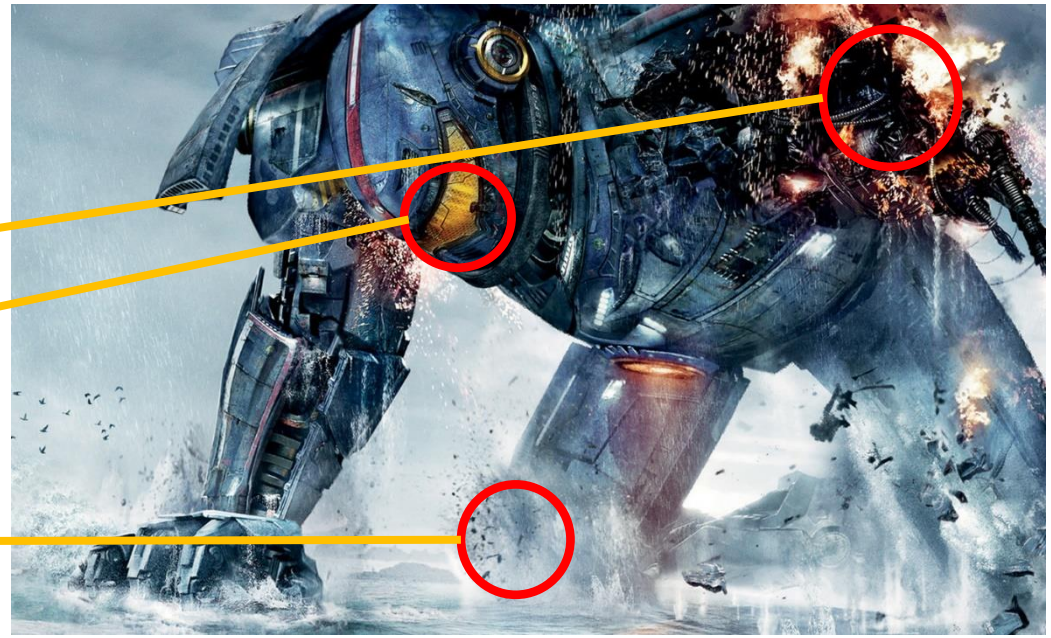
Two GPU Algorithms for Raytracing

Srinath.R
Prof.P.J.Narayanan
Centre for Visual Information Technology
IIIT Hyderabad

Production Rendering

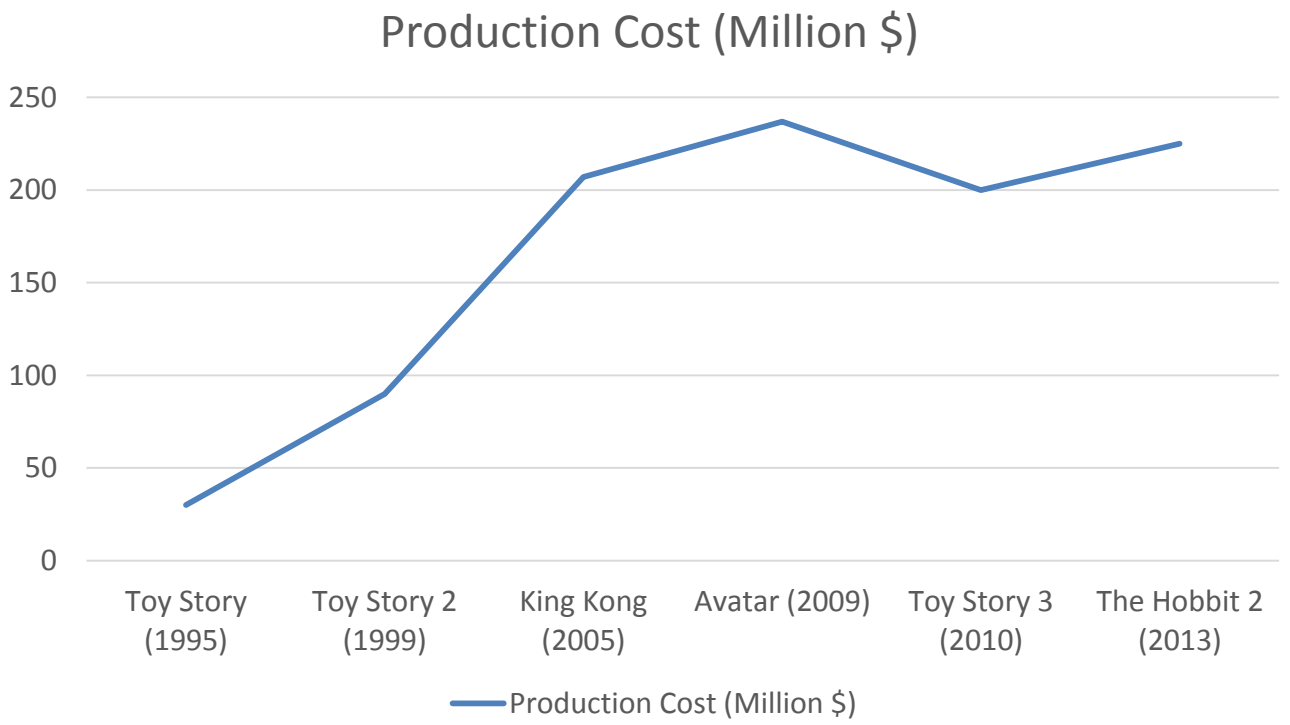
- Visual fidelity over responsiveness.
- Complex geometry.
- Complex lighting.
- Complex materials.
- Complex simulations.



Pacific Rim (2013)

Rising Costs?

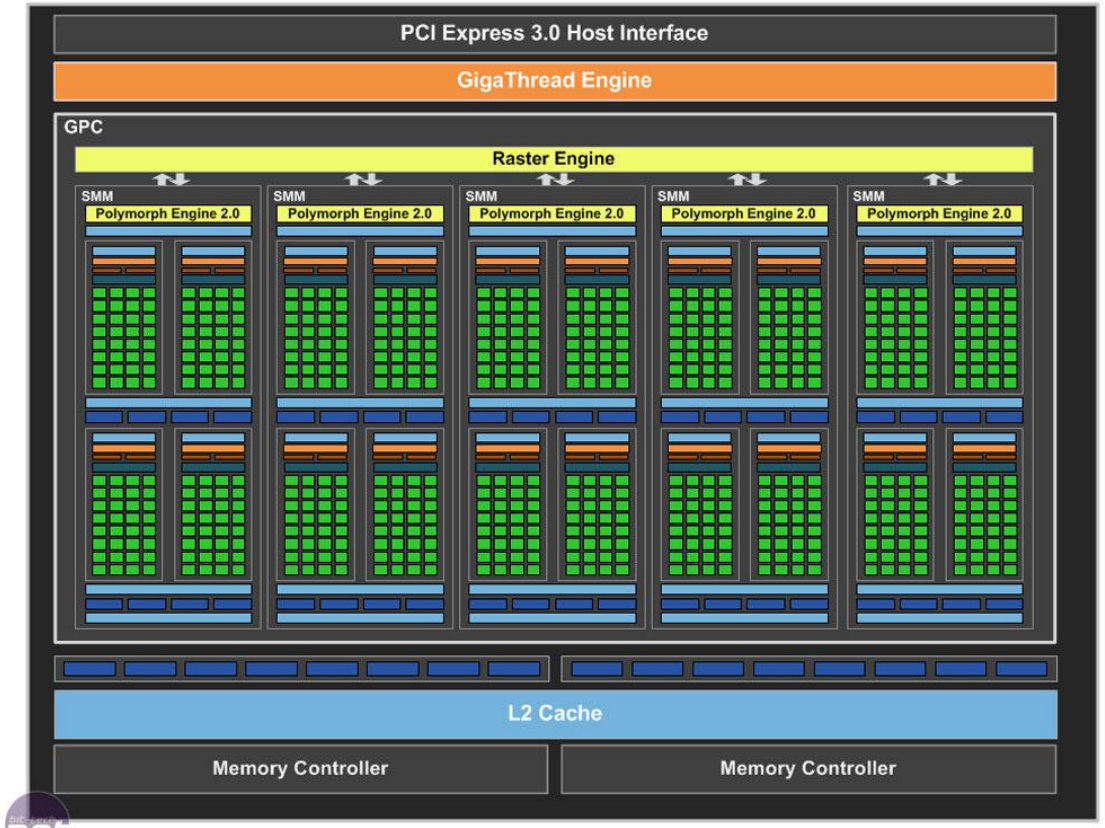
- Cost of making visual effects/animation has grown very rapidly.
- Artists as well as technical costs.
- Rendering done in large render farms.
- Hardware prices has gone down.
- As technology advances, rendering time remains constant (Blinn's law)



Commercial Renderers

- Traditionally production rendering was REYES based.
- Now moving towards fully ray traced pipeline.
- Commercial renderers were purely CPU based.
- Certain parts of the renderer use the GPU.
- Very few fully GPU based renderers.

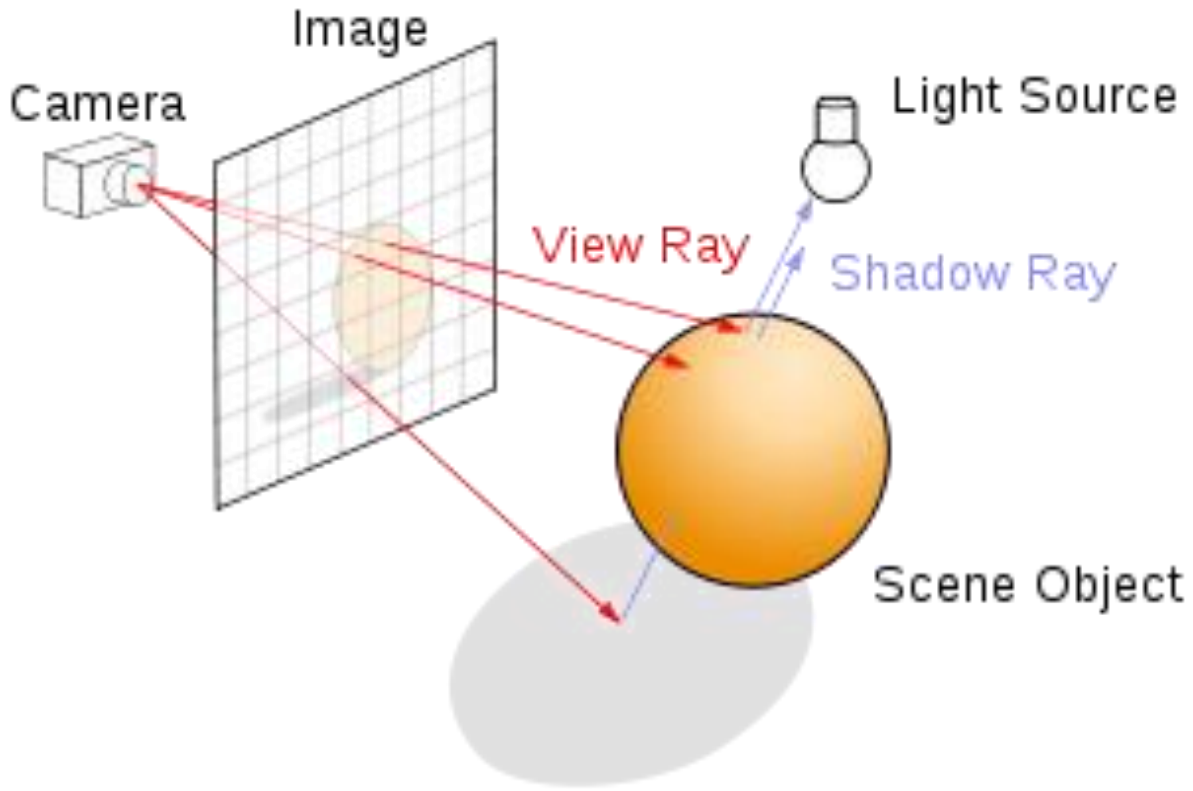
Why GPU Raytracing ?



NVIDIA Maxwell Architecture

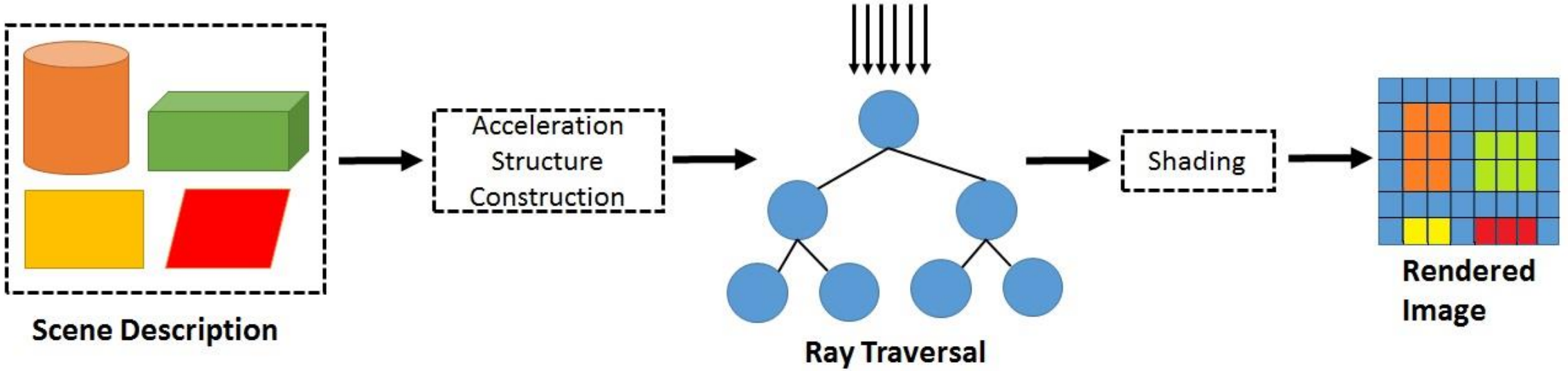
- Large parallel computation engines.
- Thousands of threads in flight.
- Very high bandwidth memory.
- Raytracing is embarrassingly parallel.
- Naïve approach – Each ray handled by one thread.
- Very effective cost/performance factors.

Raytracing 101

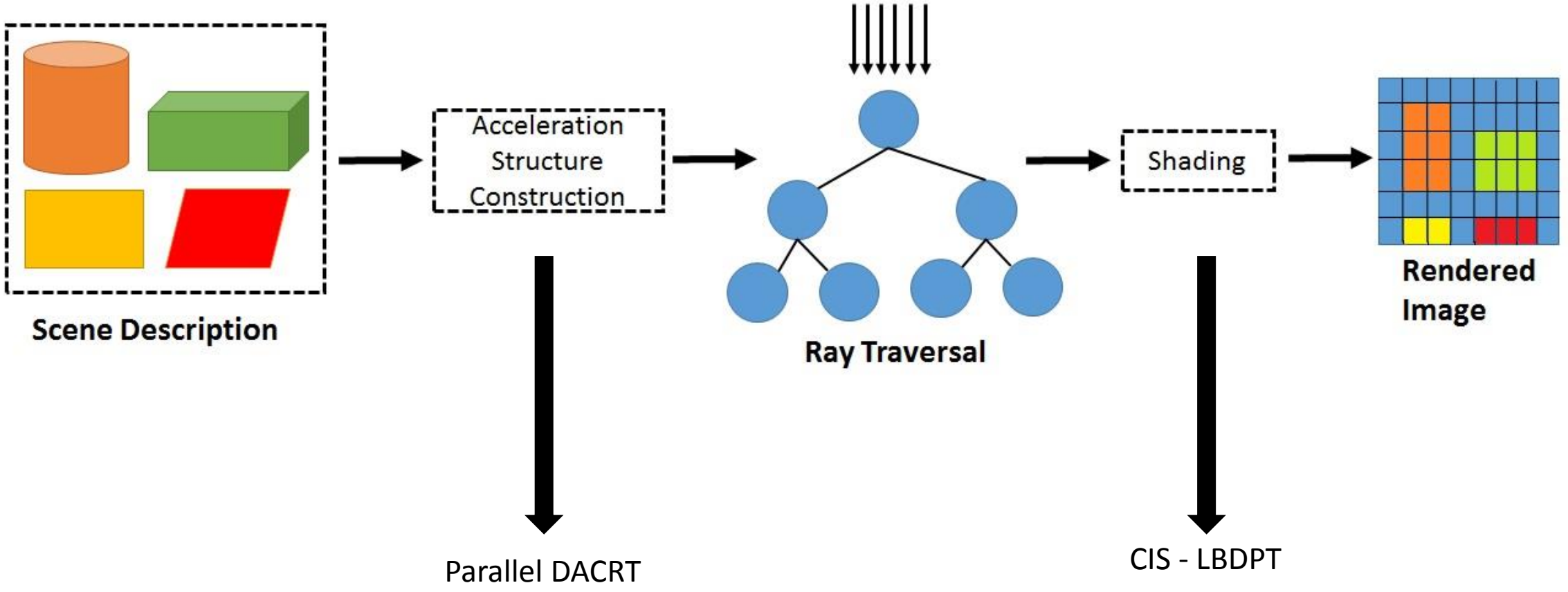


Raytracing in work (Courtesy: Wikipedia)

Raytracing Pipeline

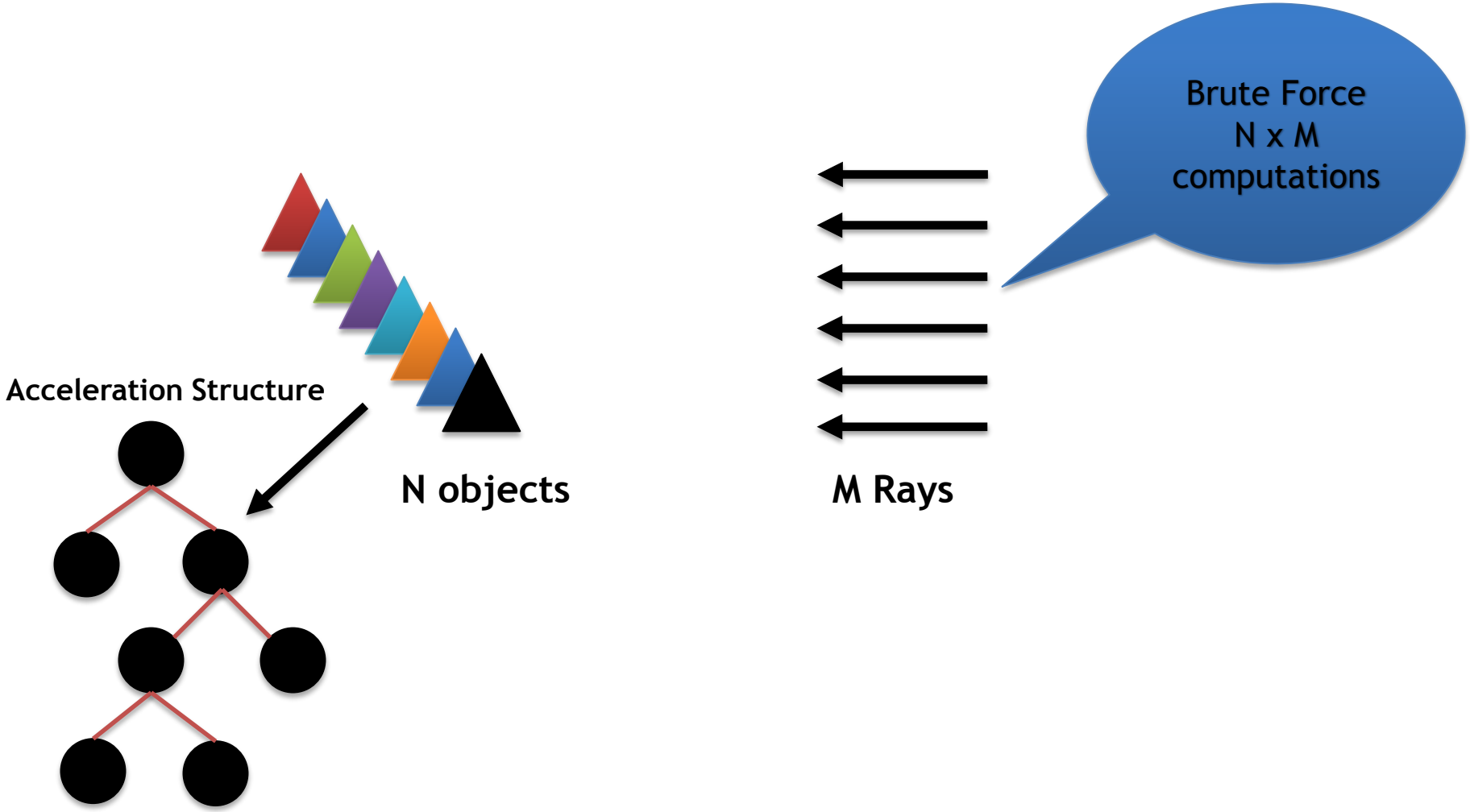


Our Work


Scene Description
Acceleration Structure Construction
Ray Traversal
Shading
Rendered Image
Parallel DACRT
CIS - LBDPT

Acceleration Structures

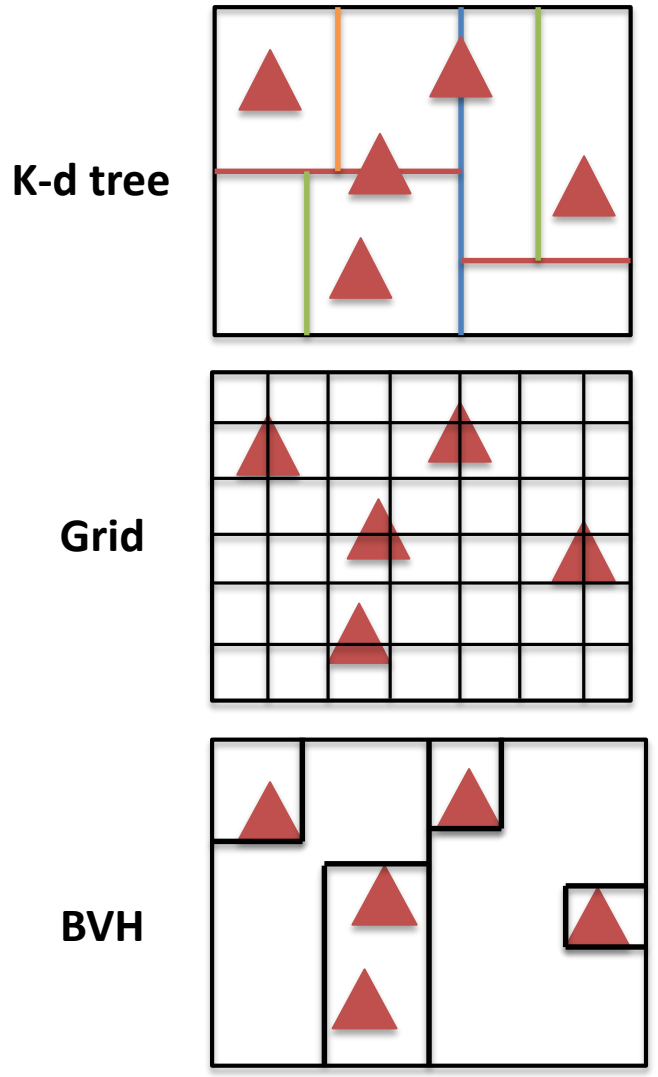
Why Acceleration Structures?



