ACCELERATION DATA STRUCTURE HARDWARE (AND SOFTWARE)

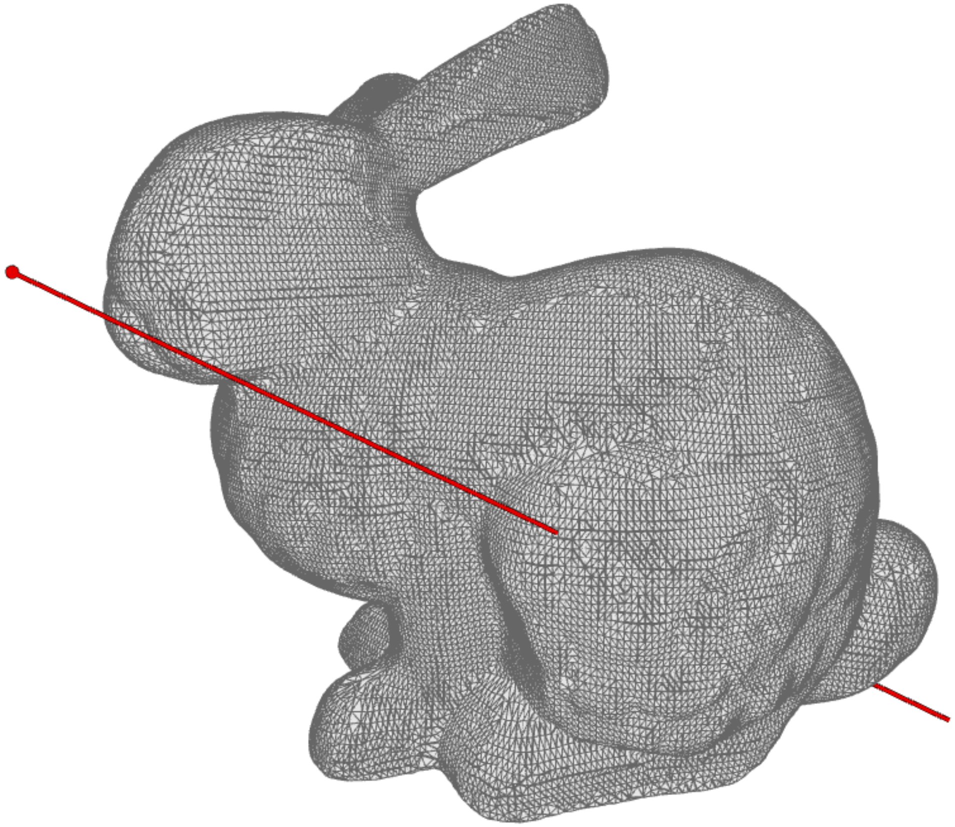
Timo Viitanen, Jul 27. 2019, SIGGRAPH 2019, LA, USA

OUTLINE

1. What is a BVH?
2. RTX BVH maintenance
3. State of the art in BVH build hardware
4. Open problems

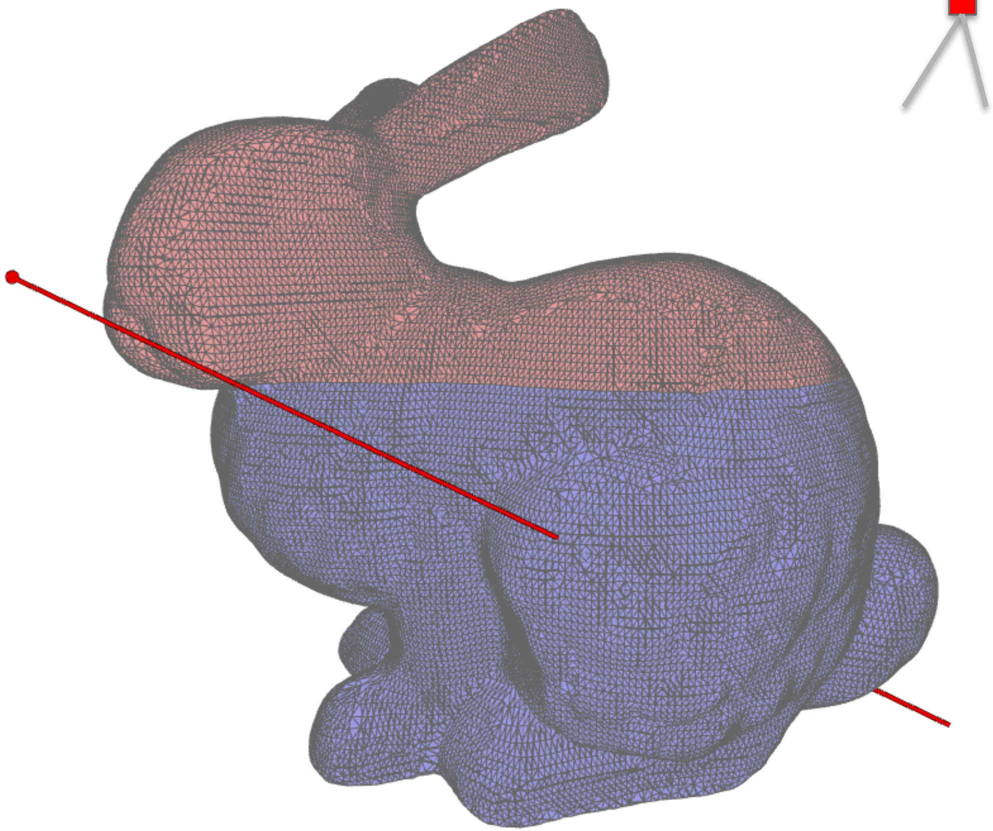
WHAT IS A BVH?

