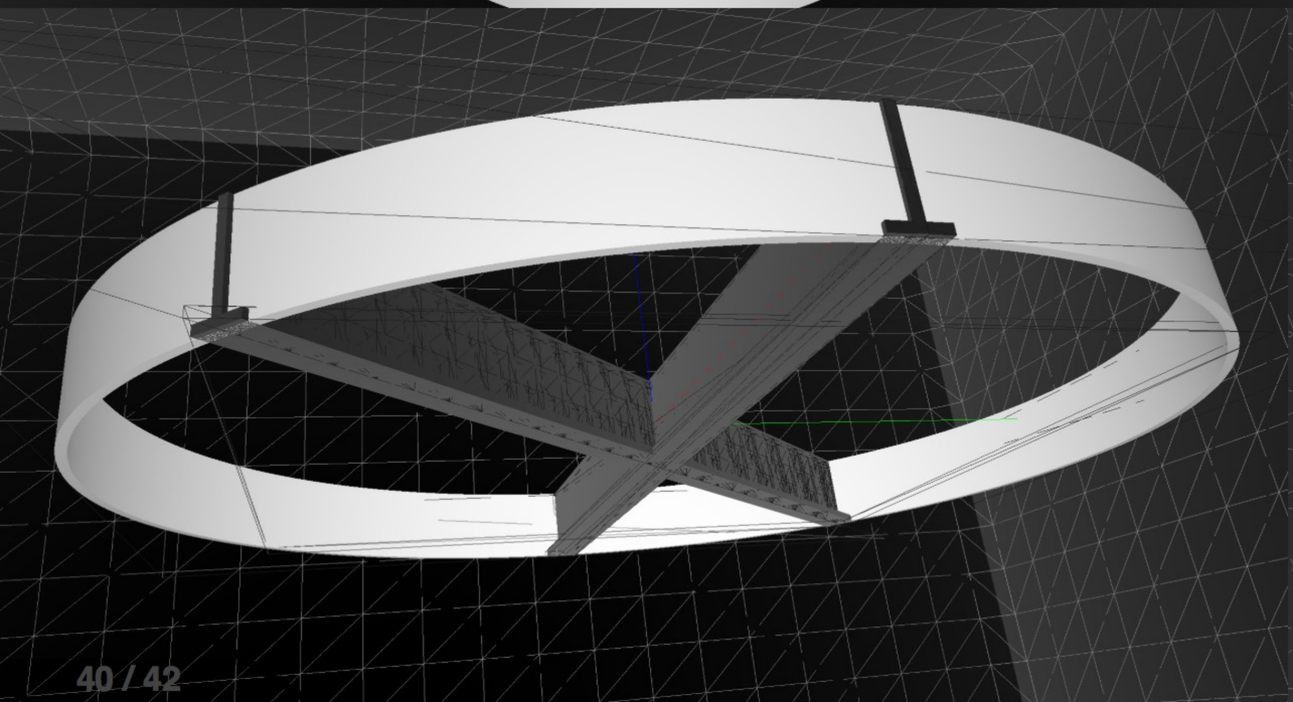
1. classify tree operators and their placement
    - mono-operator trees can easily be rearranged as union **un** and intersection **in** operators are **commutative**
    - mono-operator above bileaf level can also easily be rearranged as the bileaf can be split off and combined
2. create complete binary tree of appropriate size filled with placeholders
3. populate the tree replacing placeholders
4. prune (pull primitives up to avoid placeholder pairings)