Acceleration Structures (AS)

- Spatial Partitioning
  - K-d trees [Bentley 1975]
  - Octrees [Glassner 1984]
  - BSP trees [Fuchs et al. 1980]
  - Grids [Fujimoto et al. 1988]

- Object Partitioning
  - BVH [Rubin and Whitten 1980]



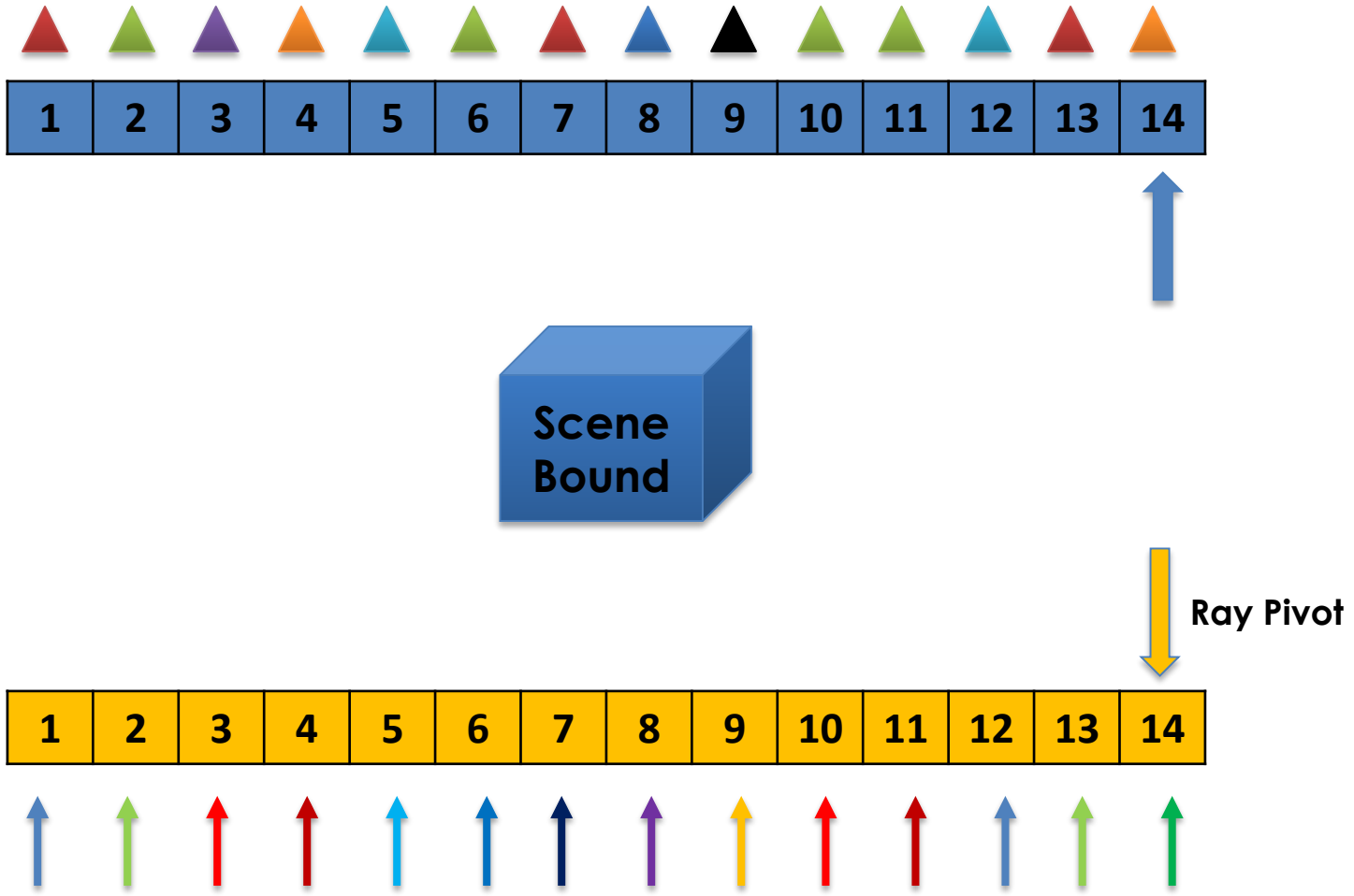
Divide and Conquer Ray Tracing (DACRT)

- Two basic steps of ray tracing
  - Construct an acceleration structure(AS)
  - Trace rays using it.

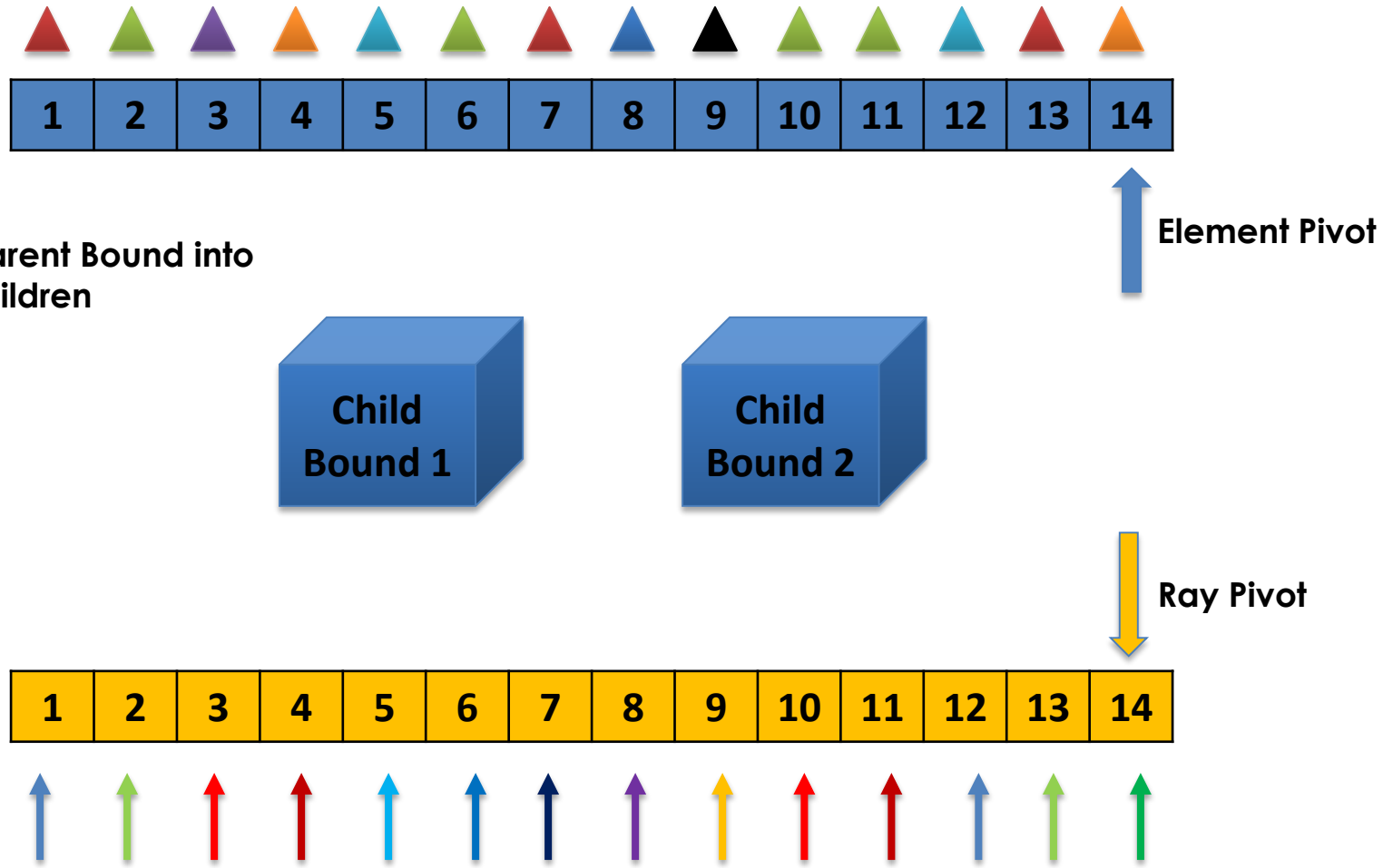
- Get away with one step instead of two?
  - Construct and Trace together?
  - Don’t construct any explicit acceleration structure?

- Divide and Conquer Ray Tracing
  - First presented by Mora et al in 2011.
  - Similar to quicksort.
  - Simple serial CPU algorithm requiring very little memory.

Serial DACRT - Working

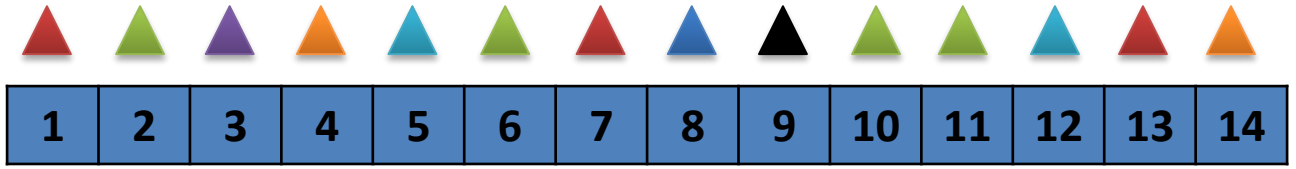


Serial DACRT - Working



- Split Parent Bound into two children

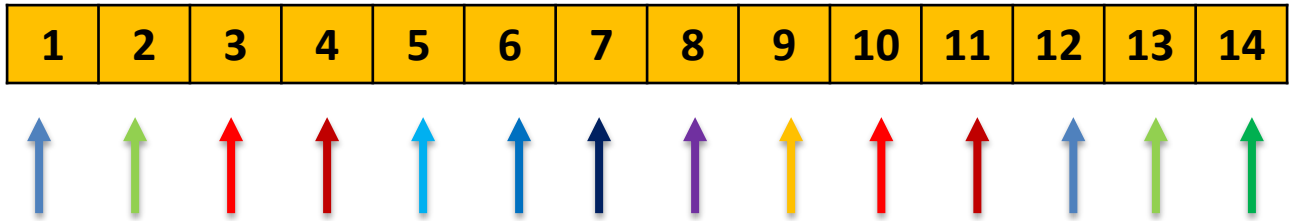
Serial DACRT - Working



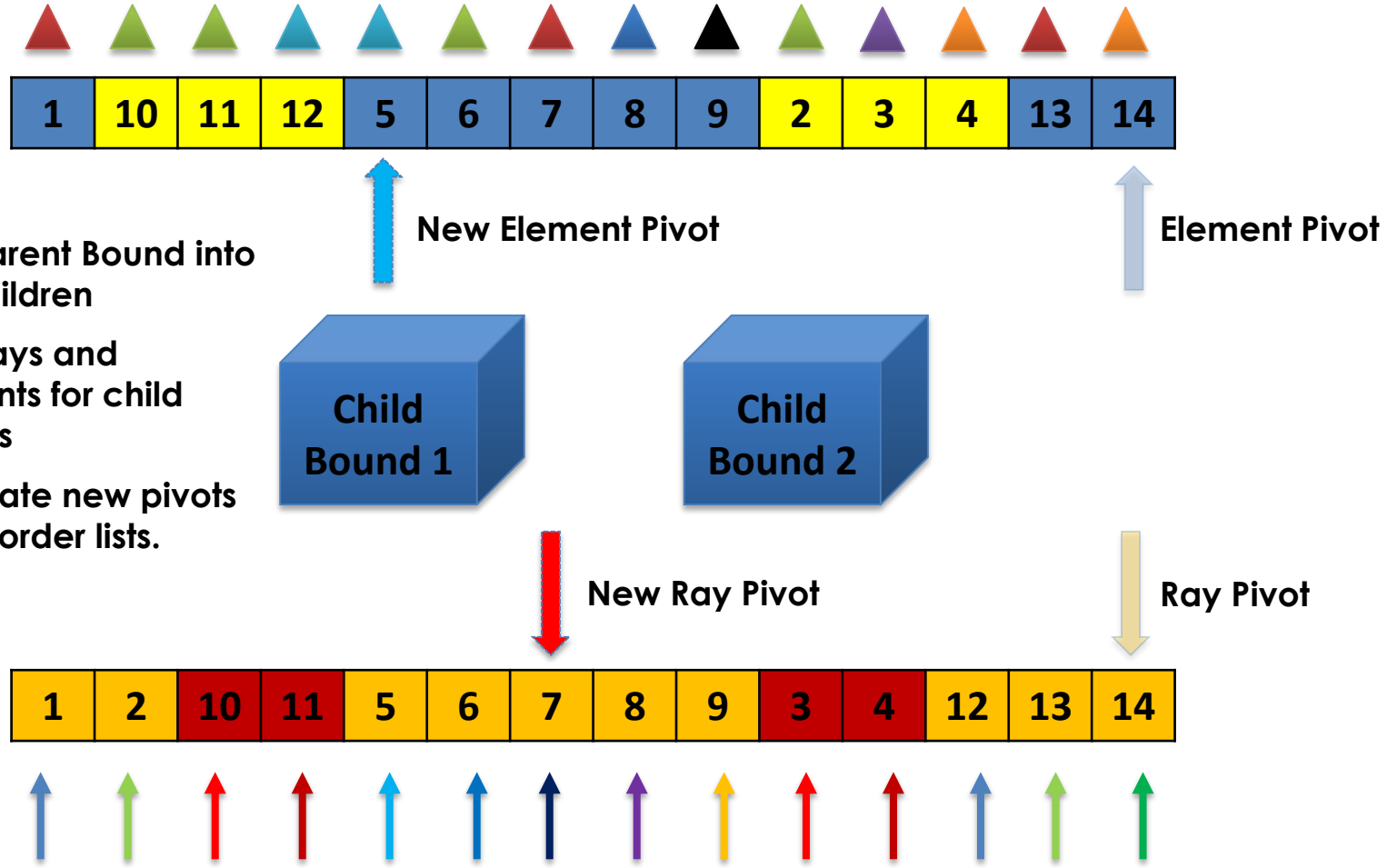
Element Pivot

- Split Parent Bound into two children
- Filter rays and elements for child bounds

Ray Pivot

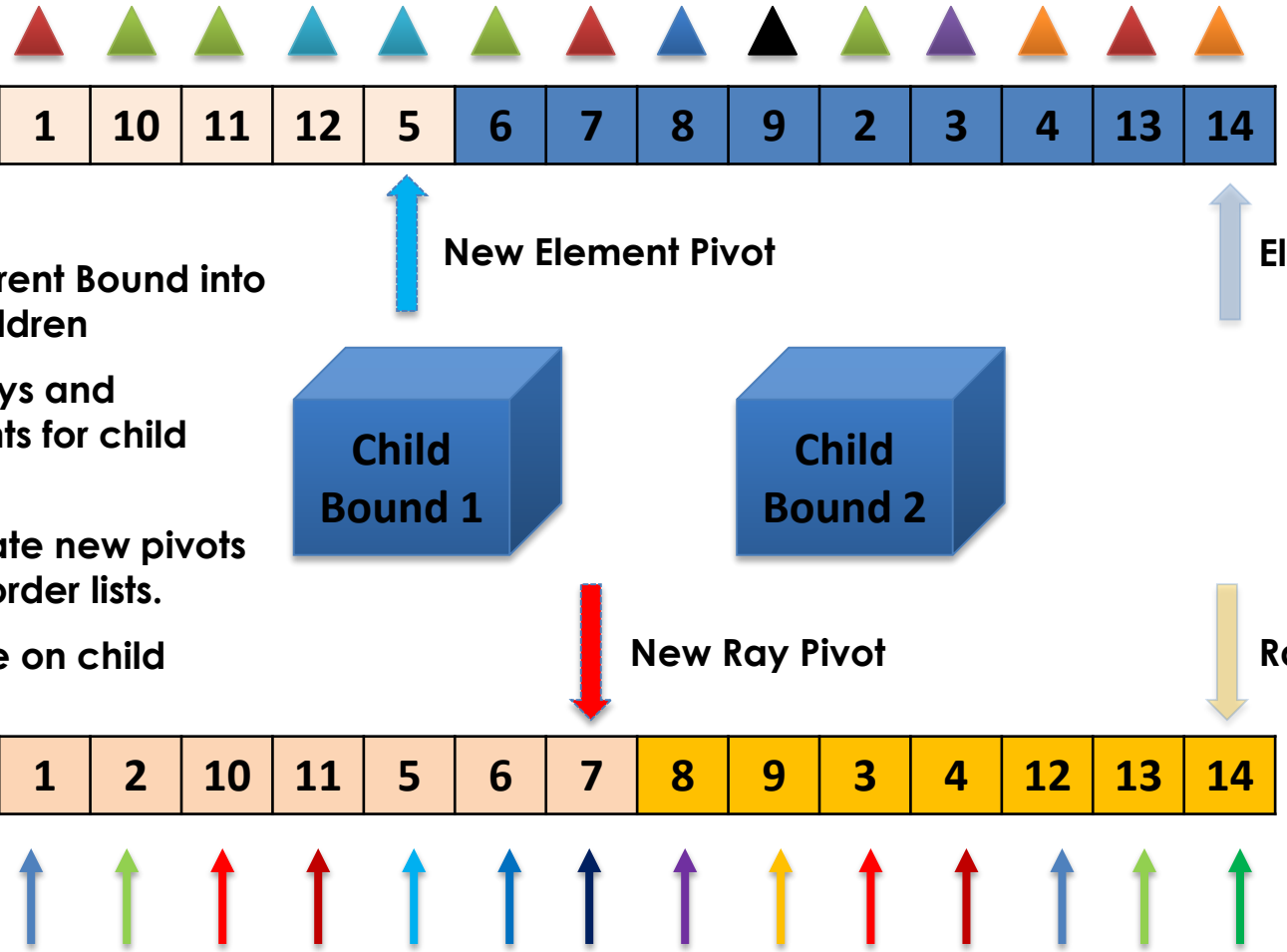


Serial DACRT - Working



- Split Parent Bound into two children
- Filter rays and elements for child bounds
- Calculate new pivots and reorder lists.

Serial DACRT - Working



- Split Parent Bound into two children
- Filter rays and elements for child bounds
- Calculate new pivots and reorder lists.
- Recurse on child node.

Serial DACRT - Overview

- DACRT(Space **S**, Set of Rays **R,** Set of Elements **E**)