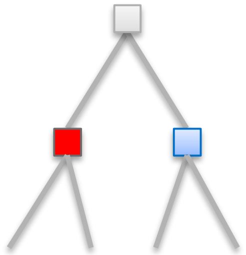
BOUNDING VOLUME HIERARCHY (BVH)



BOUNDING VOLUME HIERARCHY (BVH)

Each node divides a set of triangles into two child subsets

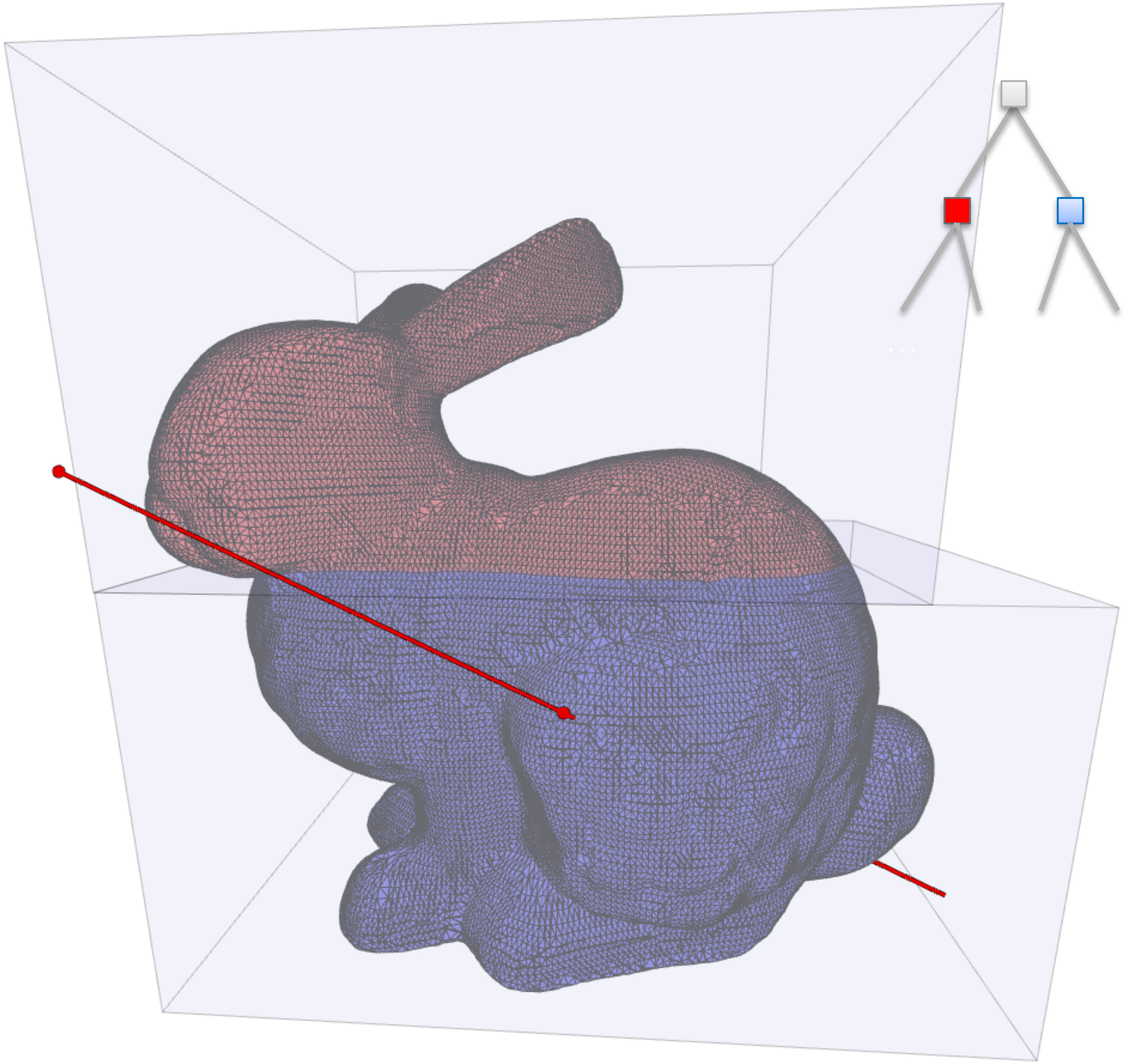




BOUNDING VOLUME HIERARCHY (BVH)