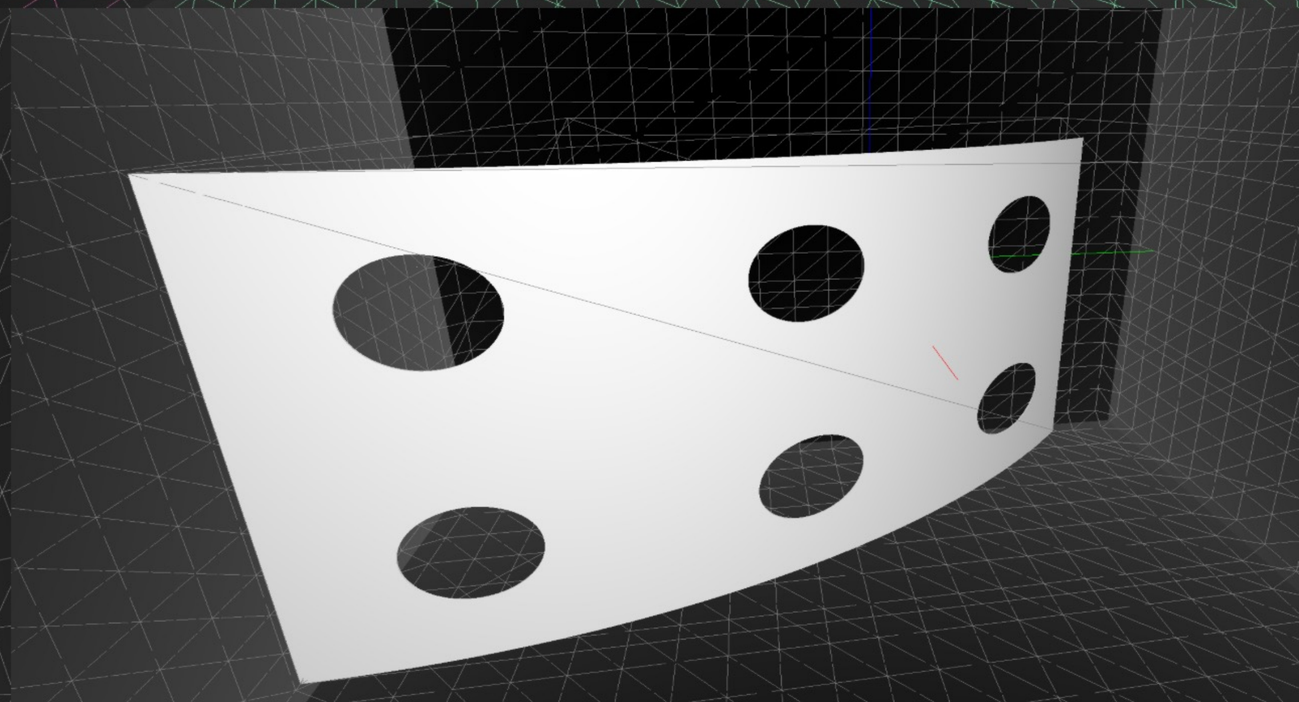
Not a general balancer : but succeeds with all CSG solid trees from Daya Bay and JUNO so far

https://bitbucket.org/simoncblyth/opticks/src/default/npy/NTreeBalance.cpp









Torus : much more difficult/expensive than other primitives

3D parametric ray : **ray(x,y,z;t) = rayOrigin + t * rayDirection**

- ray-torus intersection -> solve quartic polynomial in **t**
- A t^4 + B t^3 + C t^2 + D t + E = 0

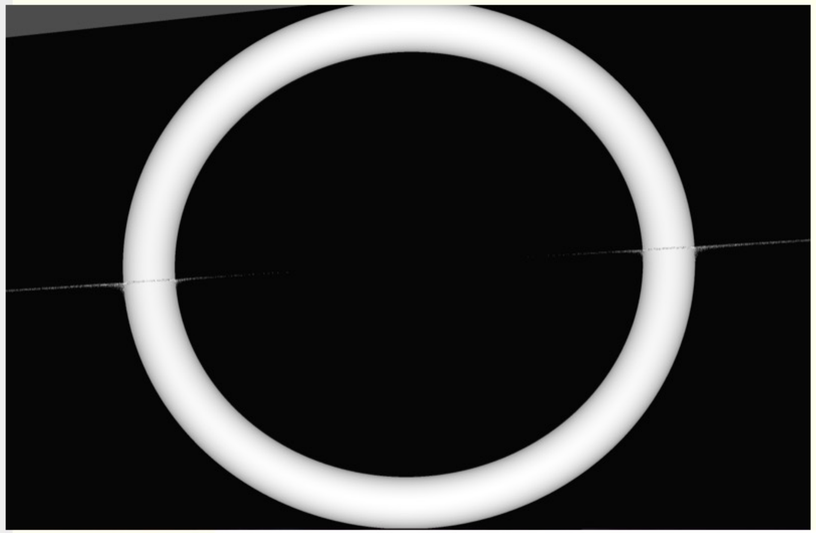
High order equation

- very large difference between coefficients
- varying ray -> wide range of very coefficients
- numerically problematic, requires double precision
- several mathematical approaches used, work in progress

Best Solution : replace torus

- eg model PMT neck with hyperboloid, not cylinder-torus

Torus artifacts



$$\left(R - \sqrt{x^2 + y^2}\right)^2 + z^2 = r^2,$$

$$\left(x^2 + y^2 + z^2 + R^2 - r^2\right)^2 = 4R^2\left(x^2 + y^2\right)$$

Torus : different artifacts as change implementation/params/viewpoint

- Only use Torus when there is no alternative
- especially avoid CSG combinations with Torus