  If number of rays/elements are small

    Compute brute force intersections

  Else

    Split **S** into ‘N’ children

    Perform filtering for each child

    Recursive DACRT on each child

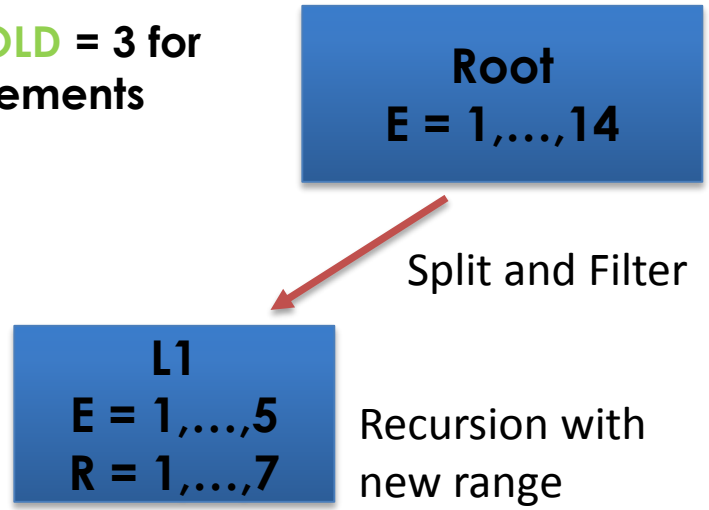
DACRT – Recursion Tree

Assume THRESHOLD = 3 for both rays and elements

Root
E = 1,…,14

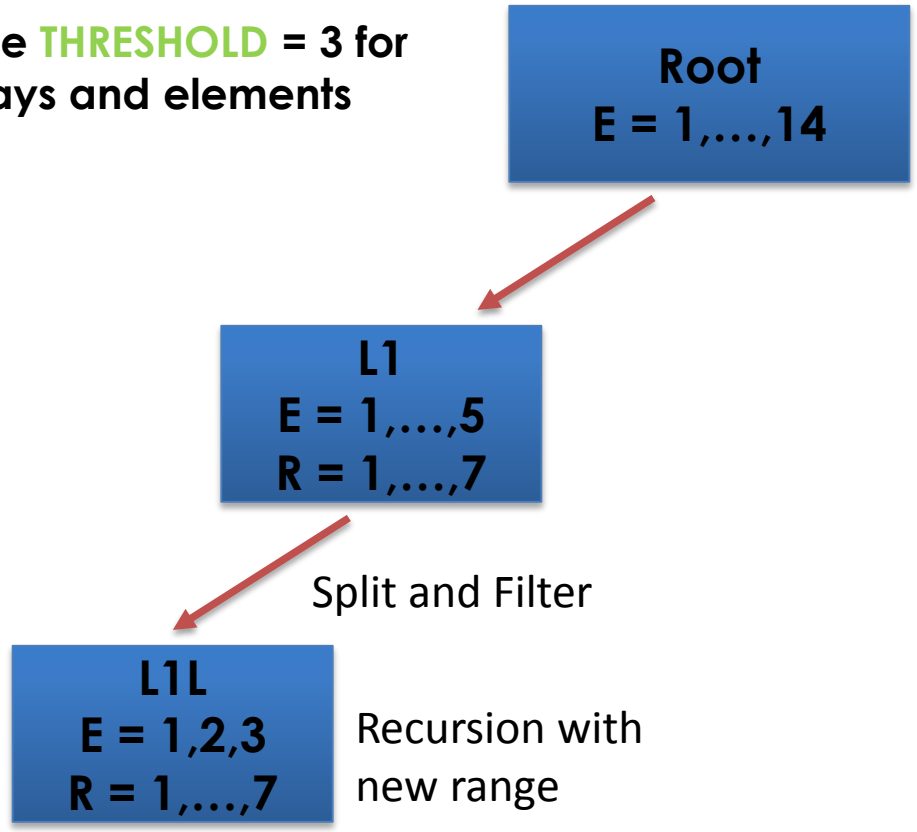
DACRT – Recursion Tree

Assume THRESHOLD = 3 for both rays and elements



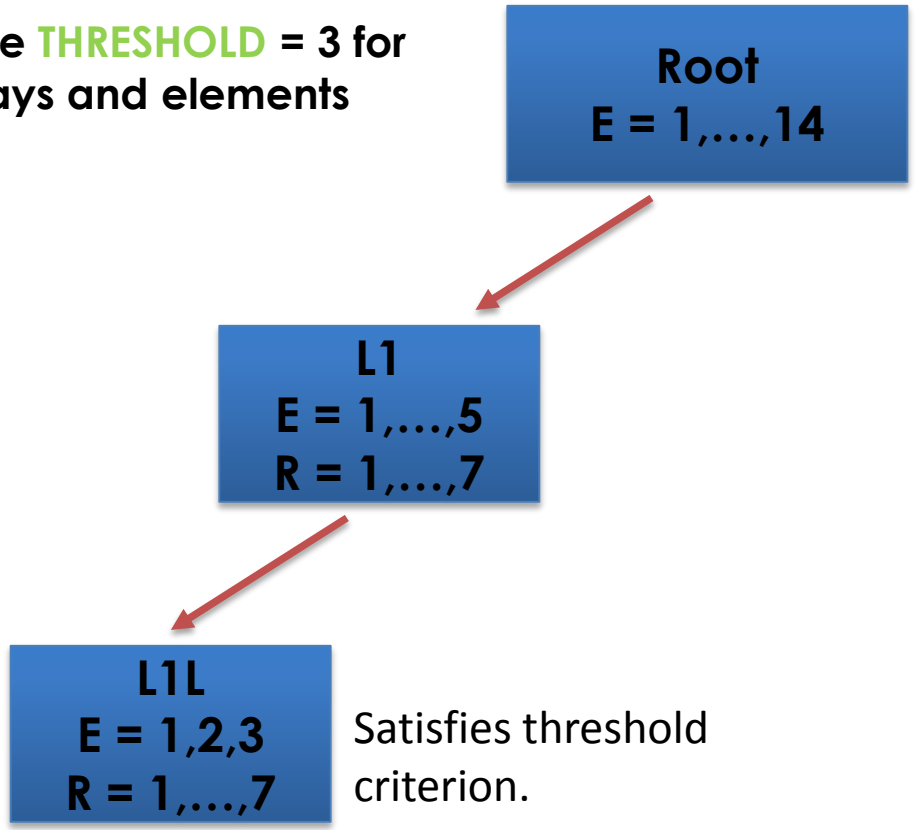
DACRT – Recursion Tree

Assume THRESHOLD = 3 for both rays and elements



DACRT - Recursion Tree

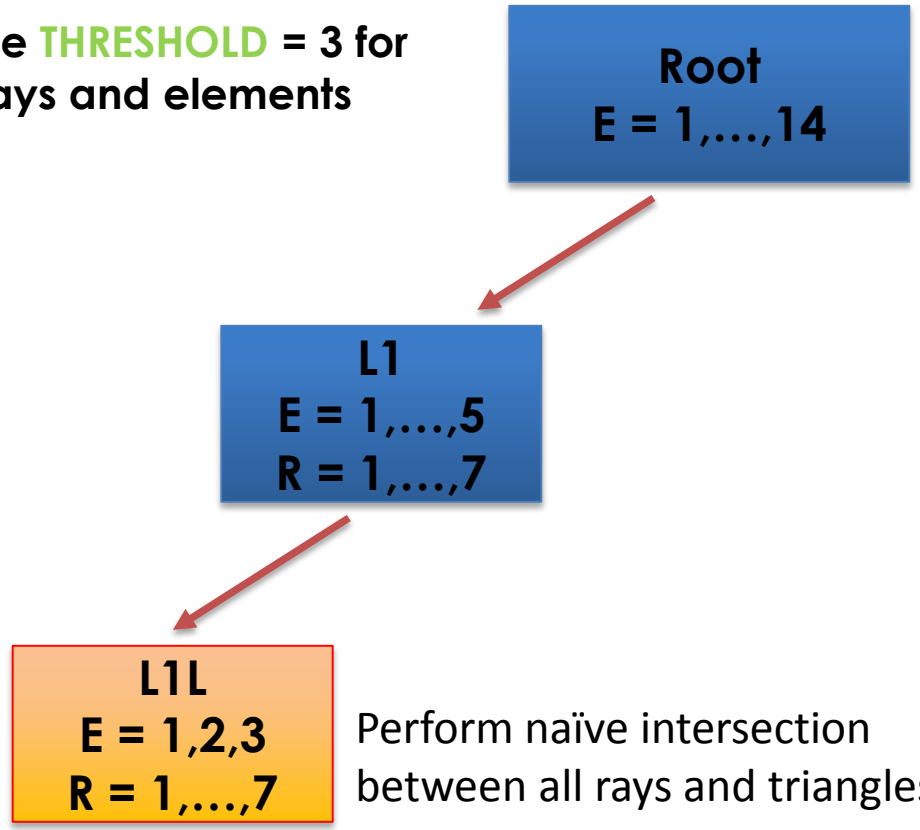
Assume THRESHOLD = 3 for both rays and elements



Satisfies threshold criterion.

DACRT – Recursion Tree

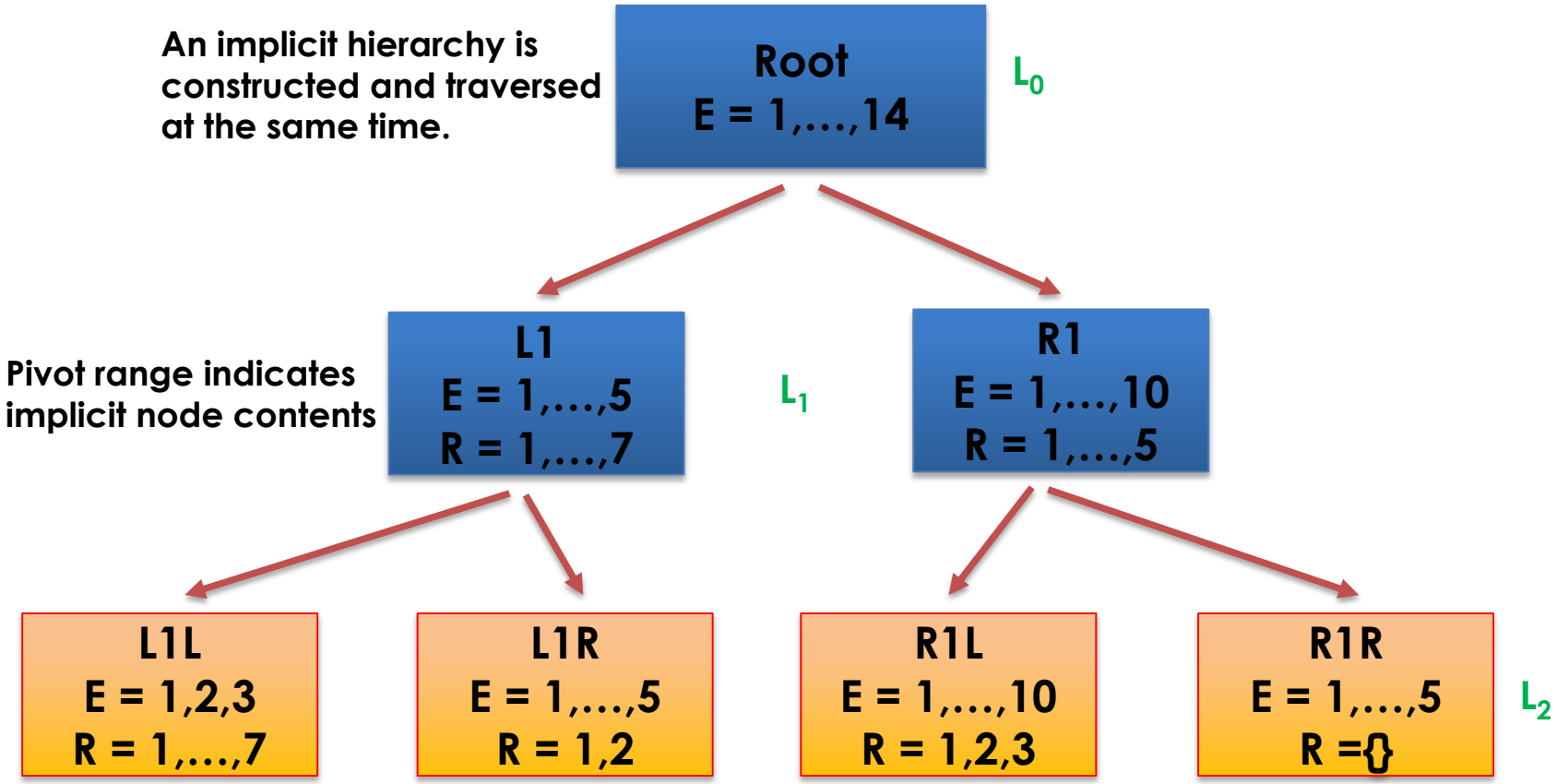
Assume THRESHOLD = 3 for both rays and elements



Perform naïve intersection between all rays and triangles

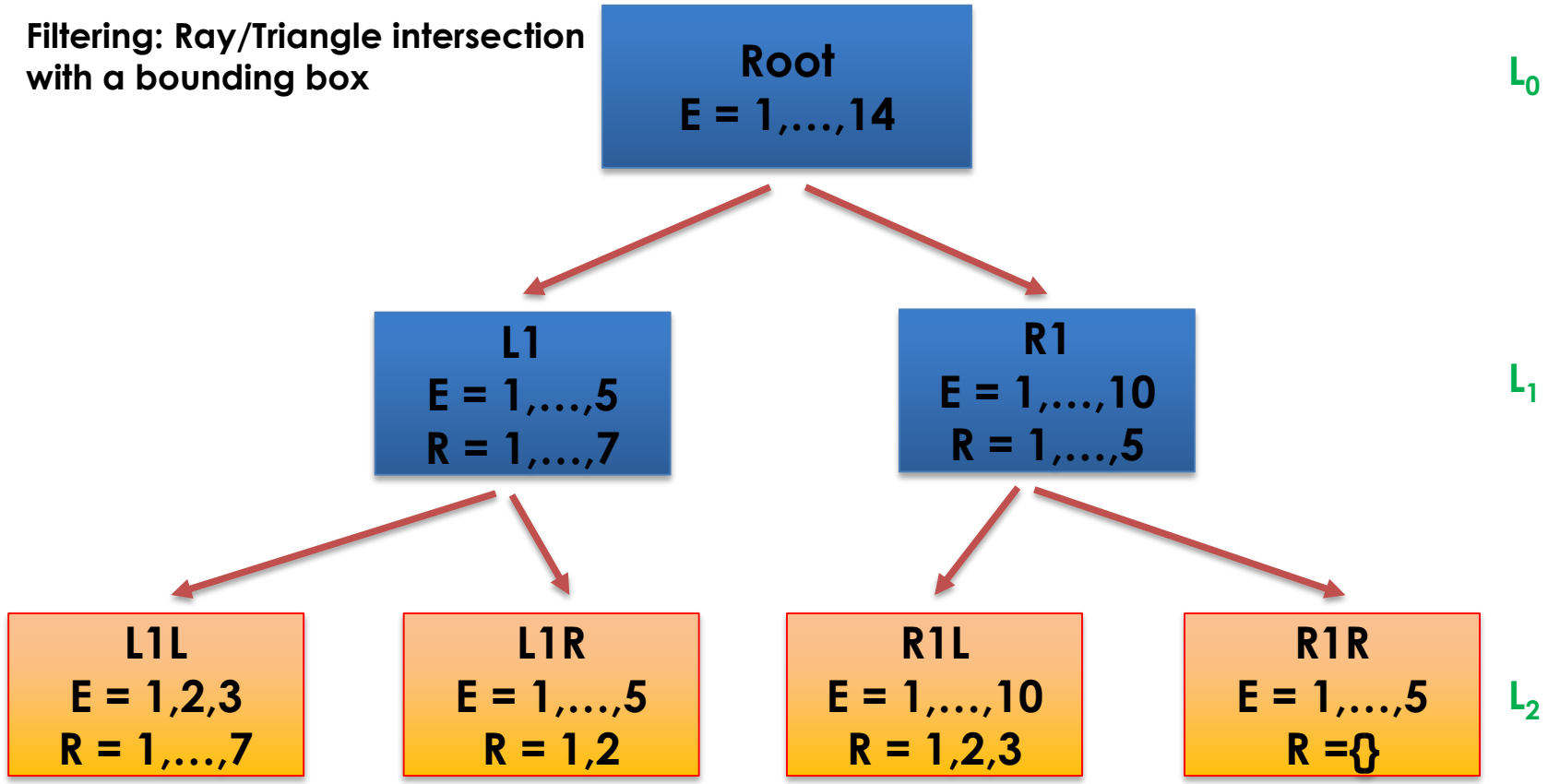
DACRT – Entire Recursion Tree

An implicit hierarchy is constructed and traversed at the same time.



DACRT – Source of Parallelism

Filtering: Ray/Triangle intersection with a bounding box

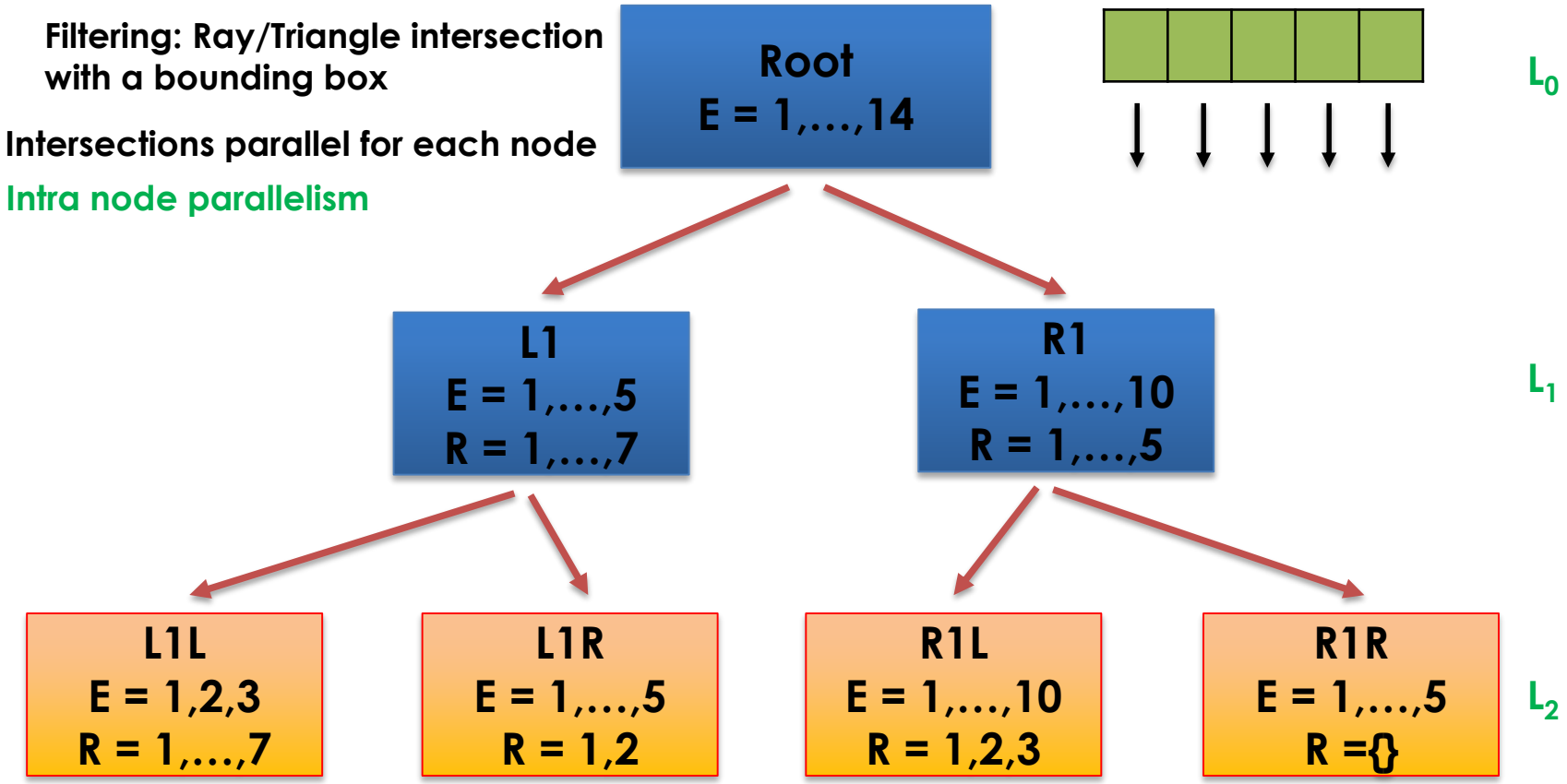


DACRT – Source of Parallelism

Filtering: Ray/Triangle intersection with a bounding box

Intersections parallel for each node

Intra node parallelism



DACRT – Source of Parallelism

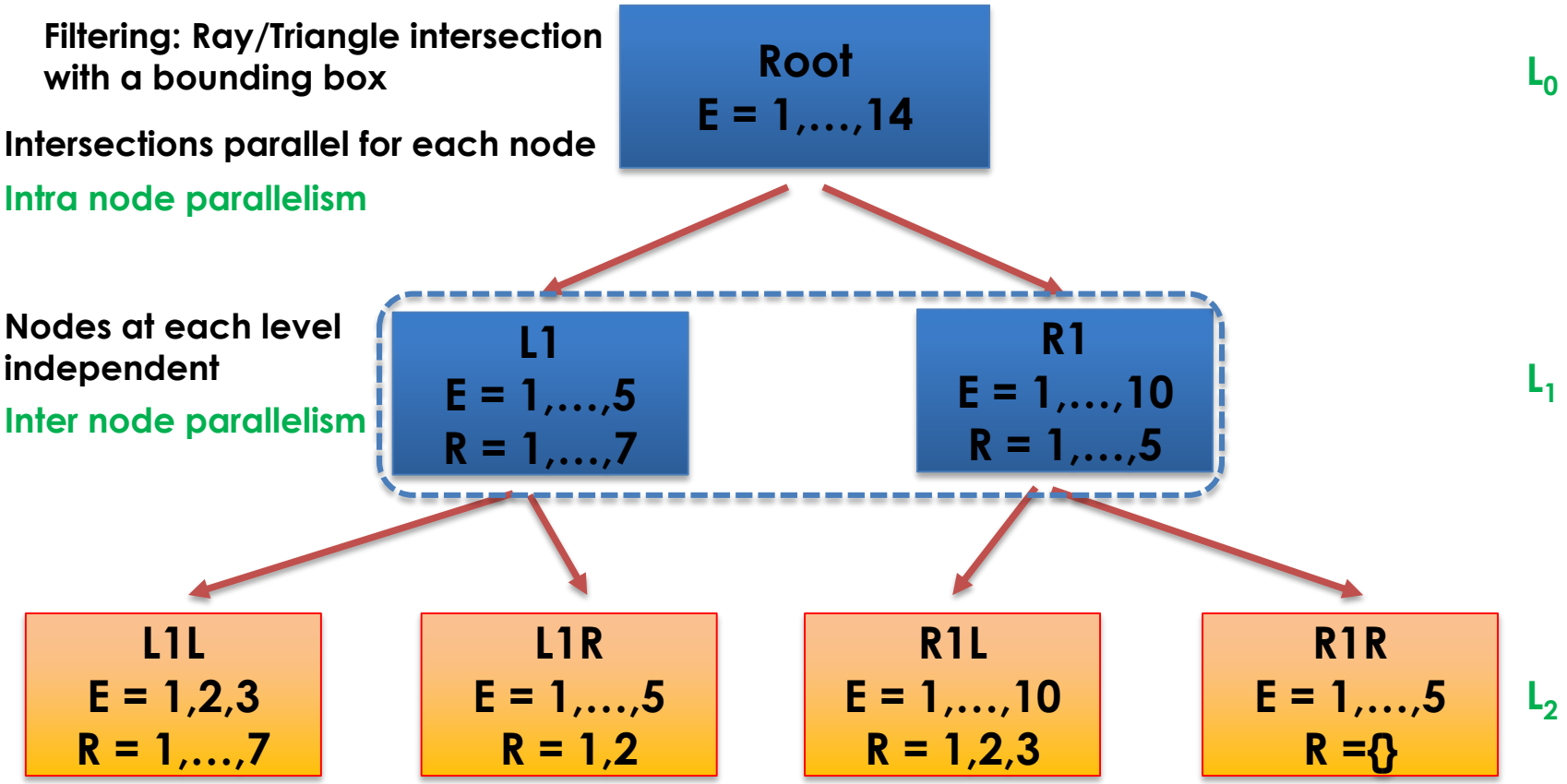
Filtering: Ray/Triangle intersection with a bounding box