Each node divides a set of triangles into two child subsets

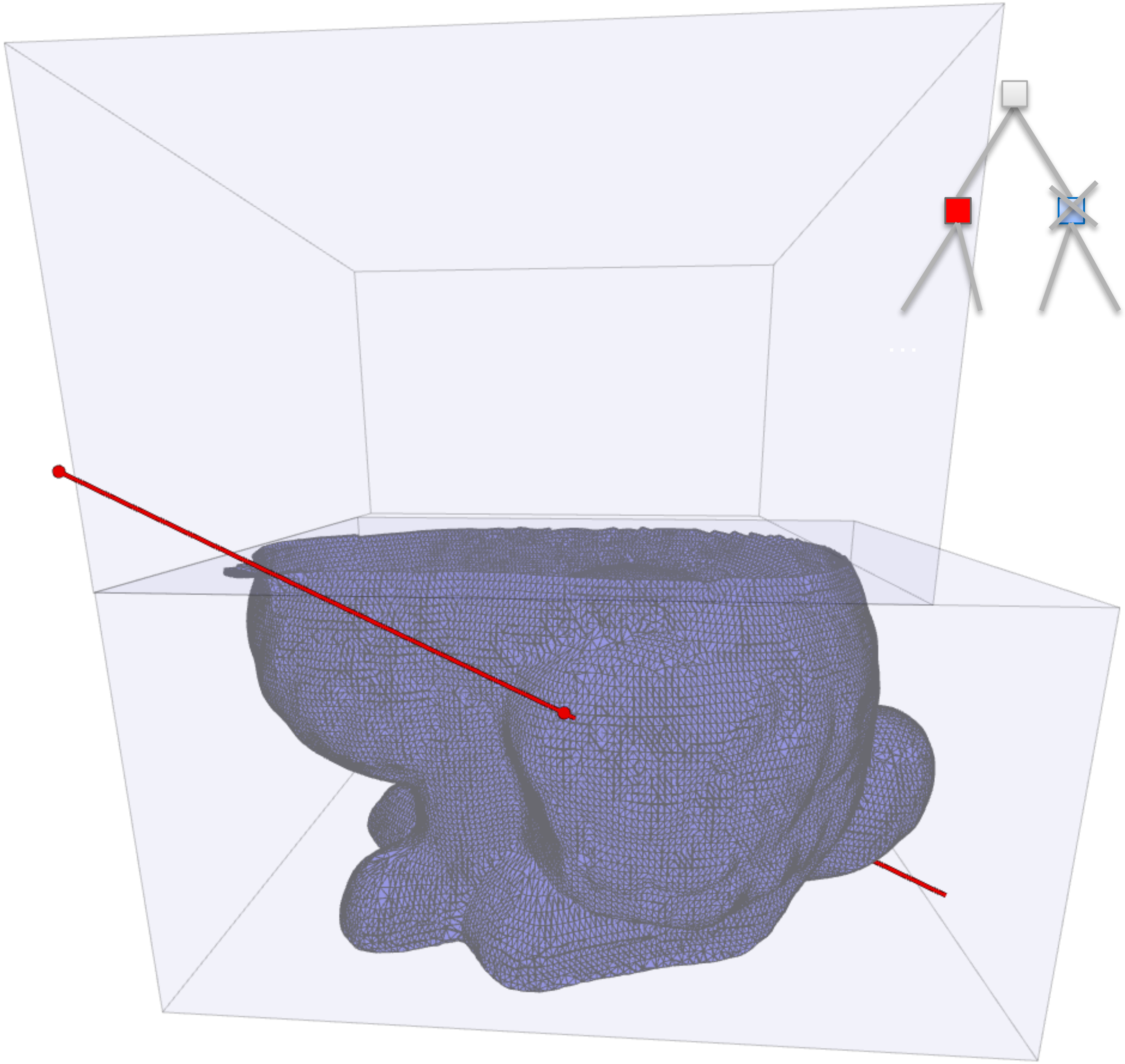
…And stores their bounding boxes



BOUNDING VOLUME HIERARCHY (BVH)

Each node divides a set of triangles into two child subsets

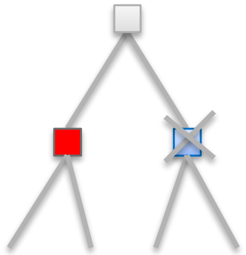
…And stores their bounding boxes

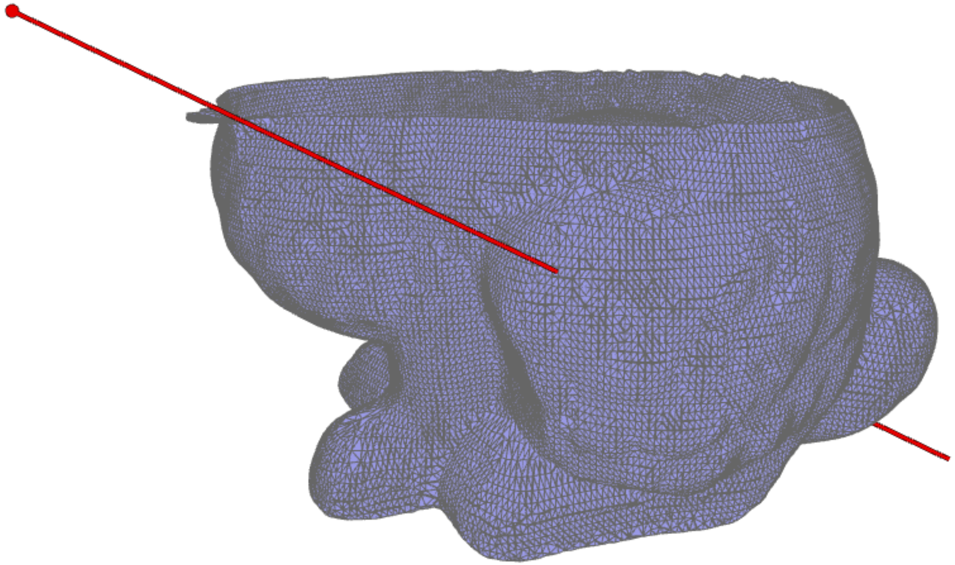


BOUNDING VOLUME HIERARCHY (BVH)

Each node divides a set of triangles into two child subsets

…And stores their bounding boxes

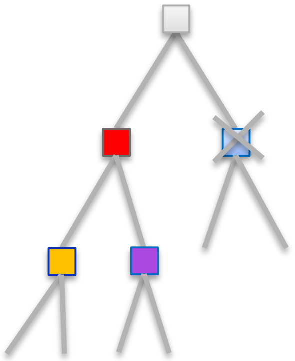


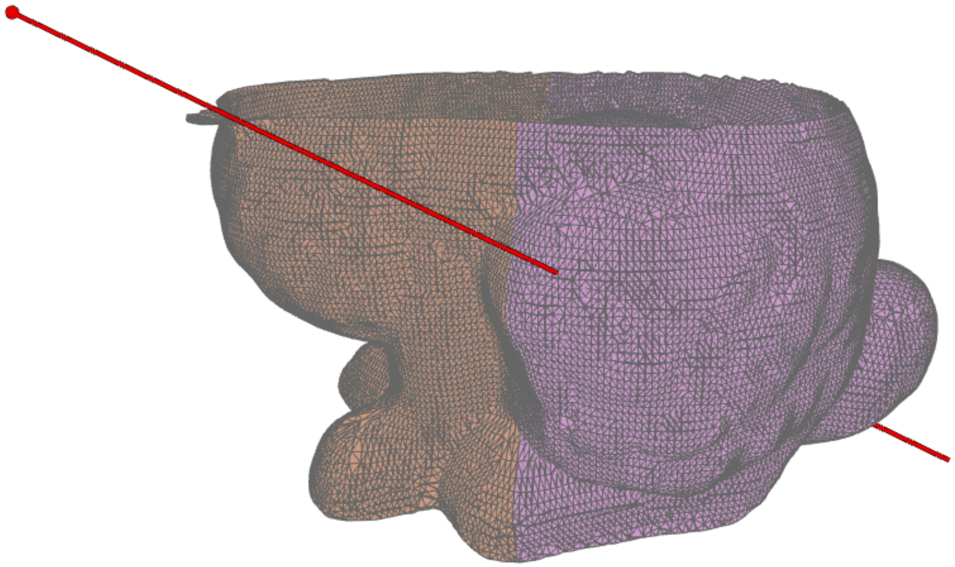


BOUNDING VOLUME HIERARCHY (BVH)