Intersections parallel for each node

Intra node parallelism

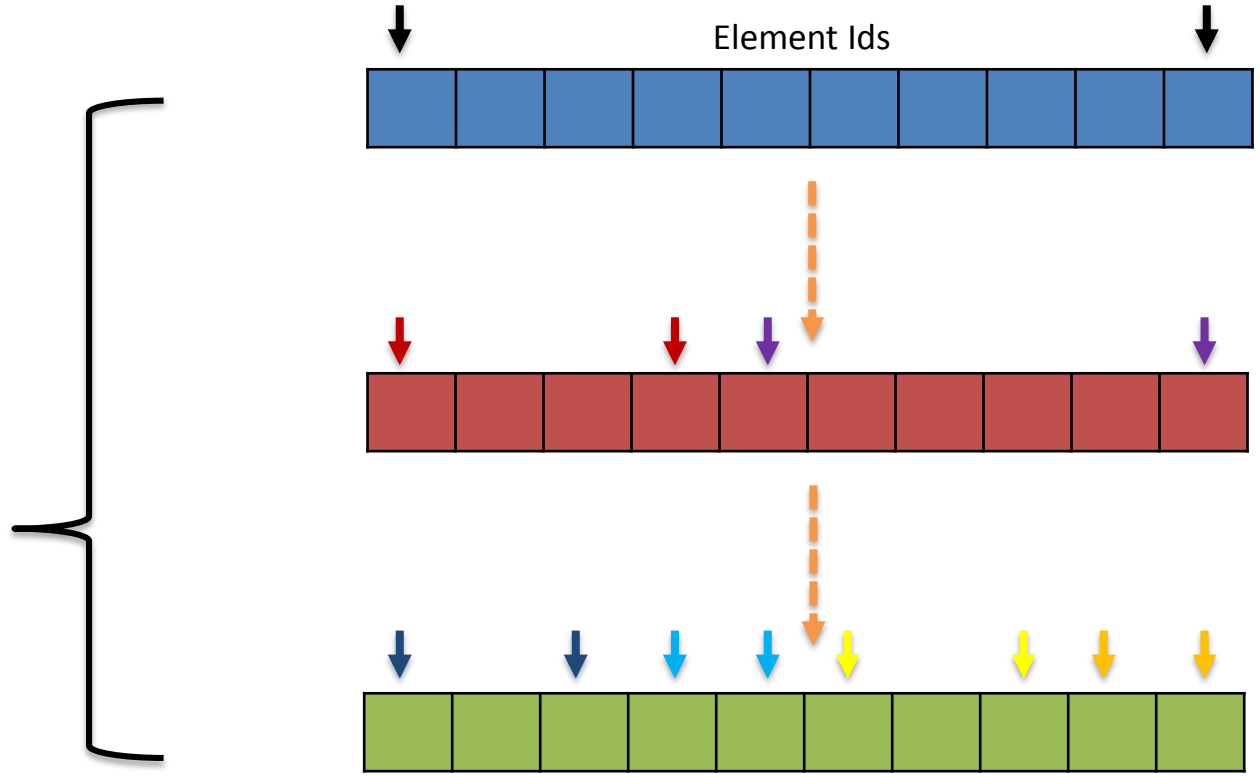
Nodes at each level independent

Inter node parallelism



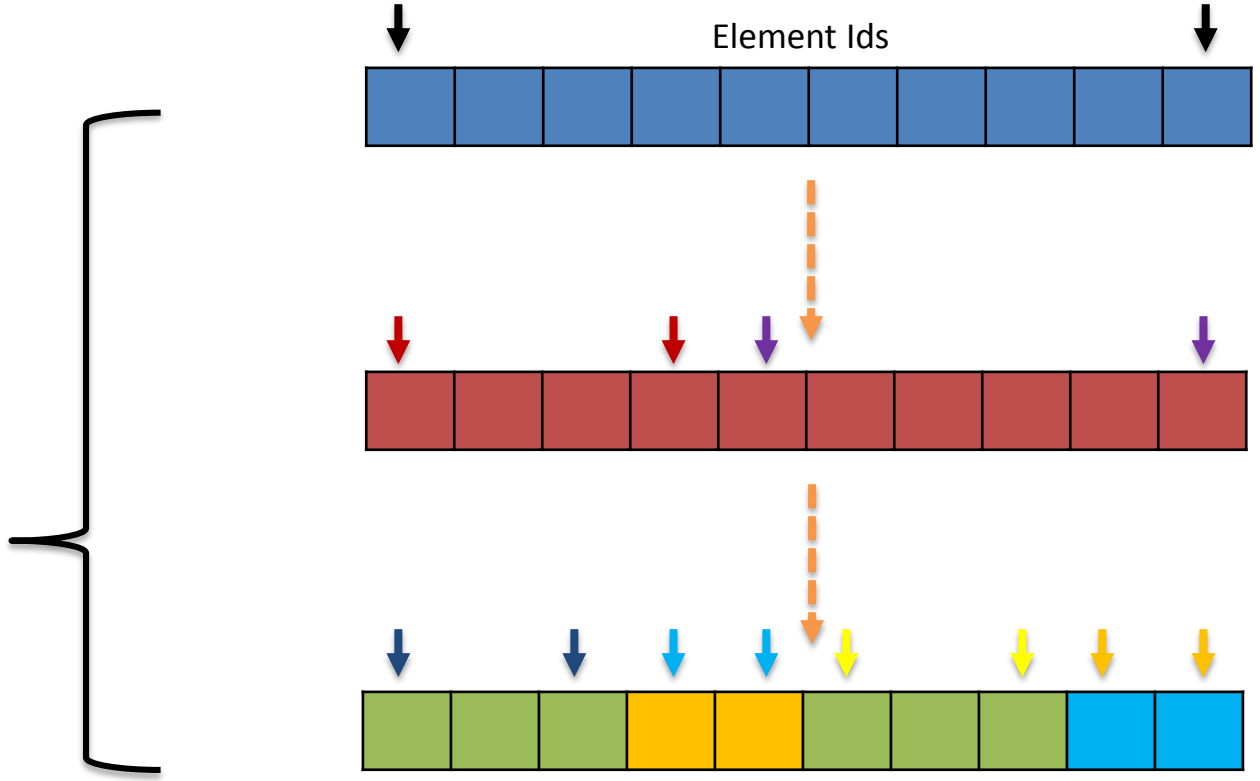
Parallel DACRT

- Pivots
- Node
- Level
- Implicit Hierarchy



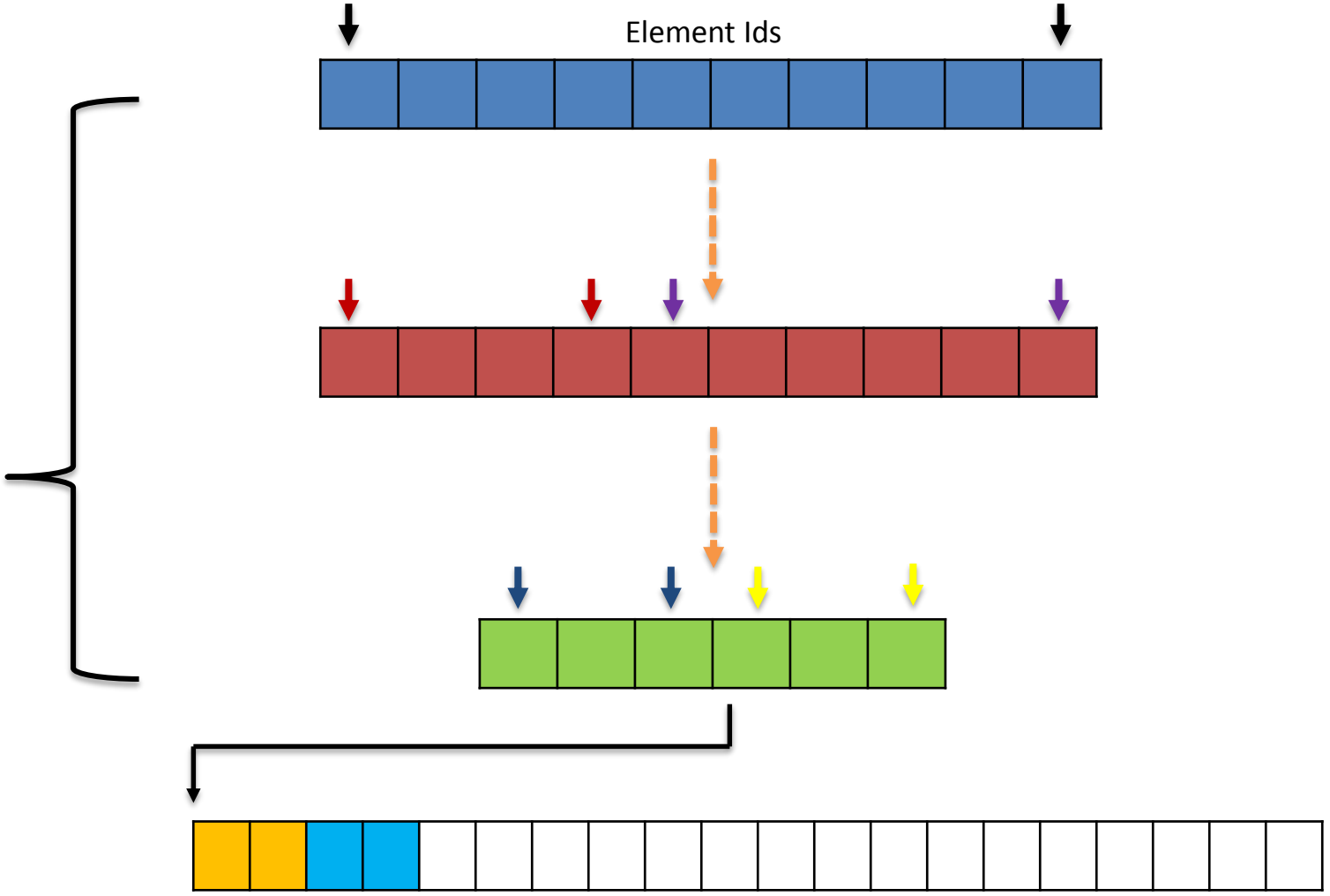
Parallel DACRT

- Pivots
- Node
- Level
- Implicit Hierarchy
- Terminal Nodes

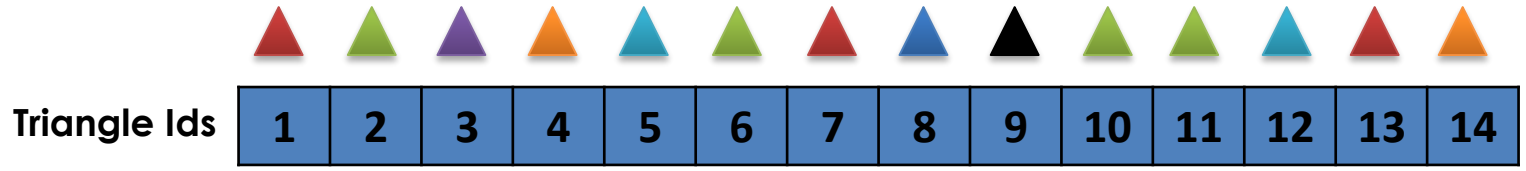


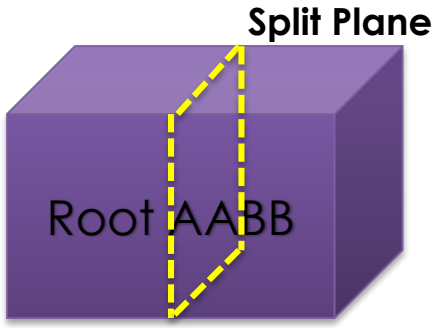
Parallel DACRT

- Pivots
- Node
- Level
- Implicit Hierarchy
- Terminal Nodes
- Terminal Node Bufer

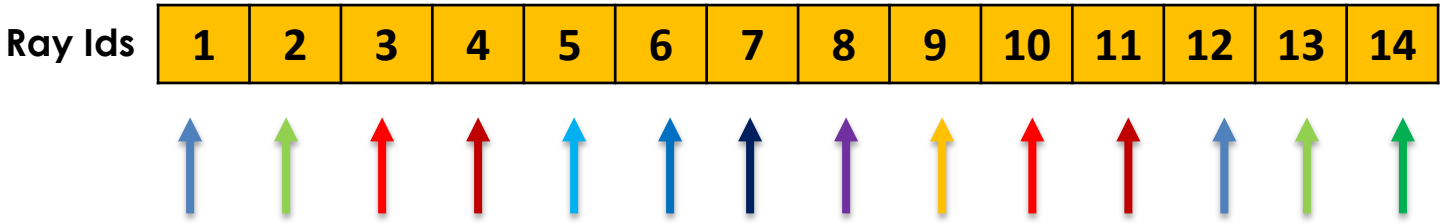


PDACRT - Visualized

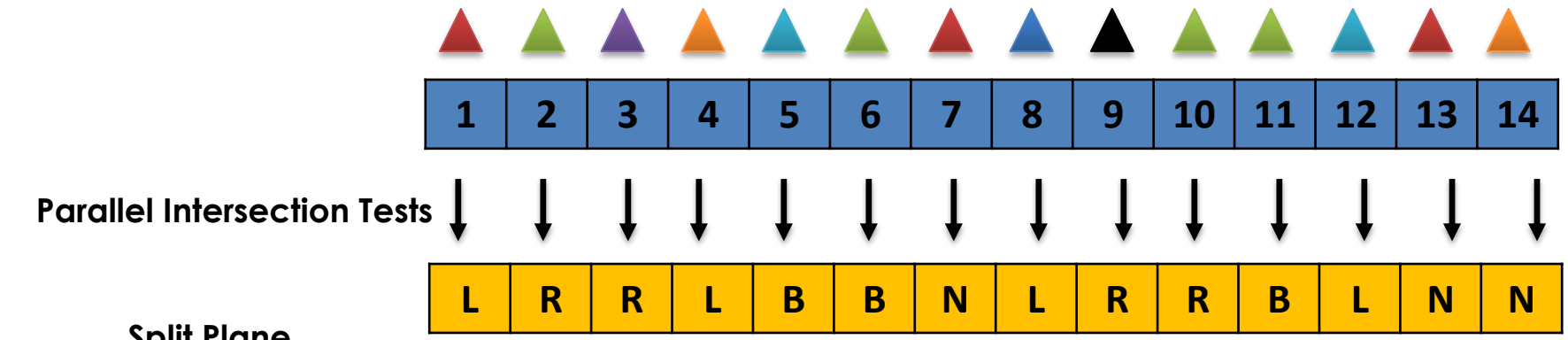




Level 0



PDACRT Filtering

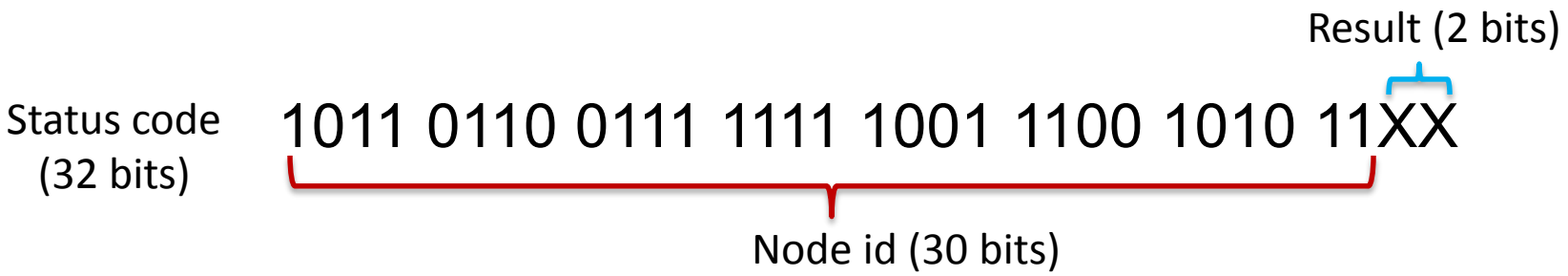


- Pivot property requires elements of same node to be in contiguous range.
- Achieved by a **segmented sort-by-key** operation.
  - Segment – node
  - Key – intersection value (**L**eft,**R**ight,**B**oth,**N**one)
  - Value – element index
- Rays processed in same way.

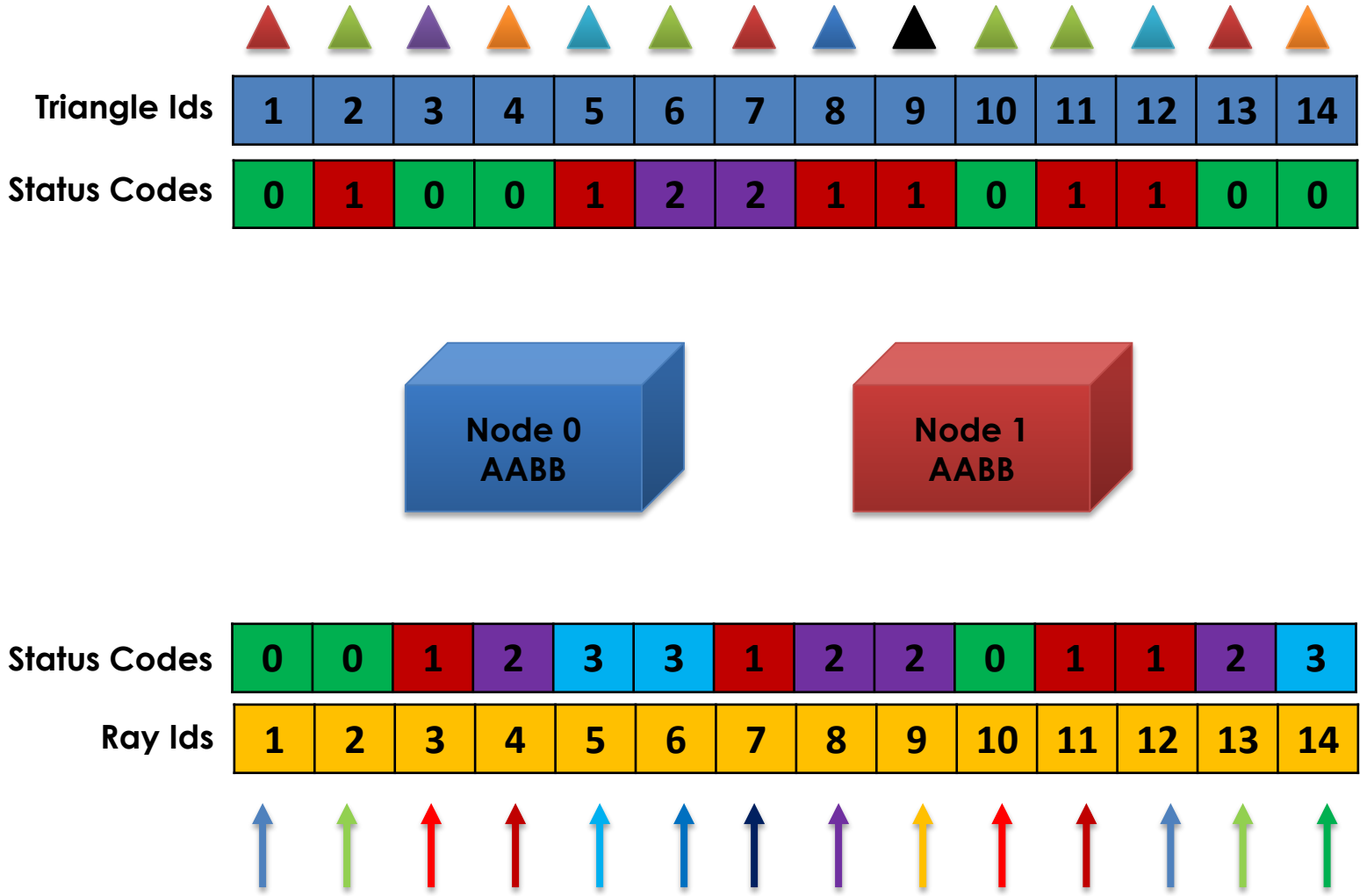
PDACRT – Status Codes

- 4 possible values for intersection result.
- **Status code** computed from **node id** and **intersection test result**.
- Pack both in a 32 bit unsigned integer
- Sorting status code ensures proper arrangement of elements intersecting both child nodes.

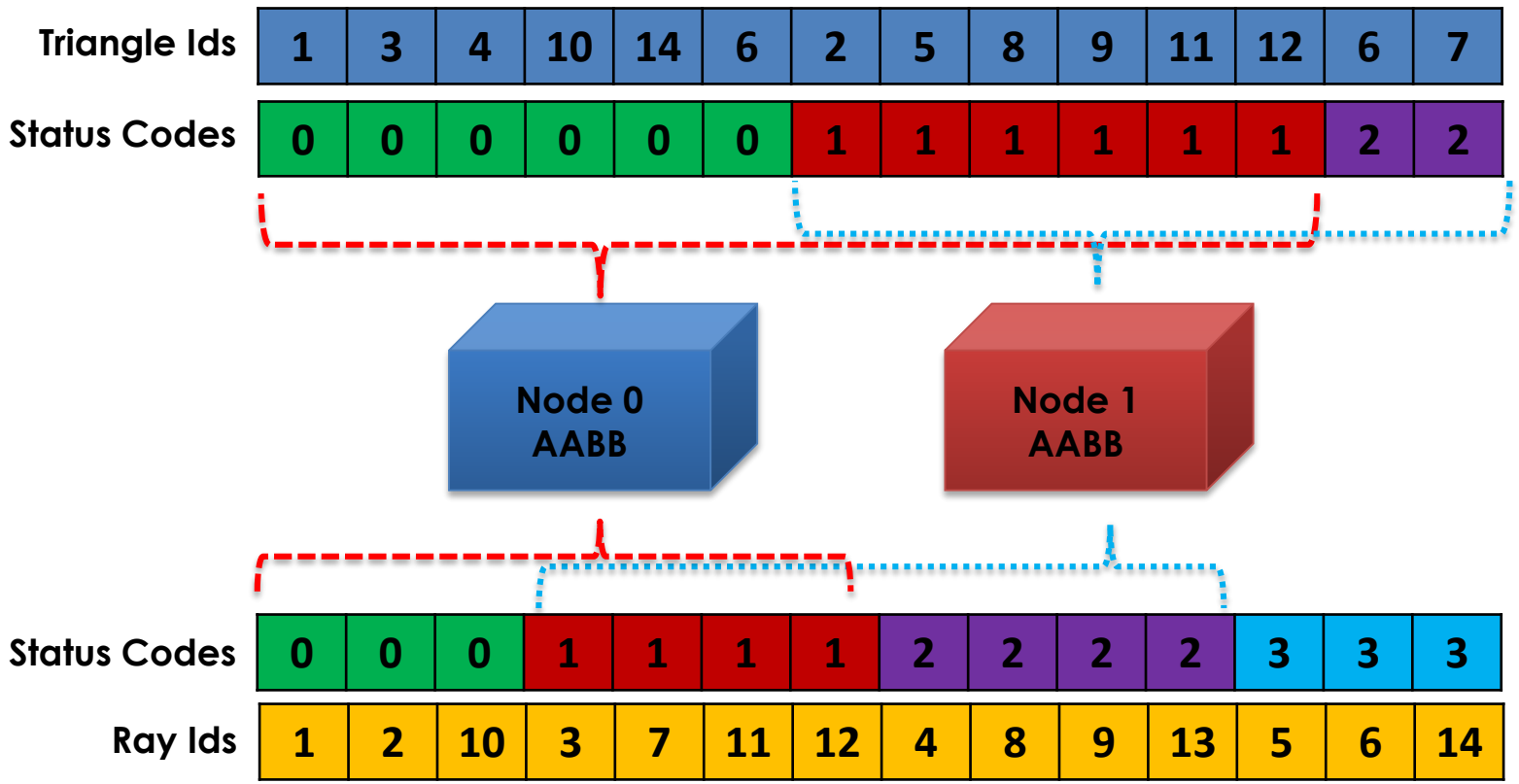
Type	Code
L	00
B	01
R	10
N	11



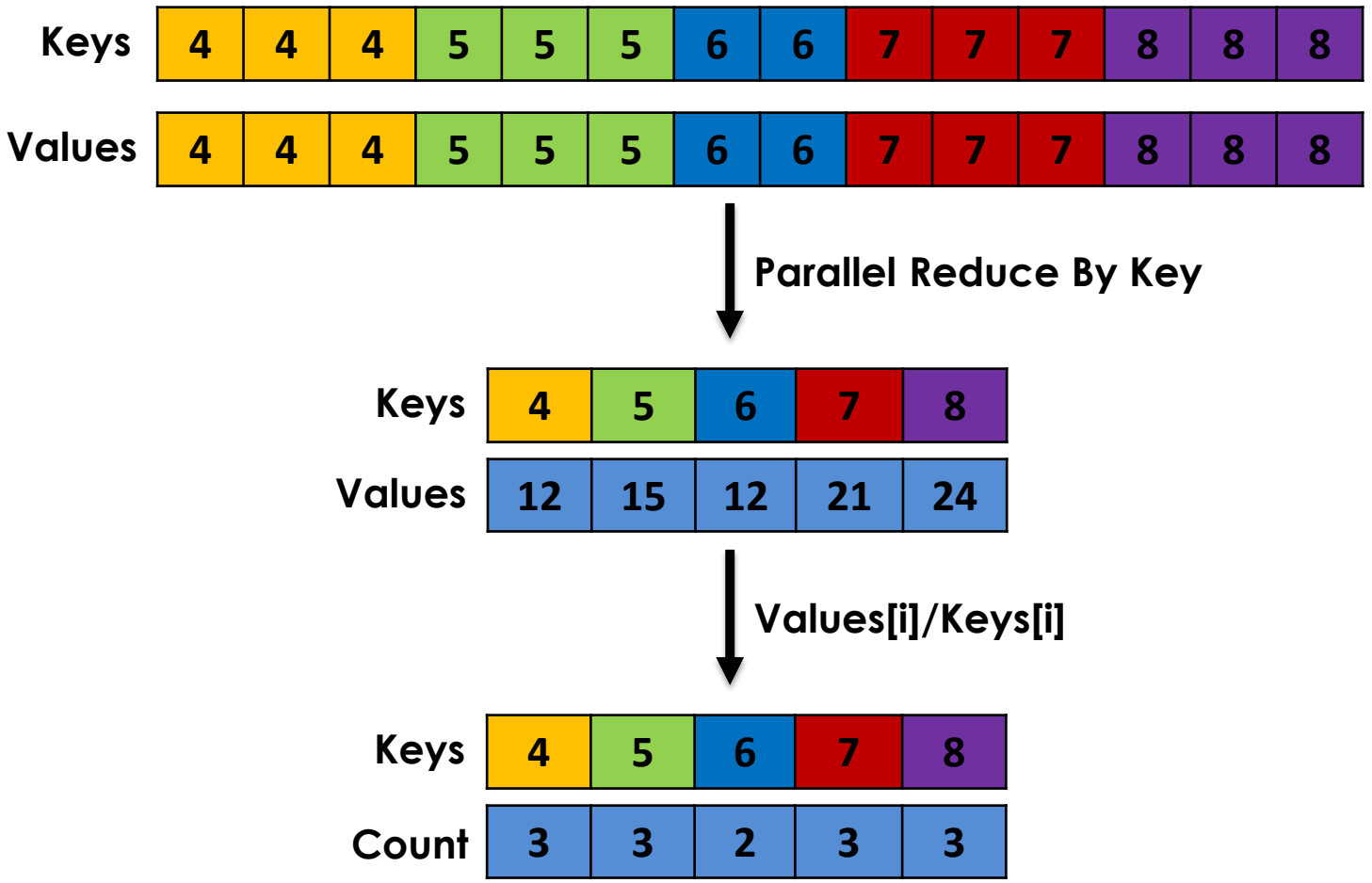
PDACRT - Node Computation



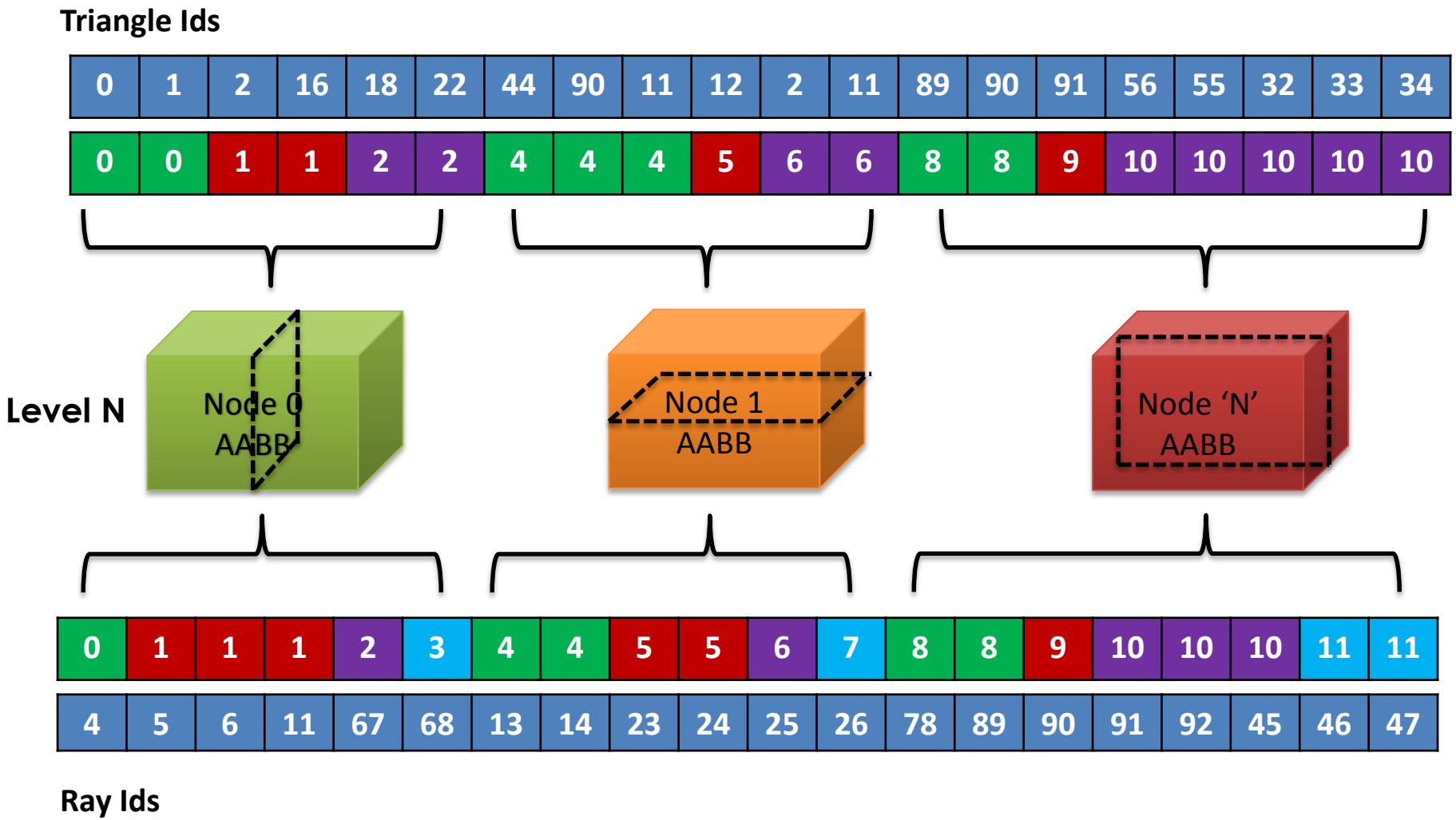
PDACRT- Node Computation



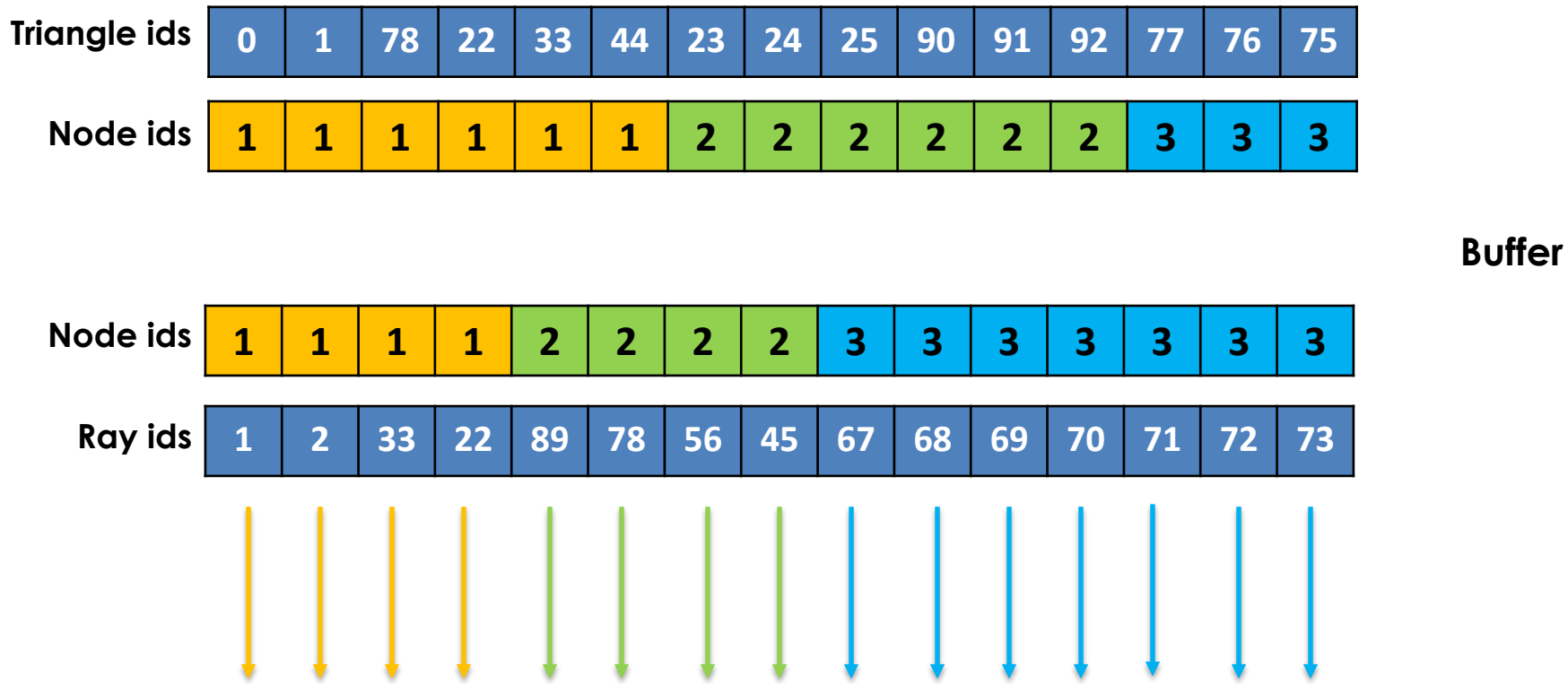
PDACRT – Parallel Size Computation



PDACRT – State at level ‘N’



PDACRT – Naïve Intersections



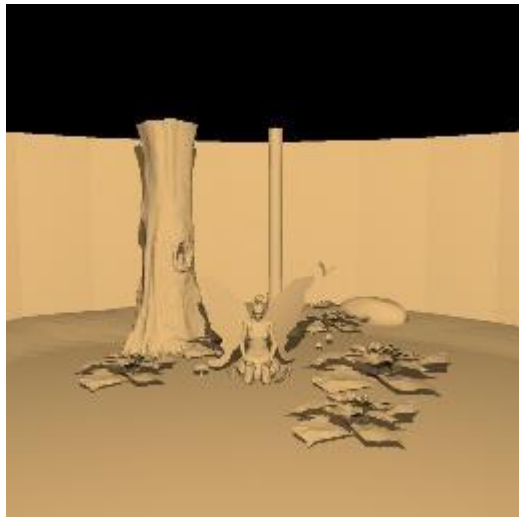
PDACRT – Algorithm

PDACRT (Set of Rays **R**, Set of Triangles **T**, Scene AABB **B**)

while(true)

- Split current level AABBs *in parallel*
- Perform ray & triangle filtering in *parallel*
- Compute child nodes and terminal nodes in *parallel*.
- Fill and process terminal node buffer in *parallel*.
- Compute next level node data in *parallel*.
- Repeat till no nodes present

Results



- Tests run on a machine with Intel core i7 CPU and Nvidia GTX 580 GPU.
- All scenes rendered at 1024x1024 resolution.

PDACRT – Primary Rays

Model	#Tris	CPU DACRT (ms)	PDACRT (ms)	Speedup
Bunny	69K	105	87	1.2x
Conference	282K	99	148	0.66x
Angel	474K	n/a	143	n/a
Dragon	871K	n/a	155	n/a
Buddha	1M	238	165	1.44x
Turbine	1.7M	285	223	1.27x

- PDACRT is faster than CPU DACRT for all the scenes except the conference scene.

PDACRT – Primary Rays

Model	#Tris	CPU DACRT (ms)	PDACRT (ms)	Speedup
Bunny	69K	105	87	1.2x
Conference	282K	99	148	0.66x
Angel	474K	n/a	143	n/a
Dragon	871K	n/a	155	n/a
Buddha	1M	238	165	1.44x
Turbine	1.7M	285	223	1.27x

- PDACRT is faster than CPU DACRT for all the scenes except the conference scene.
- Internal scenes have rays reach deep levels of hierarchy before being filtered.
- Early ray termination not possible due to breadth first processing of rays.

PDACRT – Secondary Rays

Scene	#Tris	Shadow Rays (ms)	Specular Rays (ms)	AO Rays (ms)
Bunny	69K	67	96	149
Conference	282K	197	222	240
Angel	474K	102	163	182
Dragon	871K	128	177	192
Buddha	1M	150	190	204
Turbine	1.7M	213	252	287

- Shadow rays were generated with one point light source.
- Shadow rays perform generally better due to some degree of coherence among them.
- 8 AO rays generated per primary ray intersection.
- PDACRT performance doesn't fall off rapidly with increase in ray count.

PDACRT – Memory Requirements

Scene	#Tris	GPU SAH k-D Tree (MB)	PDACRT (MB)
Bunny	69K	33.96	47.66
Fairy	174K	80.33	82.25
Exploding	252K	86.58	82.68
Conference	331K	159.98	85.82
Angel	474K	218.26	82
Dragon	871K	417.33	96.87
Buddha	1M	512.65	107.89

- PDACRT values include memory required for buffer, ray and triangle data also.
- GPU SAH k-D Tree data includes memory only for triangles.
- Considerable ray id duplication for internal scenes.

GPU Light Transport

Image Formation Model

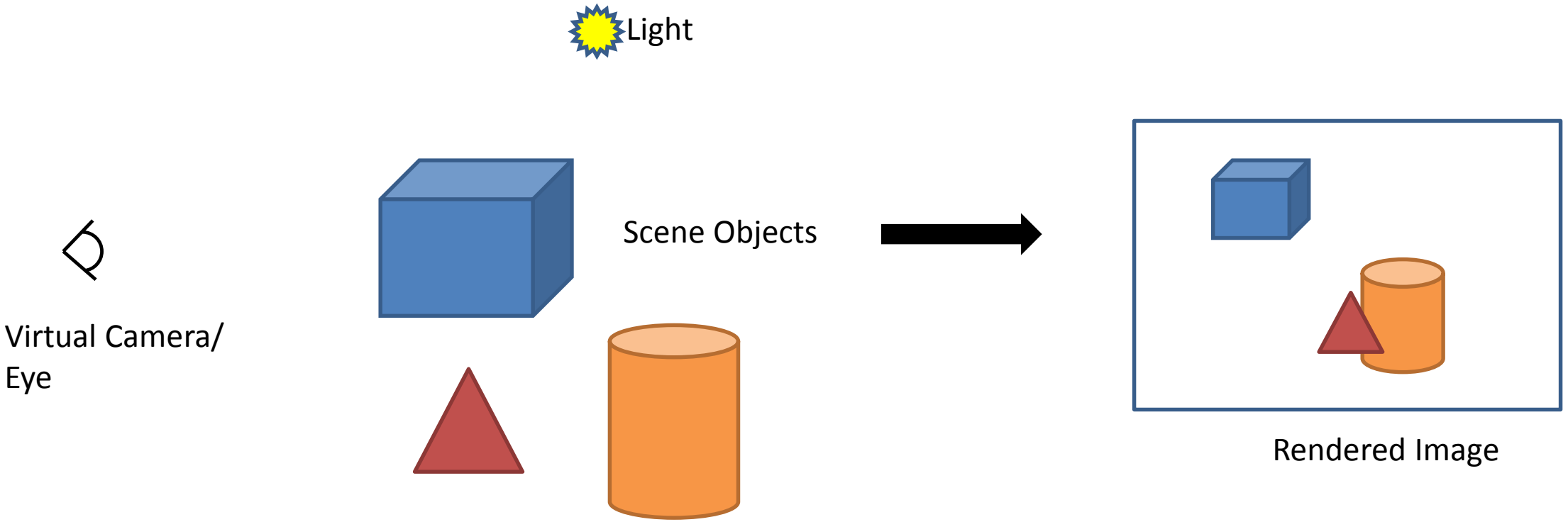
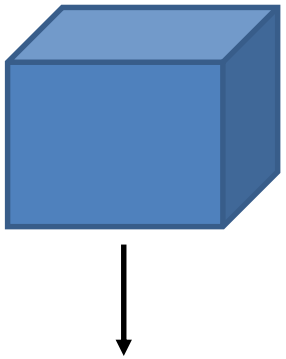


Image Formation Model - Data

Virtual Camera/Eye

1. Camera Position
2. Resolution
3. Lens Configuration
4. Aperture, etc.

Scene Objects



1. Geometry
2. Material type

Light

1. Light Position
2. Light Geometry
3. Emission Profile

Rendering Equation

$$L_o(x, \omega_o) = Le(x, wo) + \int_{\Omega} L_i(x, \omega_i) f_s(x, \omega_i, \omega_o)\, d\sigma^{\perp}(\omega_i)$$

Outgoing radiance

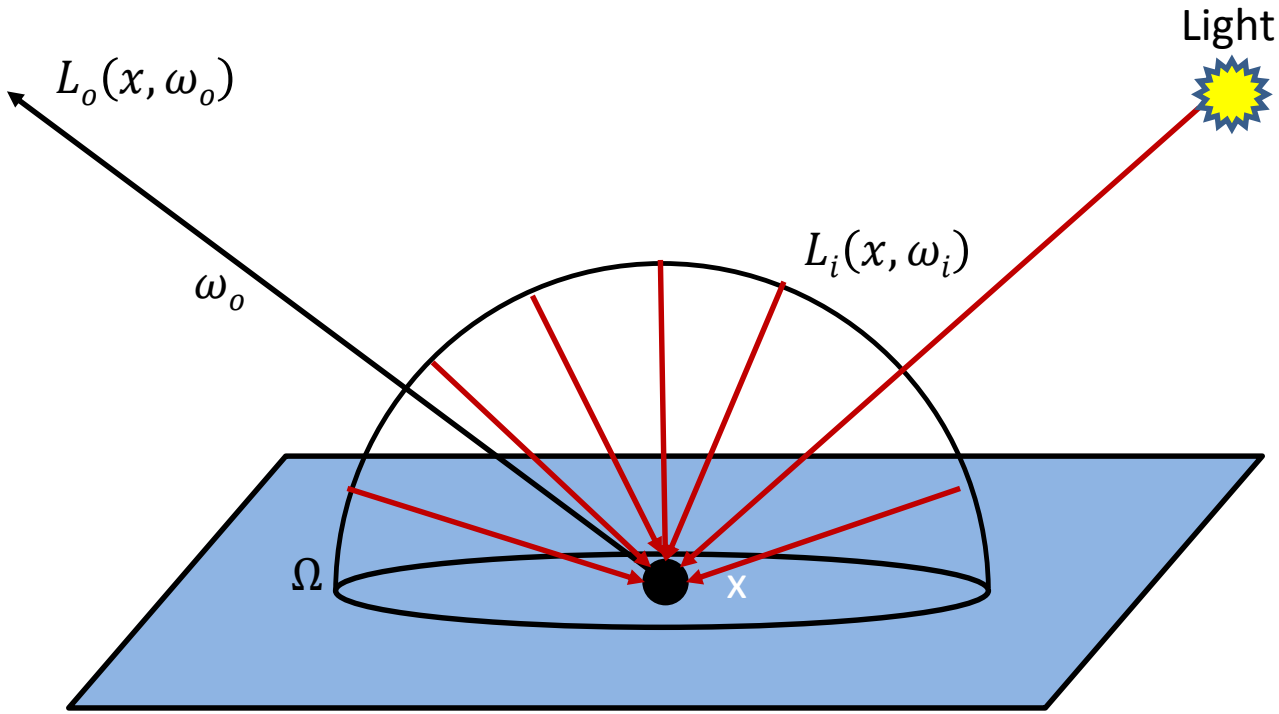
Emitted radiance

Incoming radiance

BSDF

Projected Solid Angle Measure

Rendering Equation - Visualized



Outgoing Radiance (L_o)

=

Sum of scaled incident radiance (L_i)

+

Self Emitted Radiance (L_e)

Monte Carlo Integration

- Evaluate an integral f(x) in domain Ω.
- Standard quadrature rules work good for 1 dimension.
- As dimension increases, convergence rate is very bad.
- Monte Carlo methods are oblivious to dimensions. Just take lot of samples.
- Convergence rate is O($\sqrt{N}$)

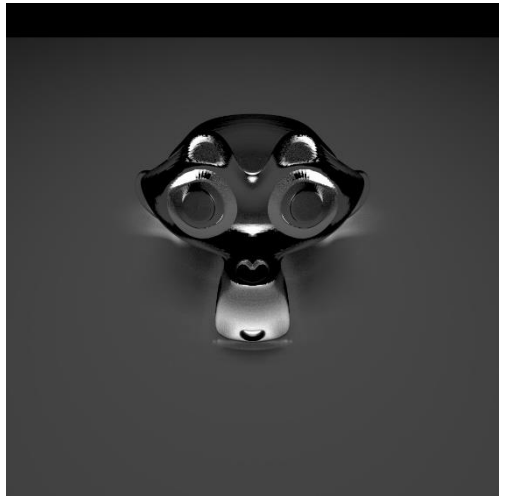
$$I = \frac{1}{N}\sum_{i=1}^{N}\frac{f(Xi)}{p(Xi)}$$

$$E(I) = \int f(x)p(x)dx \quad = \quad \int_{\Omega} f(x)d\mu(x)$$

Monte Carlo Estimator

BSDF Models

Diffuse



Glass

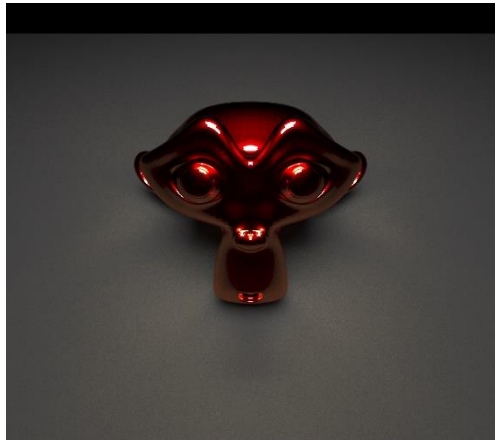
Mirror

Microfacet

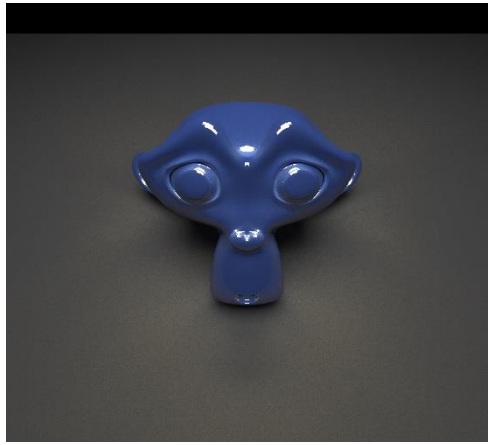
- BSDF models indicate how for a given point on the surface, light in an incoming direction is scattered in a particular outgoing direction.
- BSDF provides the scaling factor in the rendering equation.