Each node divides a set of triangles into two child subsets

…And stores their bounding boxes

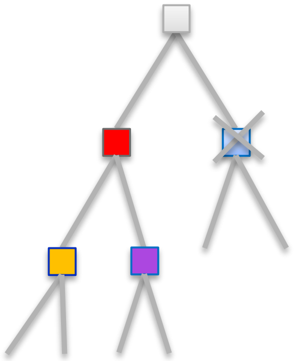


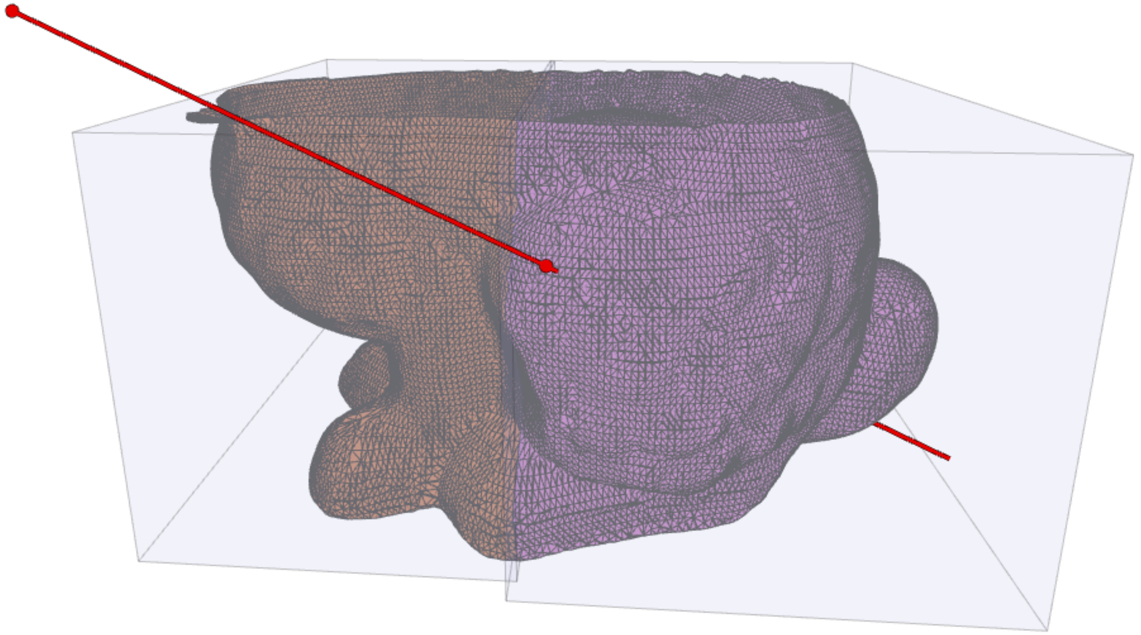


BOUNDING VOLUME HIERARCHY (BVH)

Each node divides a set of triangles into two child subsets

…And stores their bounding boxes

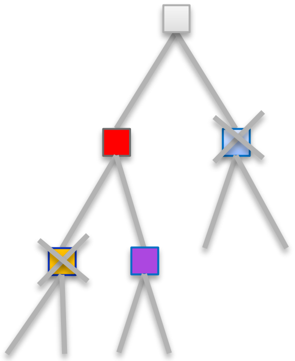


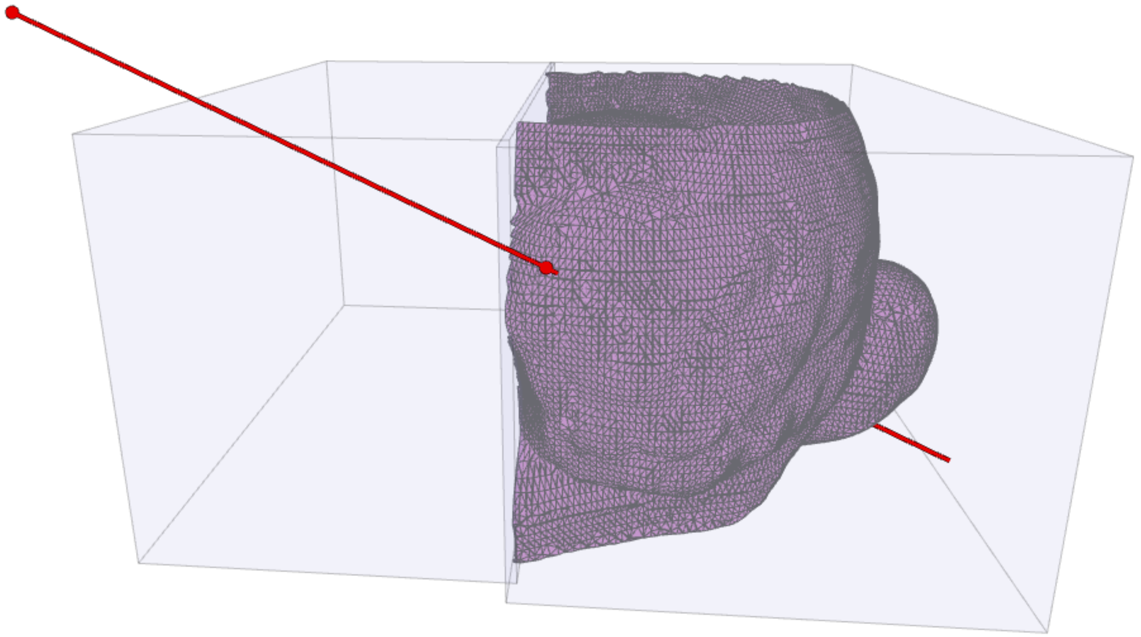


BOUNDING VOLUME HIERARCHY (BVH)

Each node divides a set of triangles into two child subsets

…And stores their bounding boxes





BOUNDING VOLUME HIERARCHY (BVH)