Materials

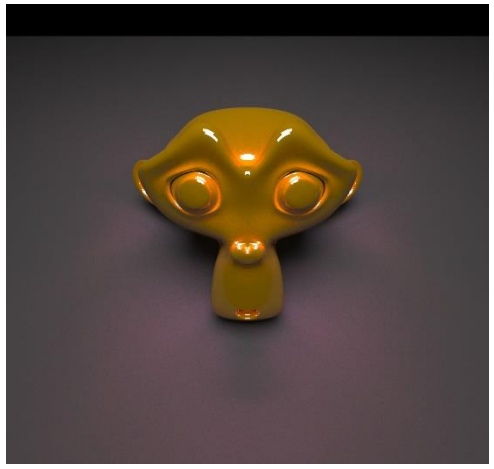
- Materials determine the appearance of objects.
- Complex materials can be composed of arbitrary layers.
- Each layer can indicate a particular BSDF.
- Monte Carlo methods samples each layer separately to capture effects.
- Rich effects are produced by complex materials.



2 Layers – Shiny Metal



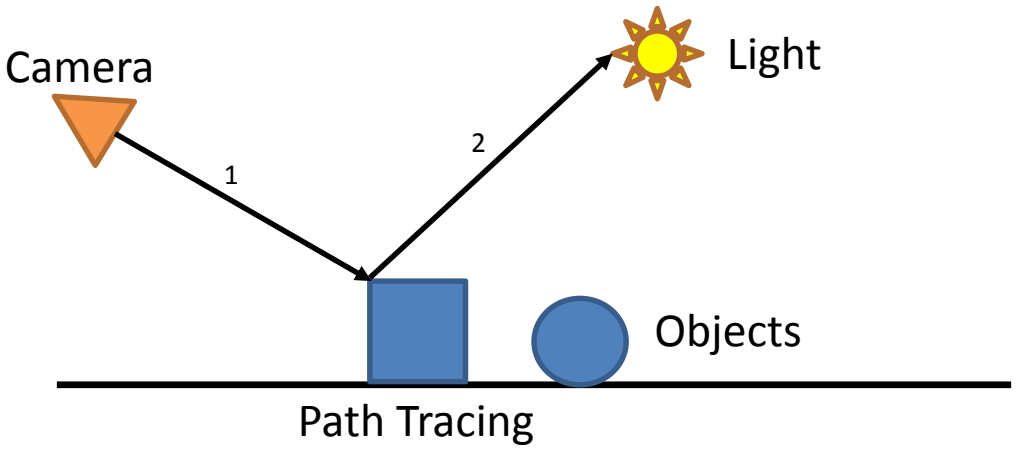
2 Layers – Glossy Paint

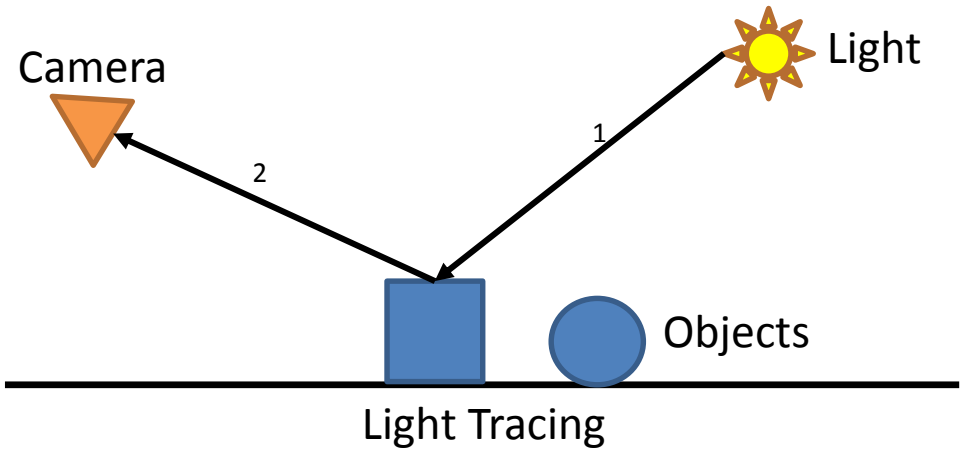


3 Layers

Unidirectional Light Transport

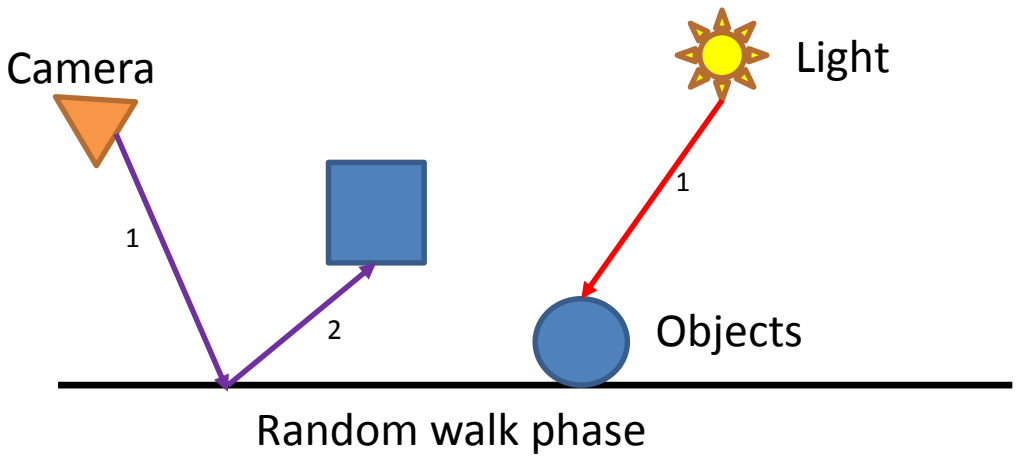
- Start from either the camera or the light
- Stochastically terminate path.
- Camera -> Path Tracing
  - Estimate radiance at each scene point
- Light -> Light Tracing
  - Estimate importance at each scene point.



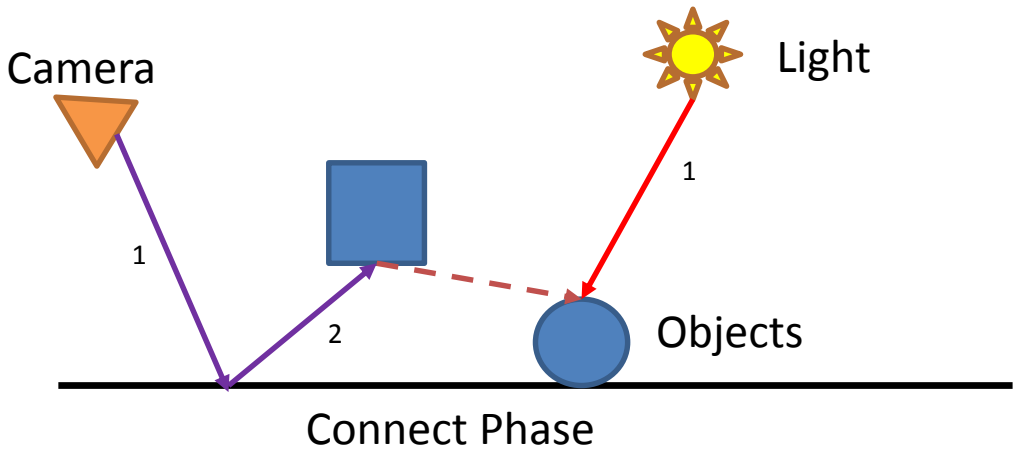


Bidirectional Path Tracing

- Start random walk from both light and camera.
- Proceed till termination (some criteria).
- Connect both paths to form a valid light path.
- Handles complicated lighting scenarios effectively.

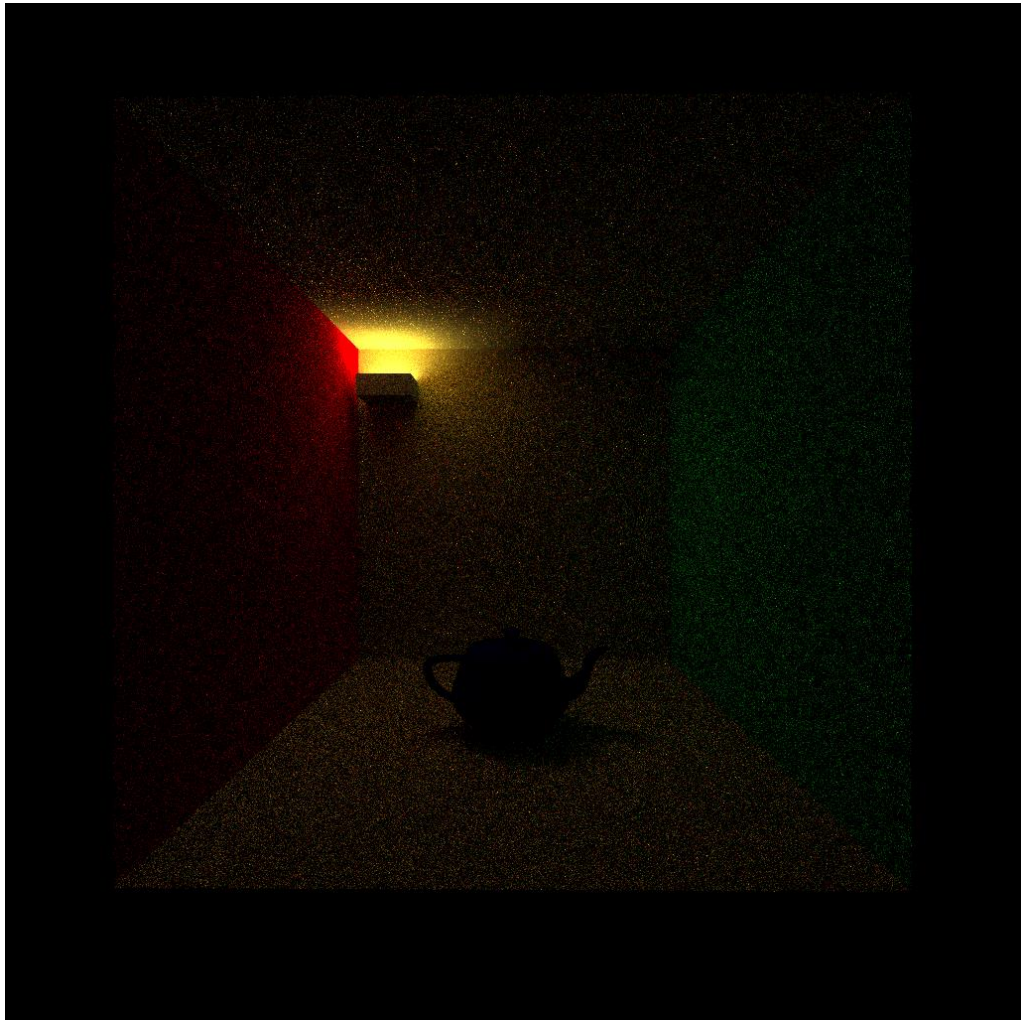


Random walk phase

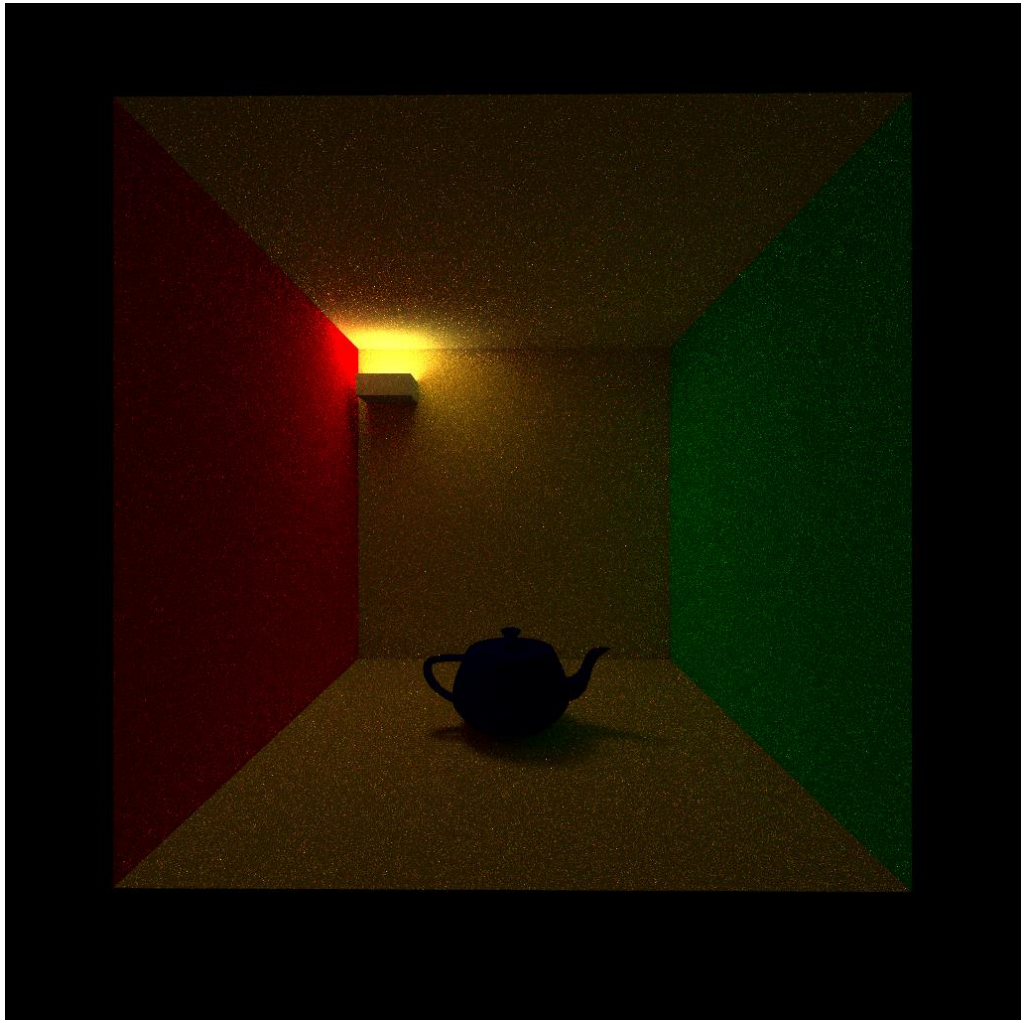


Connect Phase

PT vs BDPT



Path Tracing



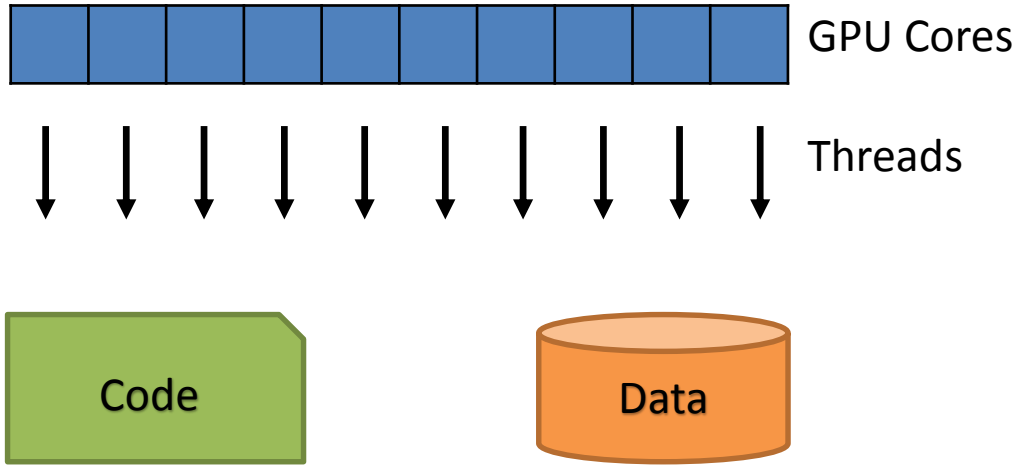
Bidirectional Path Tracing

GPU Performance Basics

1. GPU SIMT execution engine.
2. Each SMM consists of compute cores
3. Grids -> Blocks -> Warps -> Threads
4. Performance = Warp Execution Coherency

+

Warp Coalasced Data Access

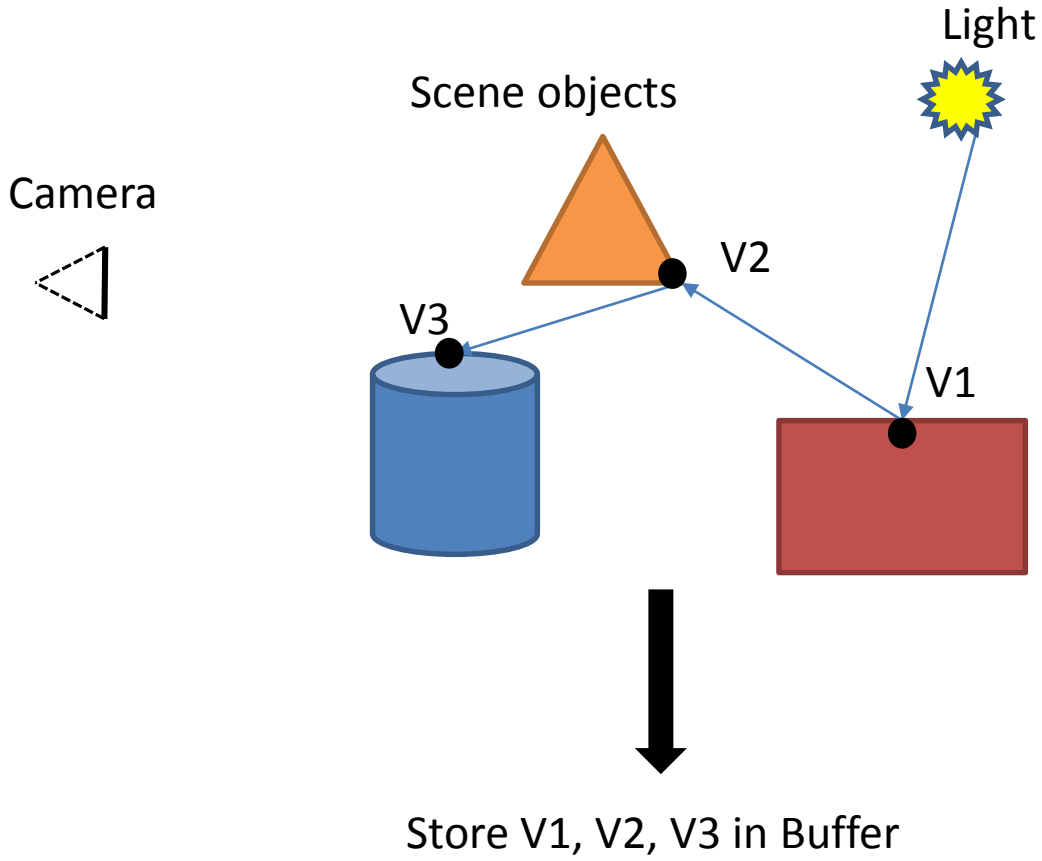


GPU Light Transport Challenges

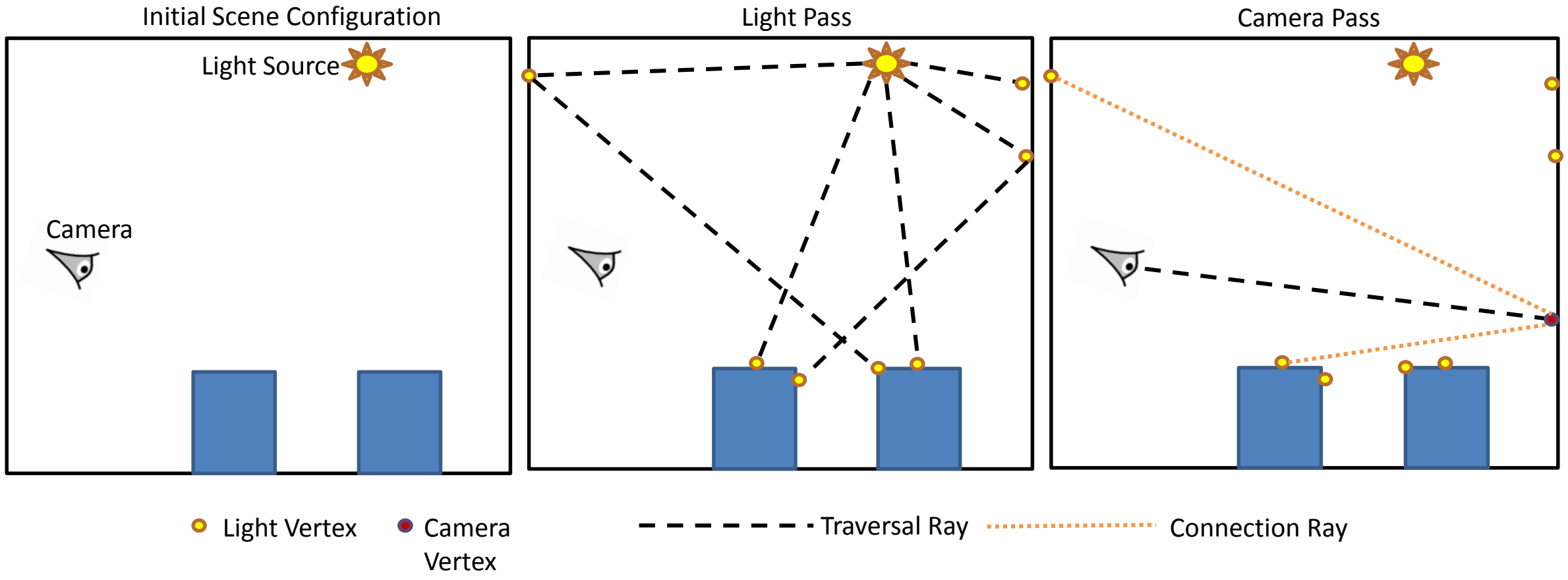
- Preallocate memory for all subpaths in case of BDPT algorithms.
- Keep GPU occupancy high (stochastic termination of samples)
- Code execution divergence (different materials – different code paths)
- Large kernels or small kernels?
  - Large kernels – Register Pressure – Occupancy Low
  - Small kernels – Too many small kernels – kernel launch overhead

Solutions for Memory Preallocation

- A problem only in BDPT.
  - Entire light and camera subpaths required for final evaluation
- Employ a Light Vertex Cache (LVC) in first light pass
  - Stores only light subpath vertices
- Camera pass done next after LVC is full
  - Connect each camera vertex to some vertices in LVC
  - Compute valid paths and compute contribution
  - Camera vertices not required to be stored



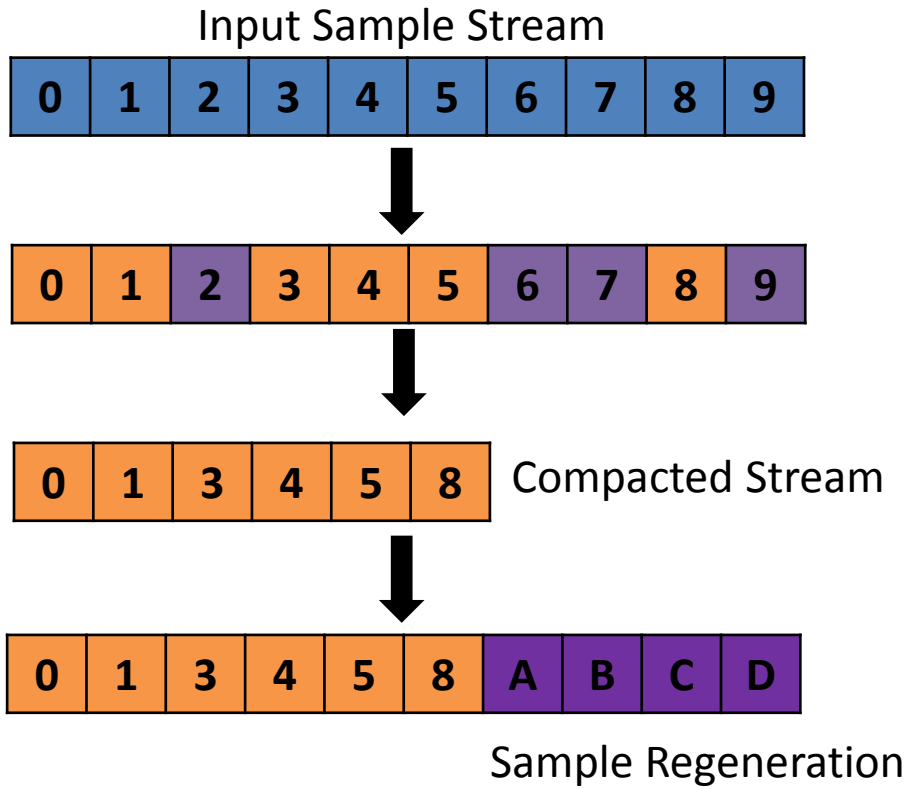
LVC – BDPT Visualized



Coherent and Importance Sampled LVC BDPT (CIS – LBDPT)

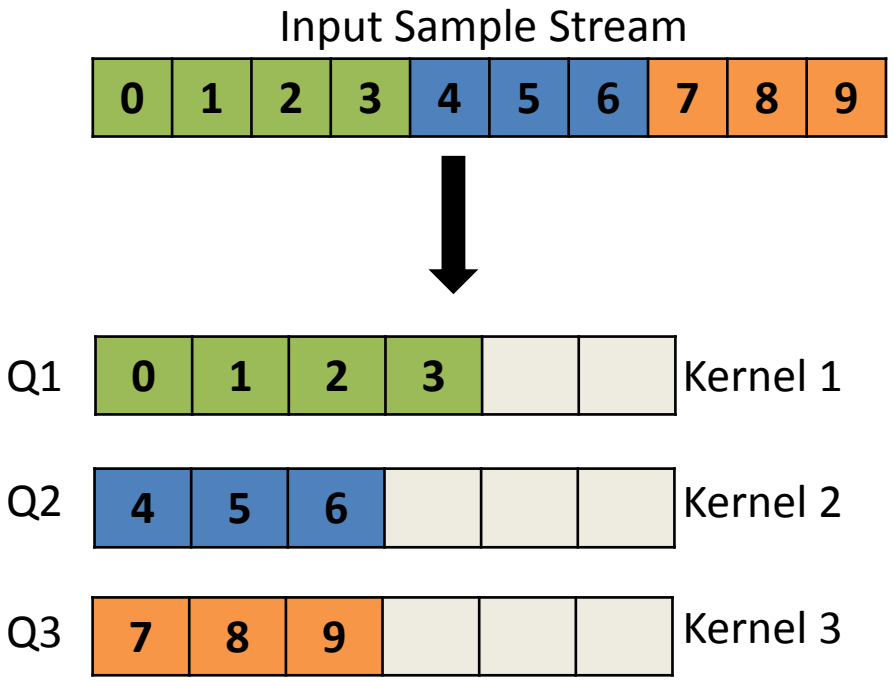
Solutions – For Occupancy

- Streaming Path Tracing
  - Large stream of samples to keep GPU busy.
- Stream compaction – Remove dead samples from stream
  - Don't let cores do wasteful work or be idle.
- Sample regeneration – Generate new samples in dead samples' place.
  - Keep all the cores of the GPU busy.



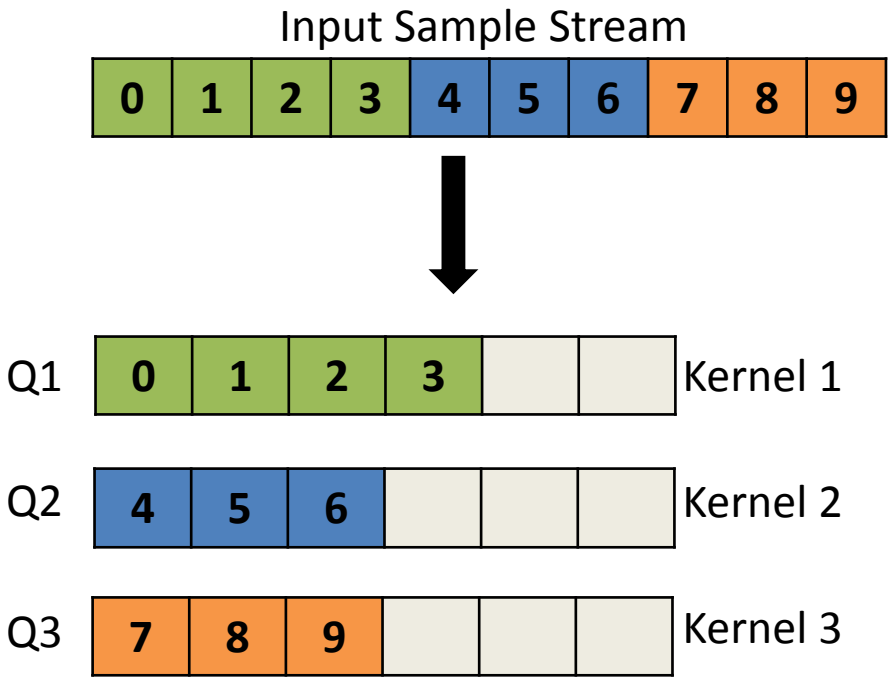
Solutions – For Coherence and Kernel size

- Employ queues for each different type of material handled.
  - Coherent work for each queue
- Employ different kernels for different materials.
  - Smaller kernels ensure less register pressure



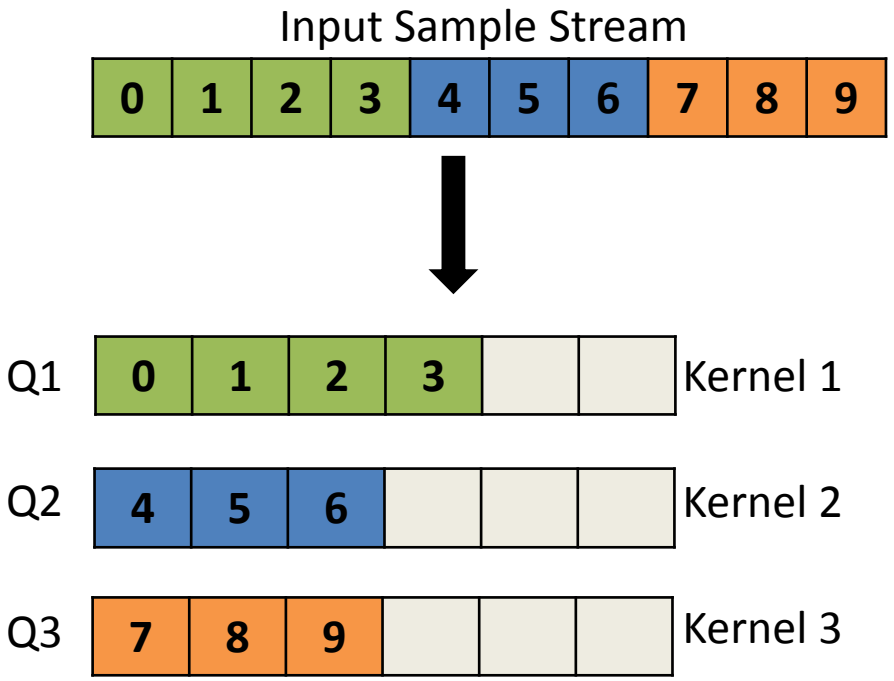
Solutions – For Coherence and Kernel size

- Cons
  - Separate queues for each coherent work load
  - Parallel Queue management very cumbersome
  - Preallocation of memory
  - Poor memory usage
  - Overflow

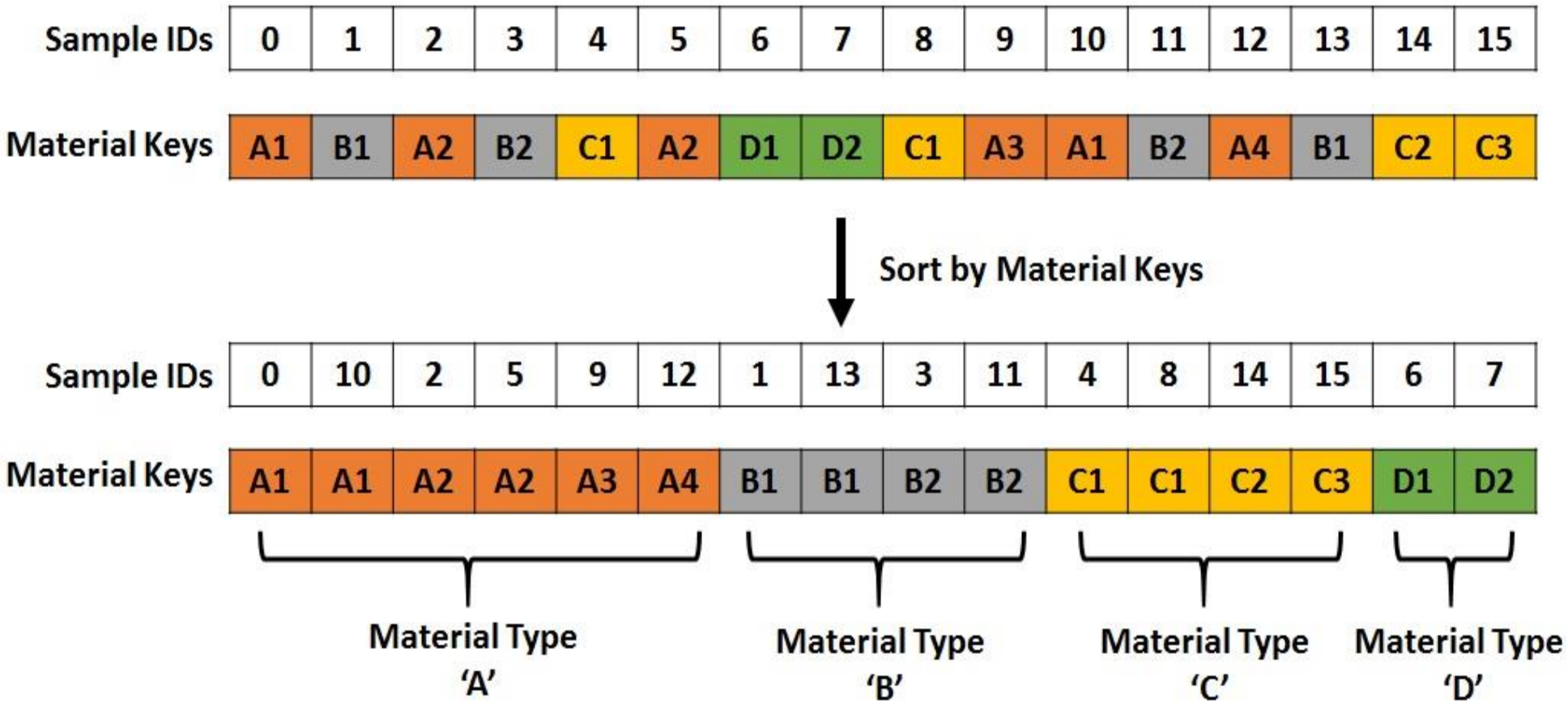


Solutions – For Coherence and Kernel size

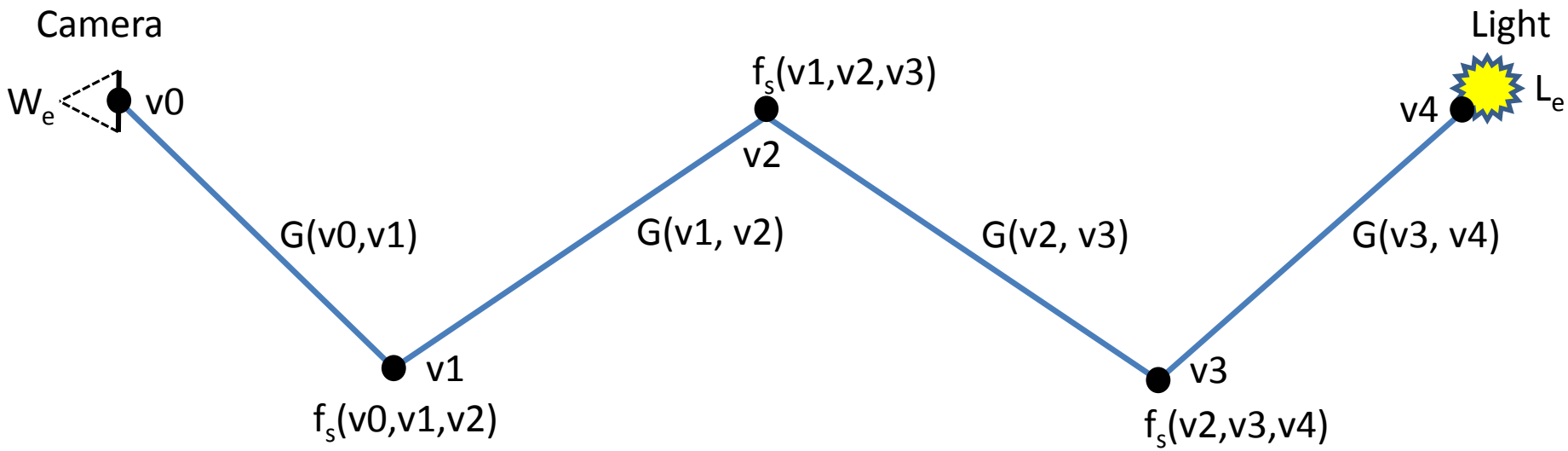
- Implicit queues to the rescue
  - Employ sorting within sample stream to form implicit queues
  - Each segment of coherent work load is the range of queue elements



Coherent Material Evaluation

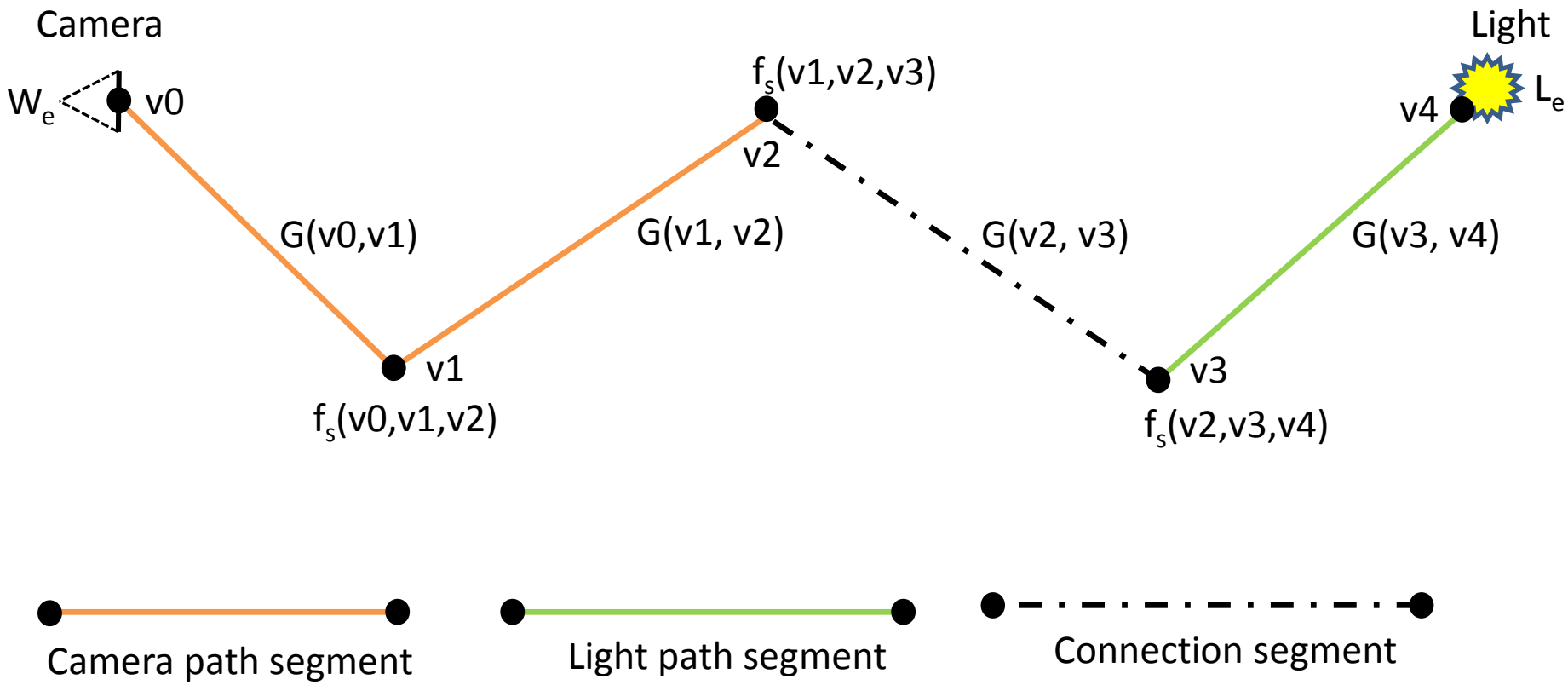


Contribution Function

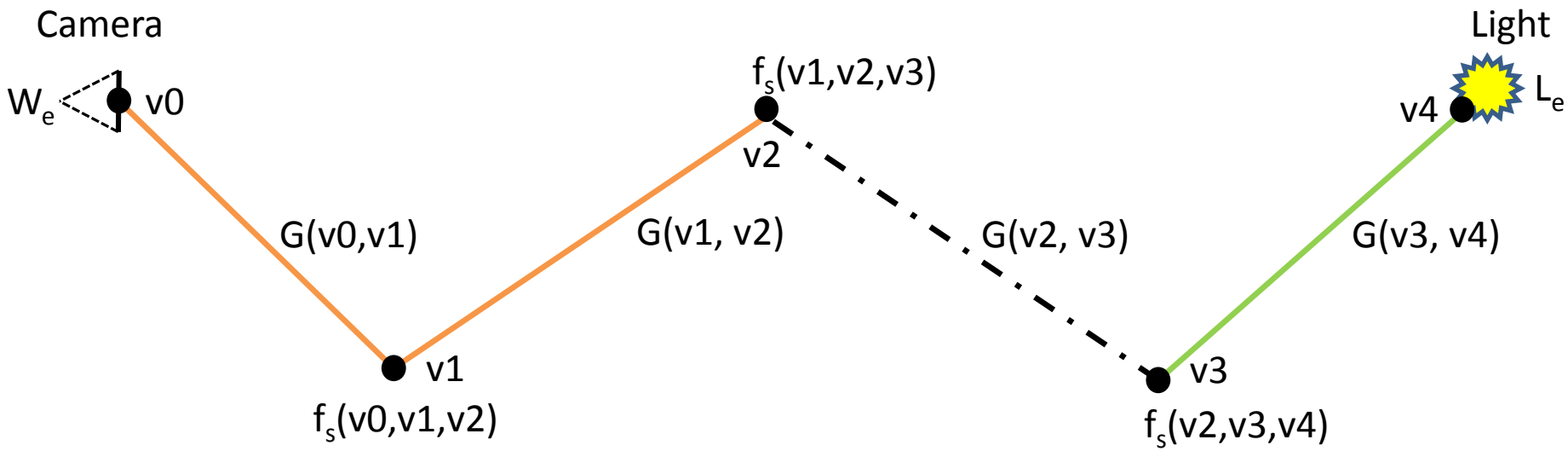


Final Contribution = W_e x G(v0,v1) x f_s(v0,v1,v2) x G(v1,v2) x f_s(v1,v2,v3) x G(v2,v3) x f_s(v2,v3,v4) x G(v3,v4) x L_e