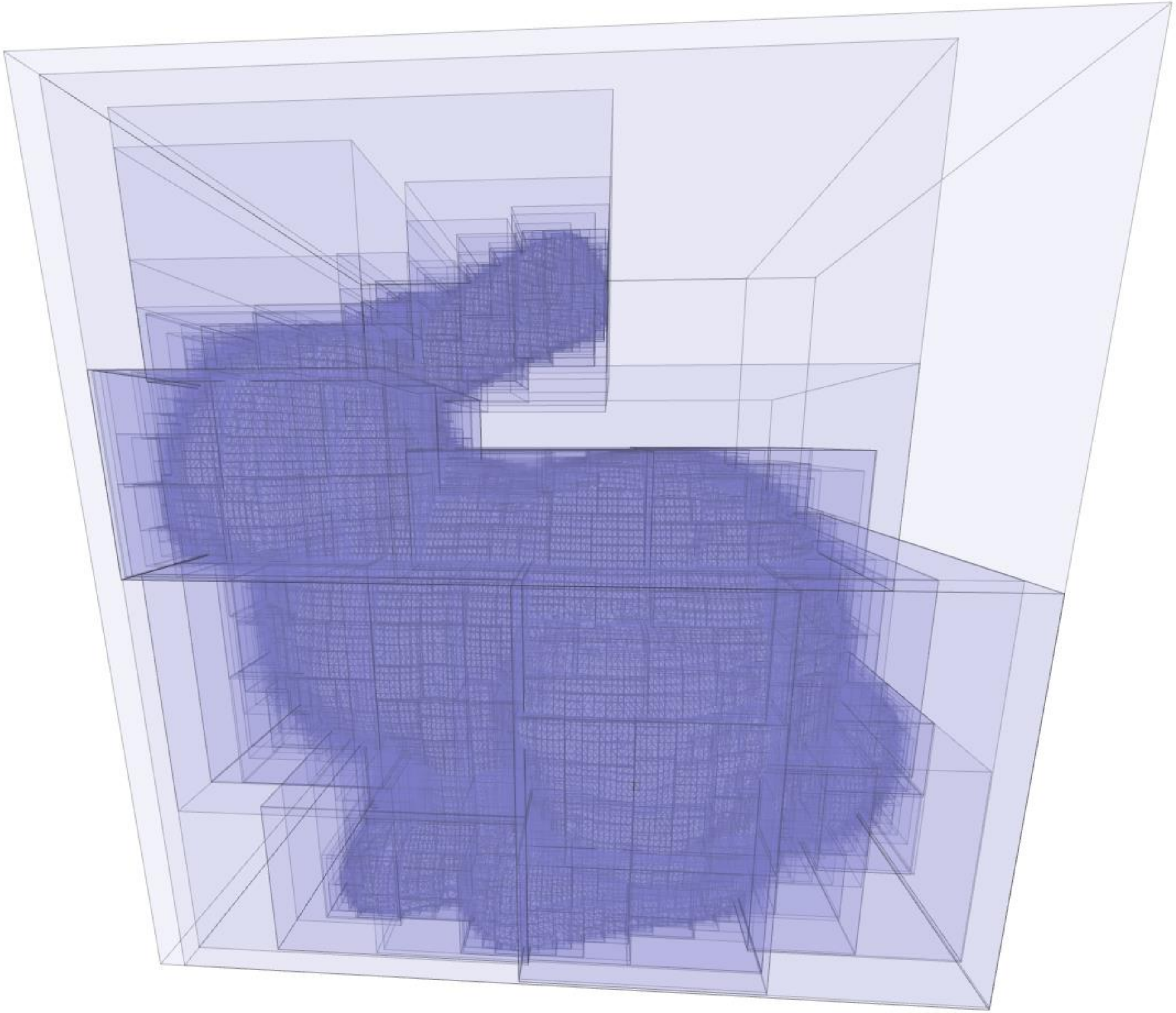
Each node divides a set of triangles into two child subsets

…And stores their bounding boxes

Allows roughly $O(\log n)$ ray-scene collision tests

...Before tracing rays, we have to build a BVH

Tree build is an expensive task and may sometimes dwarf the cost of ray tracing



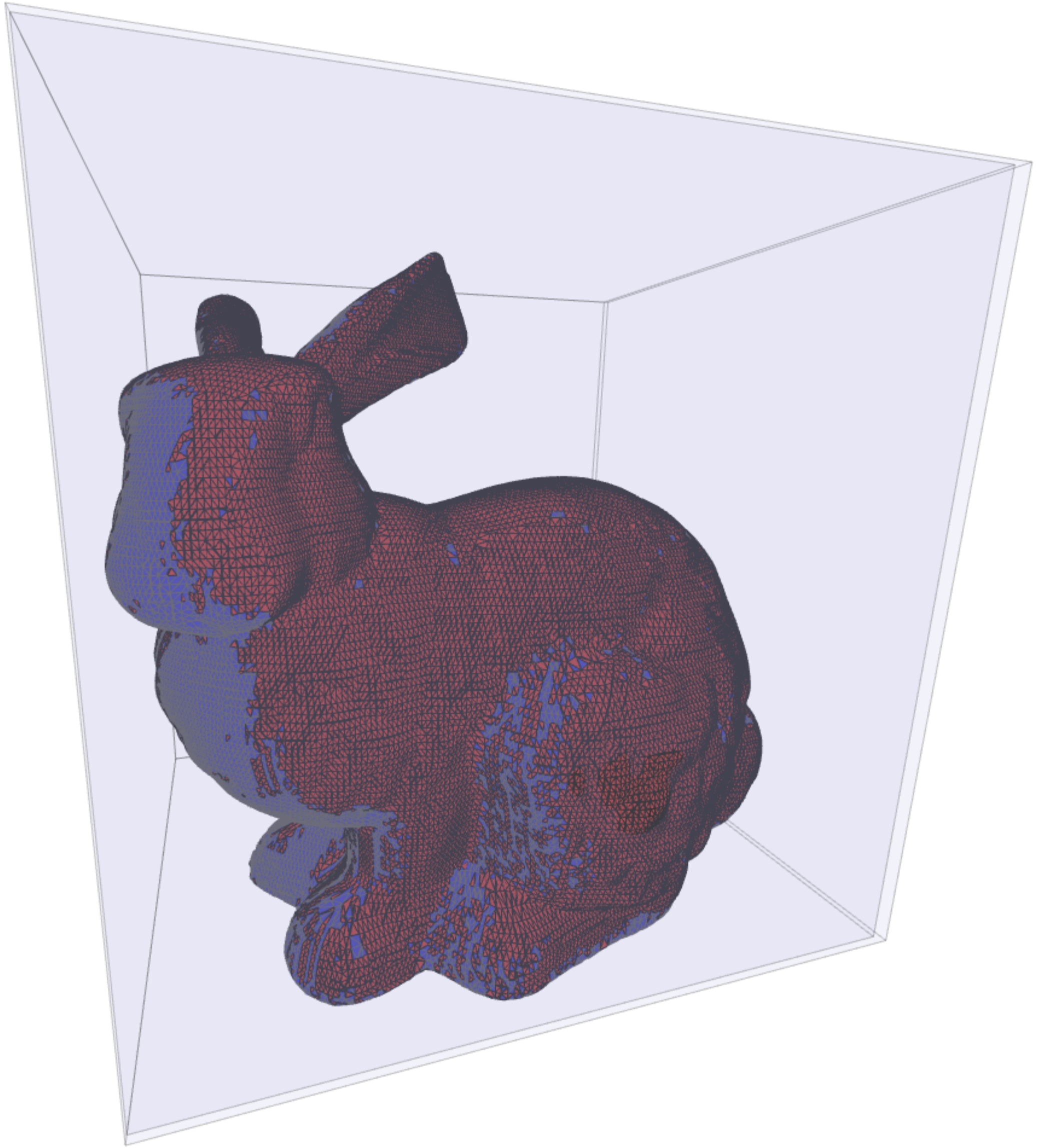
TREE QUALITY

Traversal speed (i.e tree quality) depends on choice of splits

E.g. random splits →

Quality is related to node AABB surface area (or related SAH)

Tradeoff curve between fast, low-quality and slow, high-quality builders



REFITTING

Given an animation where vertices move but the mesh topology stays the same, we can *refit* a BVH instead of rebuilding: just read the new triangle data and recompute bounding boxes.

- For example, in RTX refitting is ~10x faster than a full build.
- Quality may degrade over a long animation → sometimes trees are periodically refreshed with a rebuild.

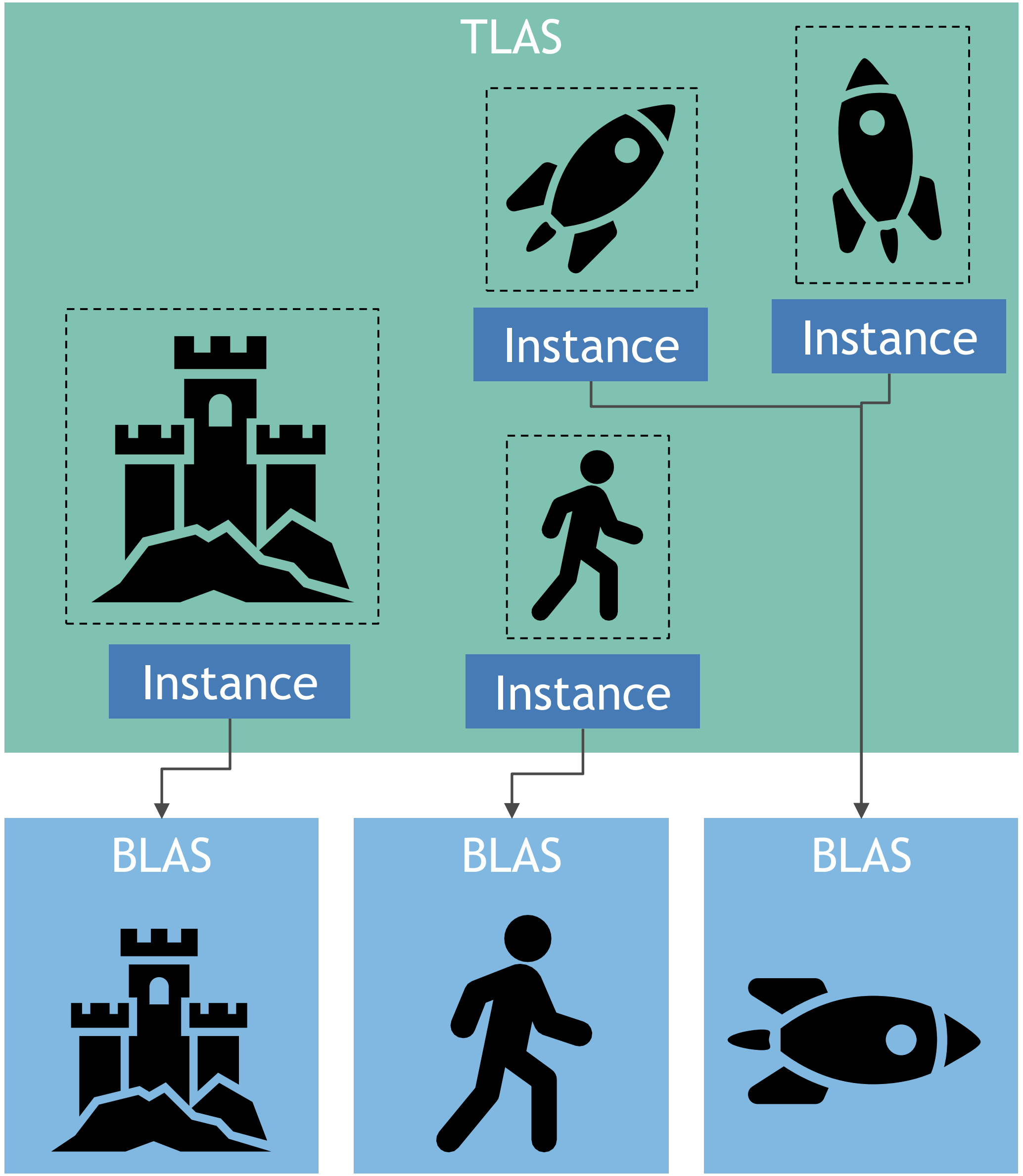
INSTANCING

Bottom Level Acceleration Structure (BLAS):

BVH of triangles

Top Level Acceleration Structure (TLAS):