BDPT Transport Path



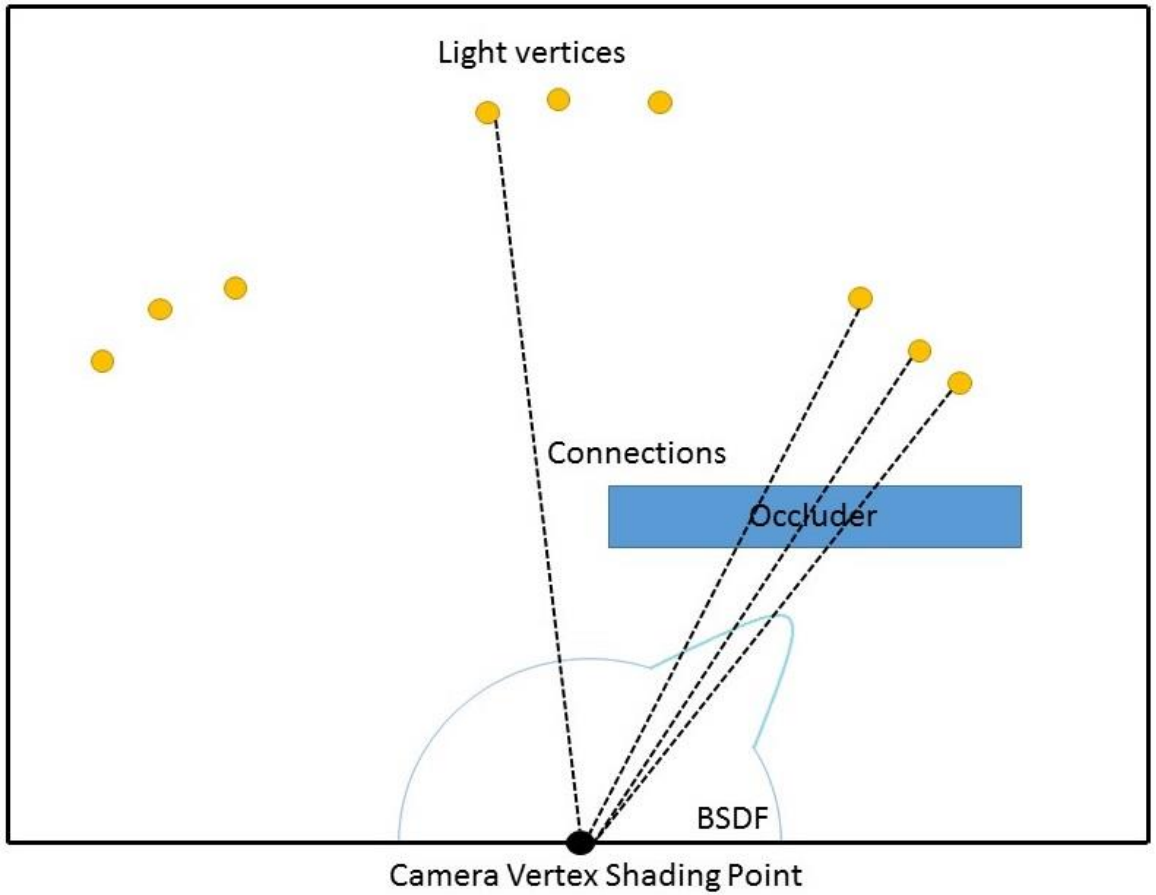
Connection Segment



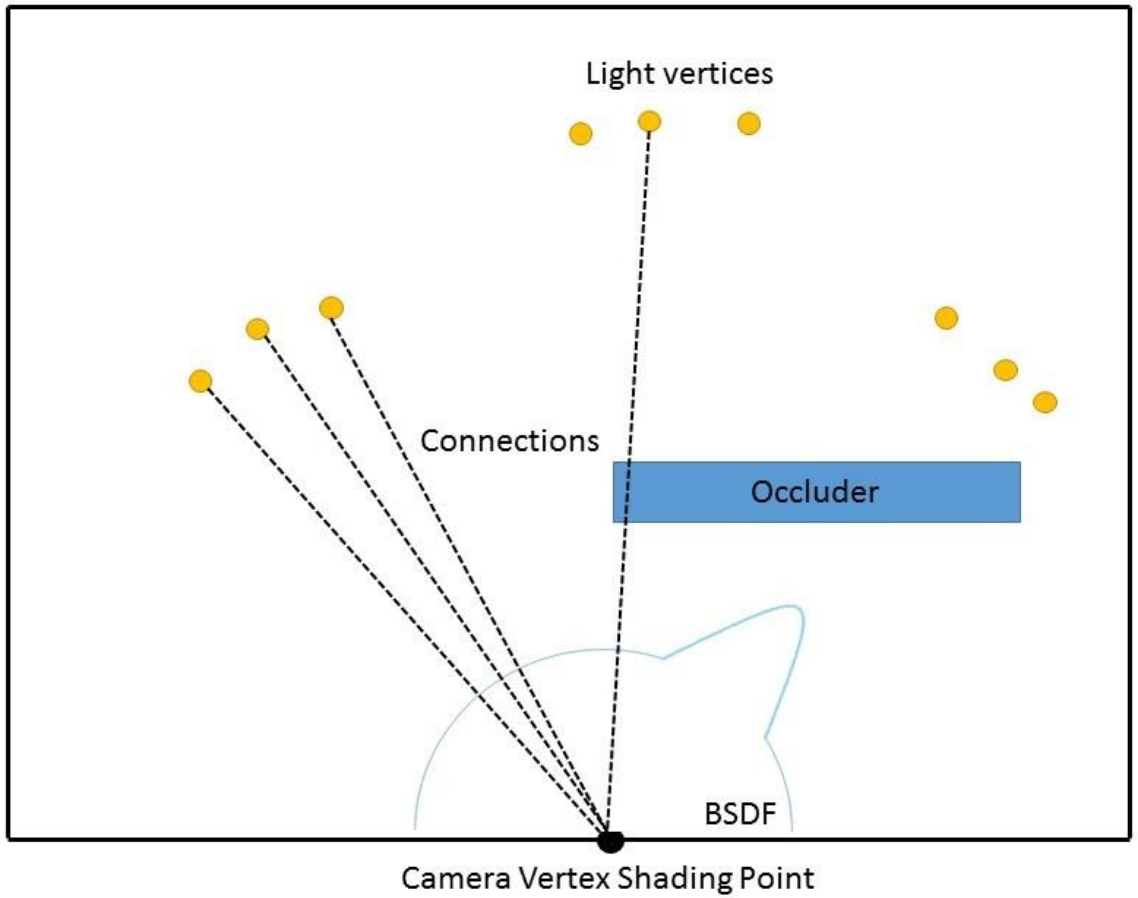
Connection Segment = $f_s(v1,v2,v3)$ x $G(v2,v3)$ x $f_s(v2,v3,v4)$ x $V(v2,v3)$

Poor Sampling of LVC vertices

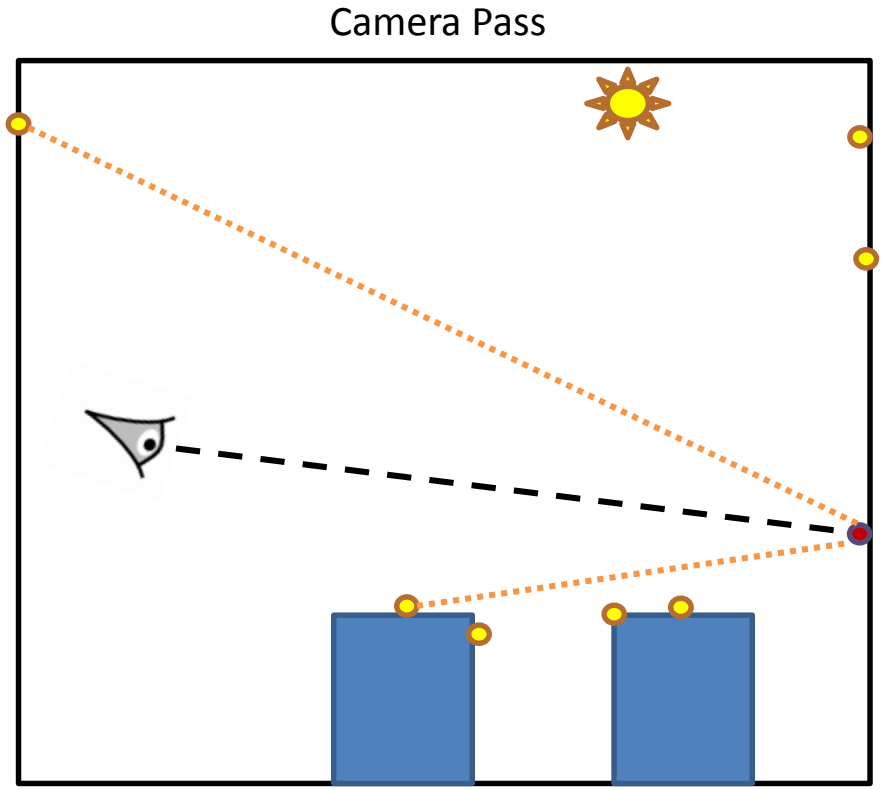
Poor choice of light vertices which are occluded



Poor choice of light vertices with low BSDF contribution



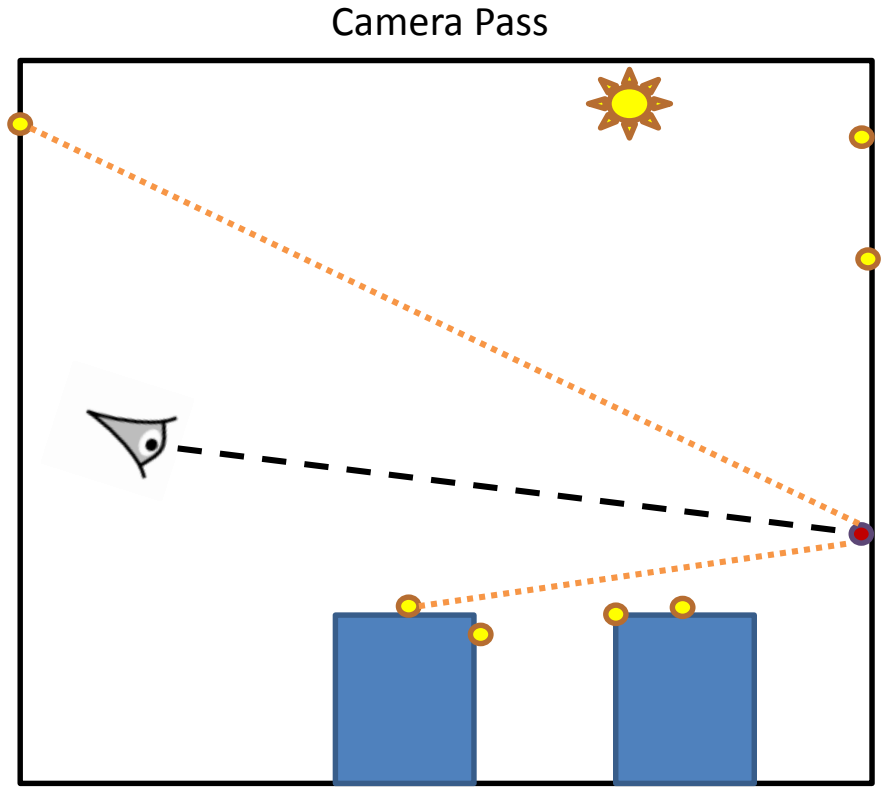
Key Idea – Importance Sampling LVC vertices



- LVC – BDPT uniformly samples light vertices
- Might not choose the best pair of vertices
- Importance sample light vertices based on contribution

CIS - LBDPT Importance Sampling

- For each camera vertex
  - Choose ‘N’ light vertices from LVC vertex at random.
  - Compute a distribution from them using the contribution term.
  - Importance sample ‘M’ vertices from this distribution.



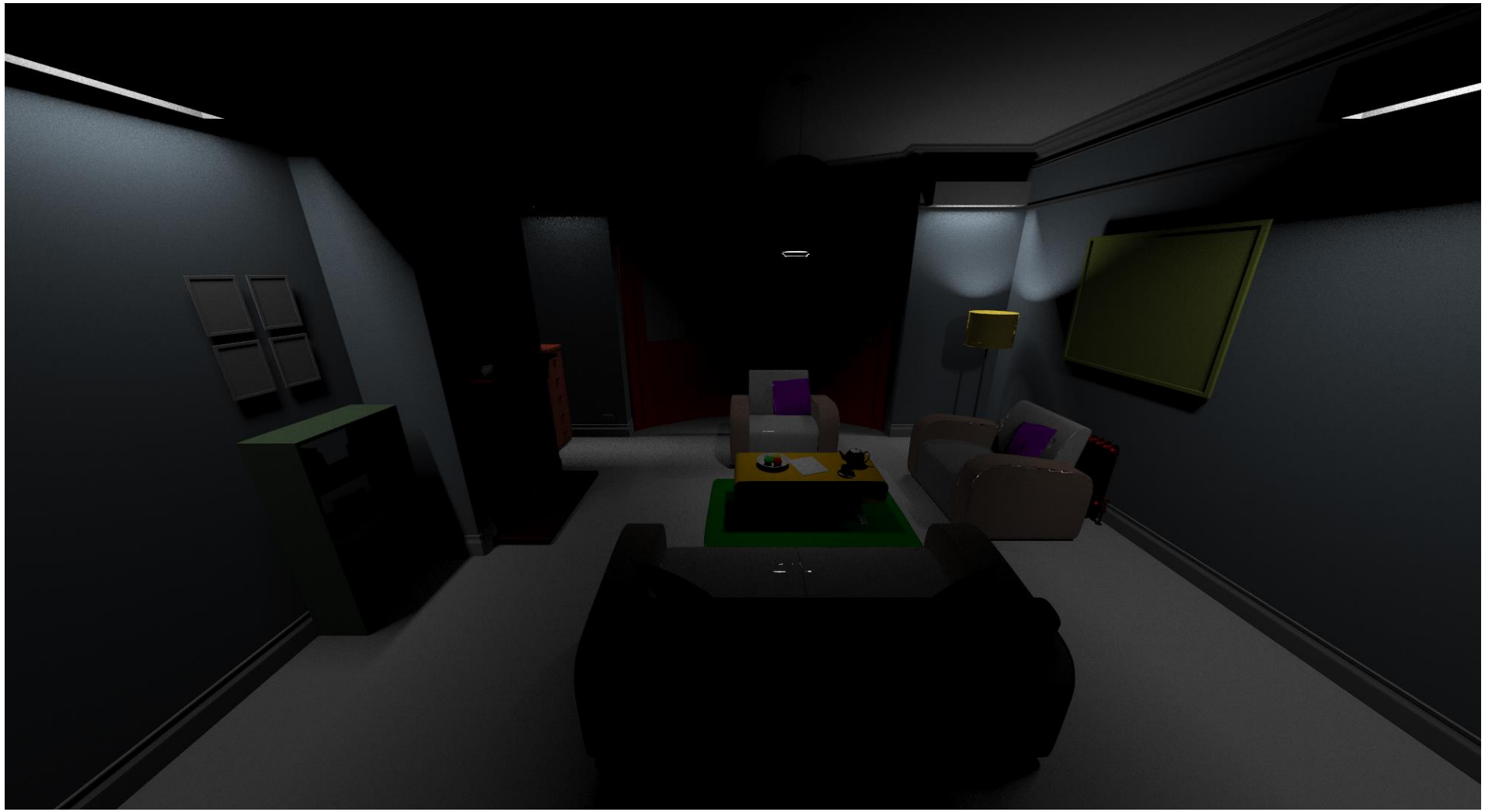
Results

Scene 1 : Bedroom – Direct Lighting only

Results

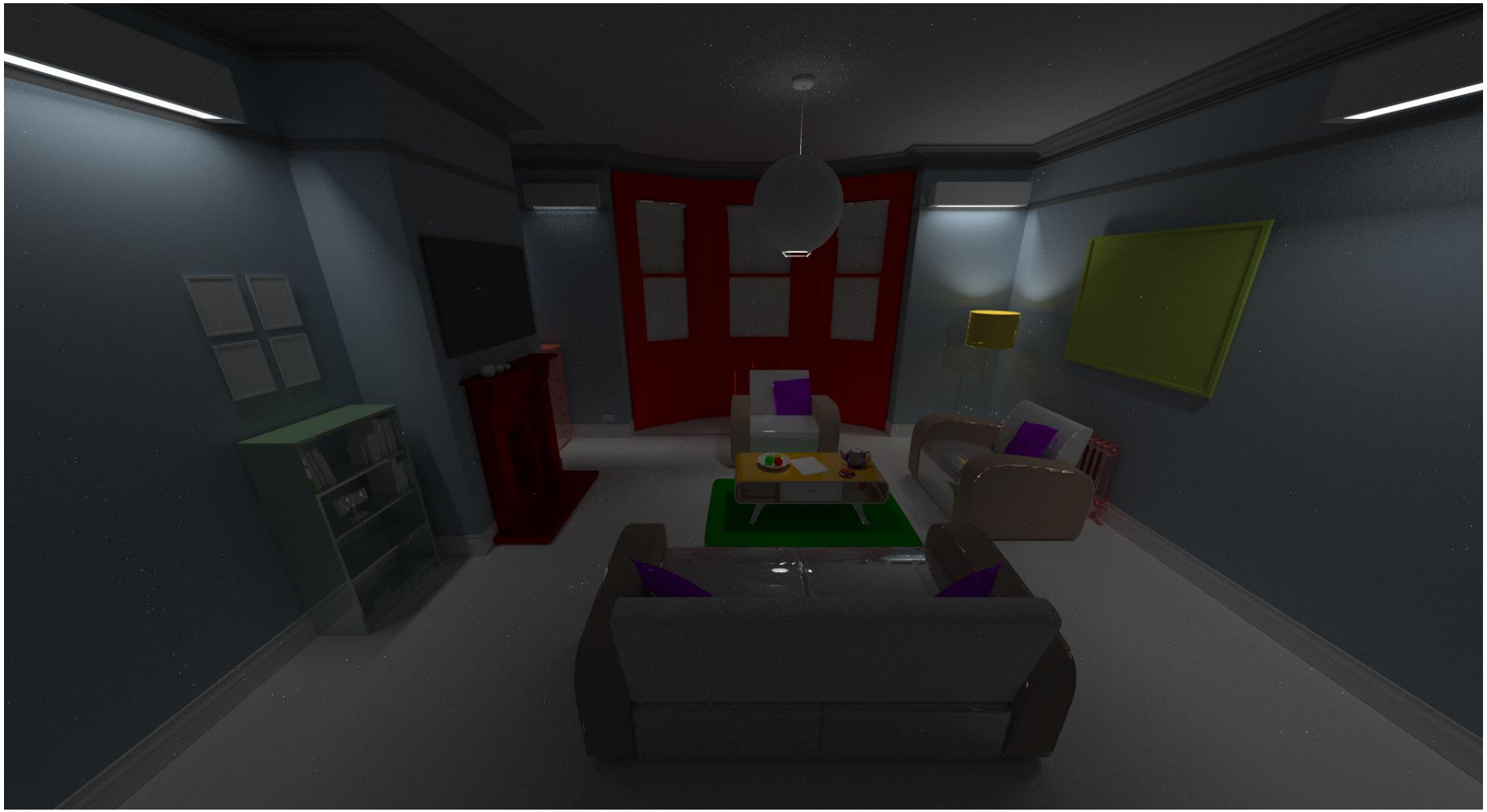
Scene 1 : Bedroom – Fully lit

Results



Scene 2 : White room – Direct Lighting only

Results



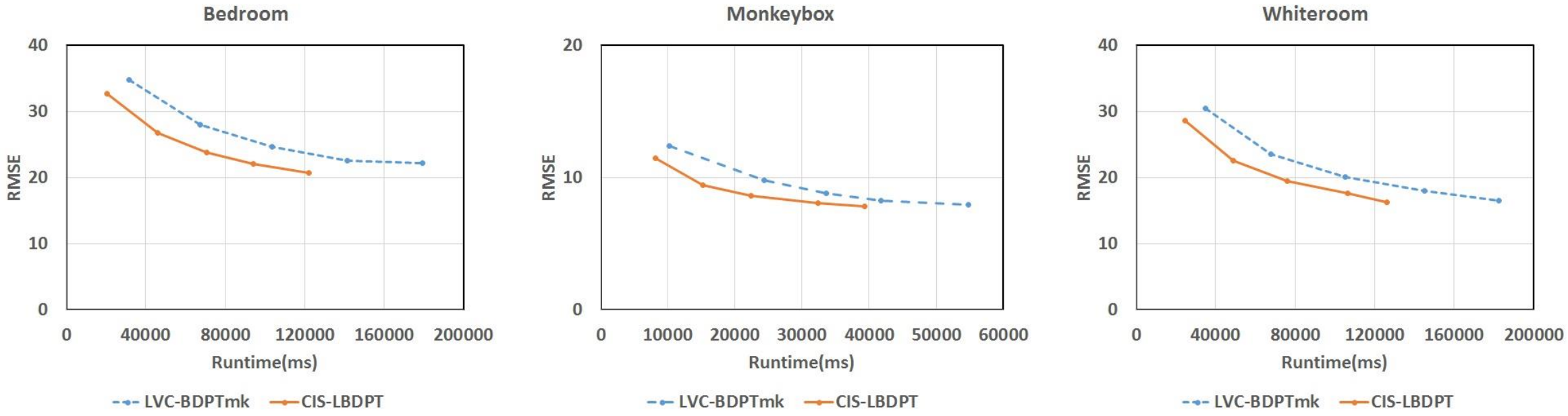
Scene 2 : White room – Fully lit

Results - Runtime

	Effective SPP	LVC-BDPTmk (ms)	CIS-LBDPT (ms)	Speedup
MonkeyBox	8	18237	14228	1.28x
	16	39534	31376	1.26x
	24	61867	46169	1.34x
	32	84471	64214	1.31x
	40	104630	78082	1.34x
Whiteroom	4	34890	24780	1.48x
	8	68011	48988	1.38x
	12	105419	76138	1.38x
	16	145022	106276	1.36x
	20	182498	126046	1.44x
Bedroom	4	31595	20374	1.48x
	8	67509	46053	1.46x
	12	103618	70717	1.46x
	16	141286	94269	1.49x
	20	179323	122053	1.46x

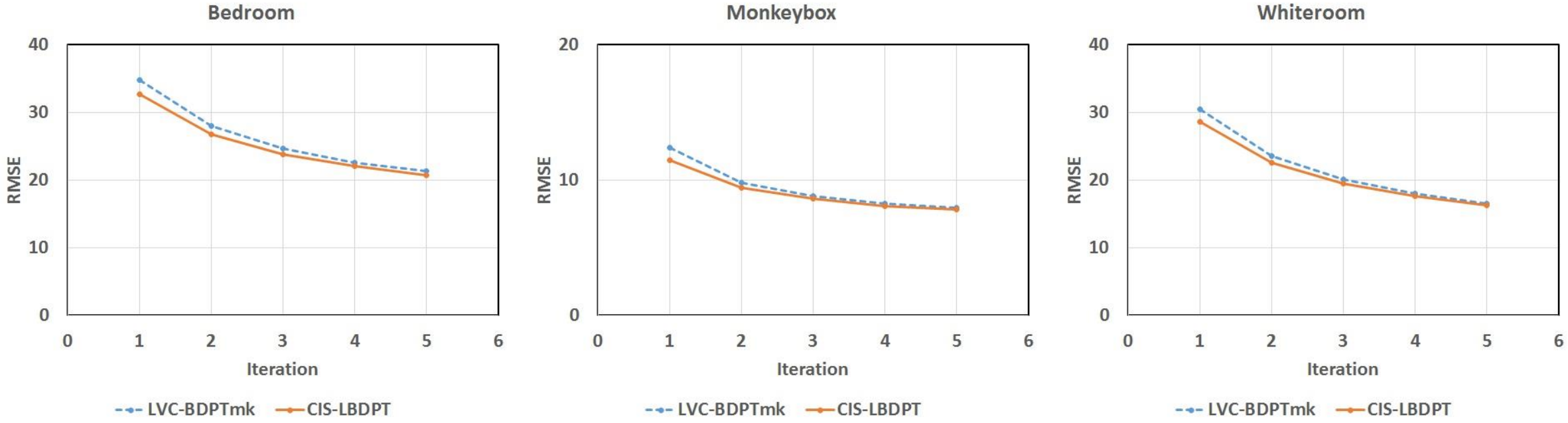
- Results obtained by progressive rendering.
- Each iteration had 4 or 8 spp.
- Results indicate the performance benefits due sorting material evaluation requests.
- Code execution coherence enforced in same material **type** evaluation.
- Data execution coherence enforced in same material evaluation.

Results - Quality



- RMSE values computed using ground truth computed using naïve BDPT algorithm.
- Ground truth images consisted of very little noise.
- Our importance sampled connection scheme results in faster RMSE drop as time increases when compared against LVC-BDPTmk method.

Results - Quality



- Results indicate RMSE error drop across iterations.
- Both methods have similar RMSE as iterations increase due to image getting progressively refined better.
- Importance sampled connections yield better results in earlier iterations

Conclusion

- Presented two methods for GPU raytracing.
- Our PDACRT scheme very GPU friendly.
  - Simple algorithm – Easier adoption
  - Very less memory requirements – GPU friendly
- Our CIS – LBDPT method improves light transport on GPU.
  - Coherent evaluation scheme using sorting.
  - Importance sampled connections for better convergence.
- GPUs present a powerful parallel solution for accelerating rendering pipeline.
- GPU memory has been increasing and will be adopted by production rendering industry.
- Future commercial renderers both CPU and GPU based.

Publications

- Parallel Divide and Conquer Raytracing (PDACRT)
  Srinath Ravichandran and P.J.Narayanan
  Siggraph Asia 2013 – Technical Briefs

- Coherent and Importance Sampled LVC BDPT
  Srinath Ravichandran and P.J.Narayanan
  Siggraph Asia 2015 – Technical Briefs (under review)

Thank You

Questions?