BVH of *instances* with a BLAS reference and a transform matrix



INSTANCING

Static geometry → Build once

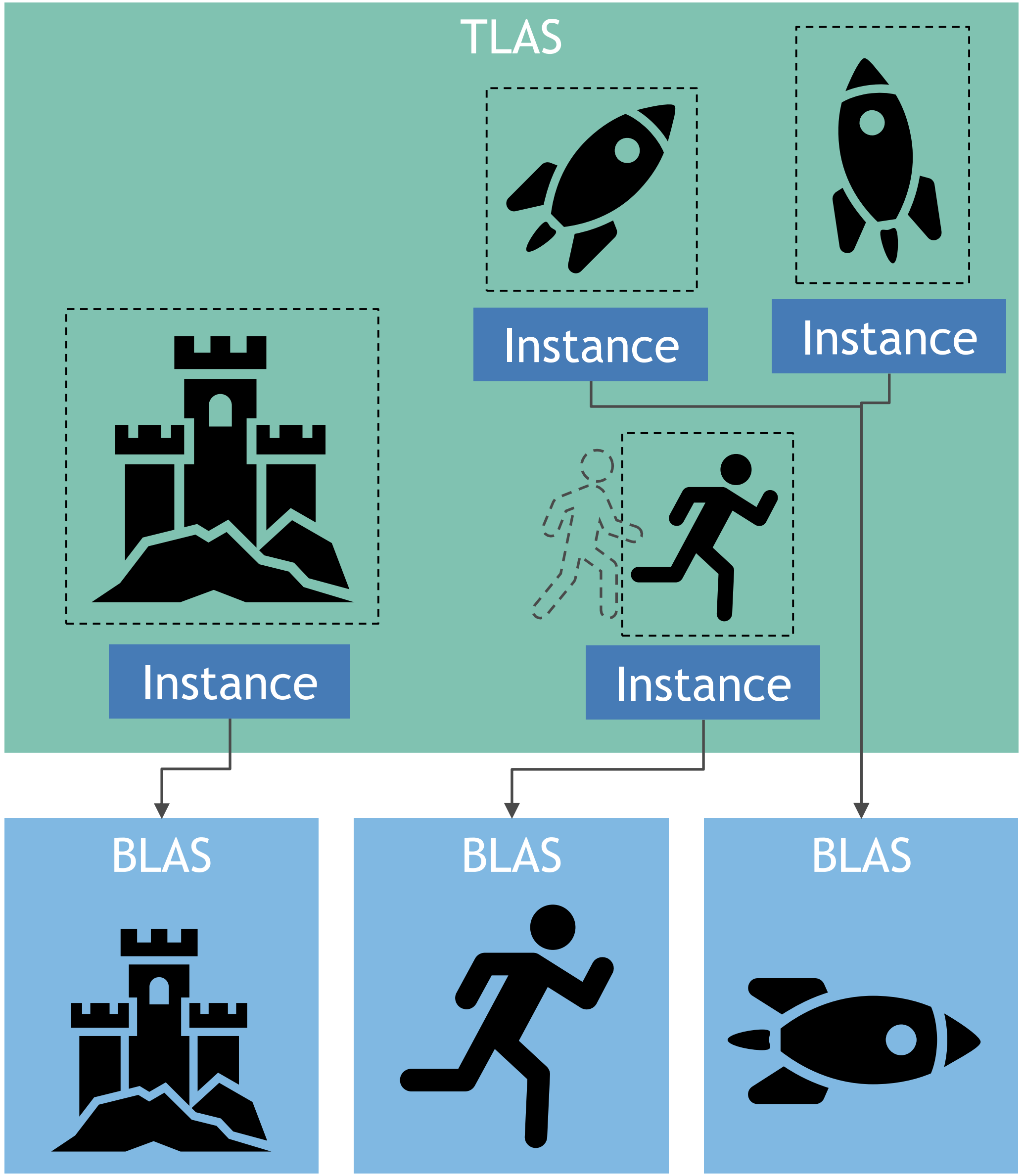
Instancing → Share a BLAS between objects

Rigid body animation

→ Modify the transform matrix

Mesh deformation → Refit

Only run a full rebuild for difficult cases



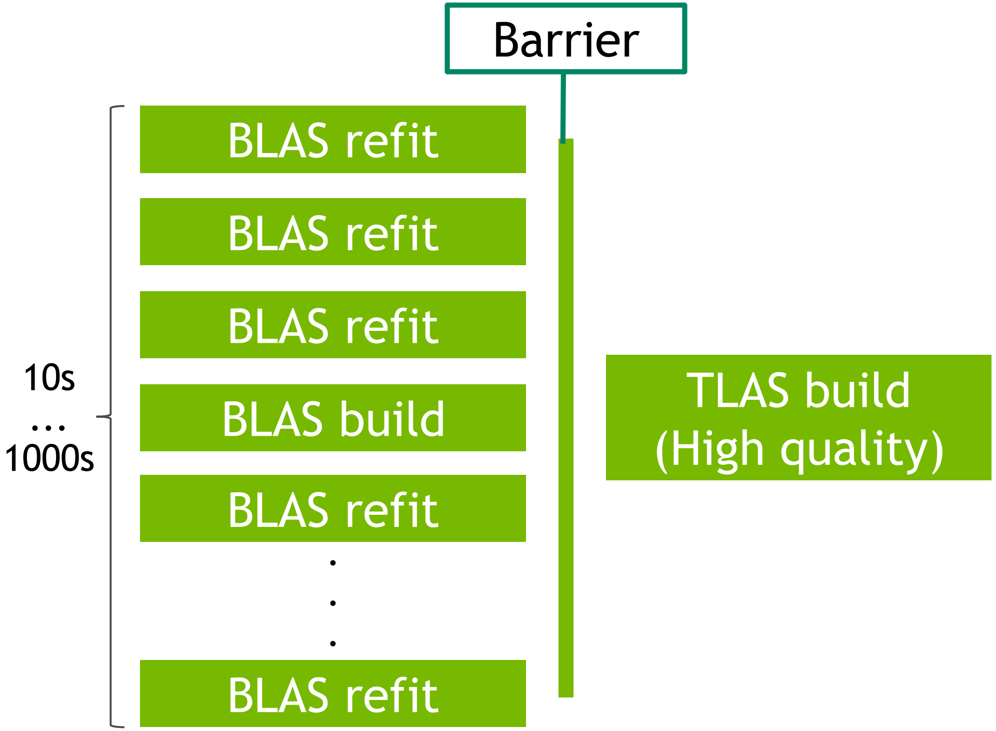
RTX BVH MANAGEMENT

RTX BVH BUILDER

- Supports refits and instancing
- Perf rules of thumb:
    - Refit ~1000Mtris/s
    - Build ~100Mtris/s
- Storage ~33B/tri after compaction
- Continual optimization: e.g. ca. +36% average build throughput since launch (in internal benchmark)

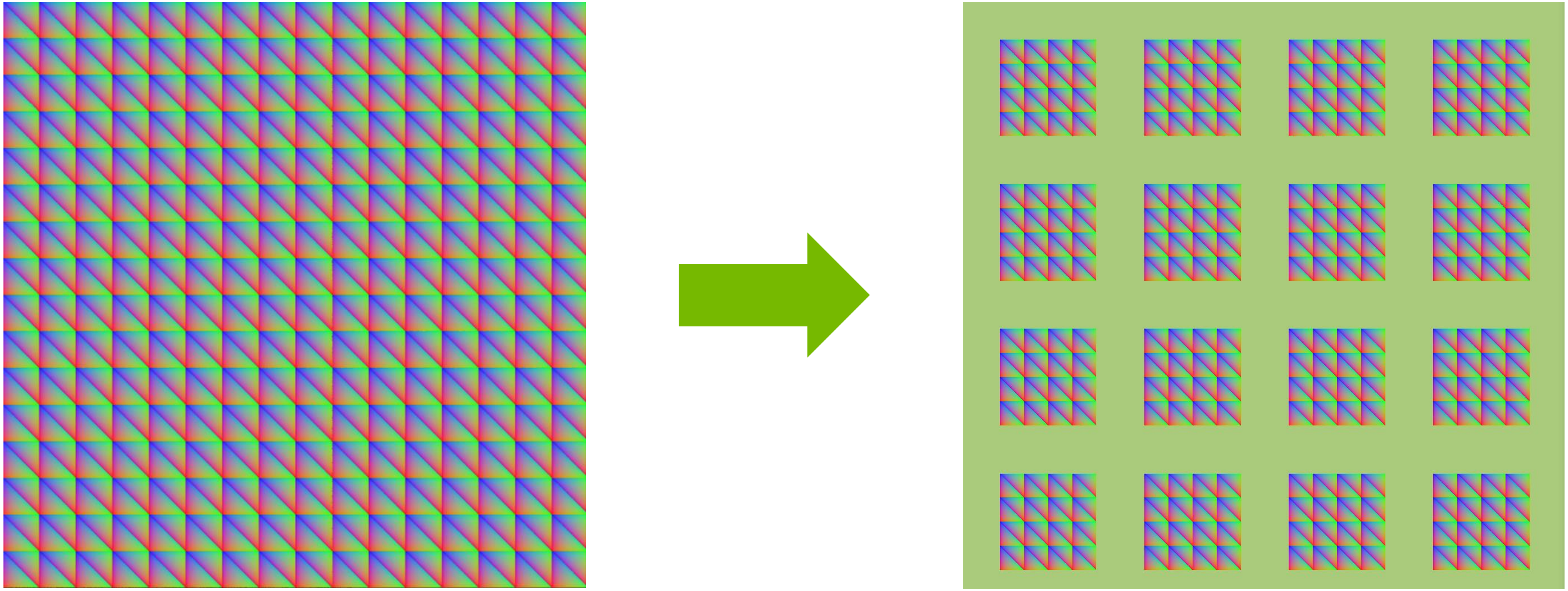
TYPICAL GAME BVH WORKLOAD

- 60fps game → must fit in small fraction of 16.66ms
- Many BLAS FAST_BUILD refits
- Many of the BLASes are small, 10s to 1000s of triangles
- TLAS build of ~1000...10000 instances
- Already the result of heavy optimization: geometry culling (BFV), build throttling (Metro), overlapping other work with BVH
  - E.g. In BFV builds took 64 ms on the first try, optimized down to 1.15 ms (Shyshkovtsov 2019)



RTX INSTANCED BUILD PERFORMANCE

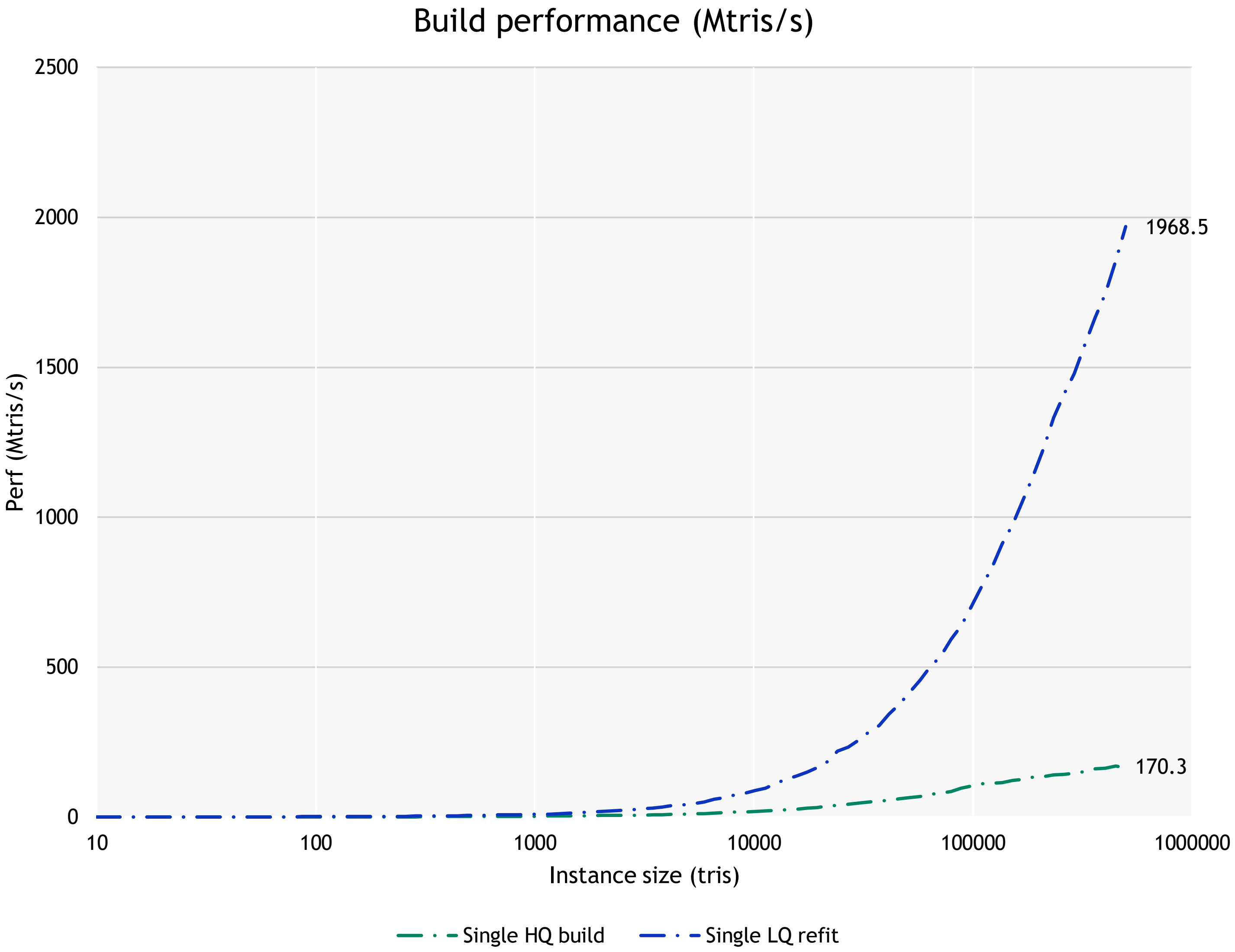
Toy benchmark



RTX BUILDER PERFORMANCE

Toy benchmark

A single build needs to be very large to utilize the GPU

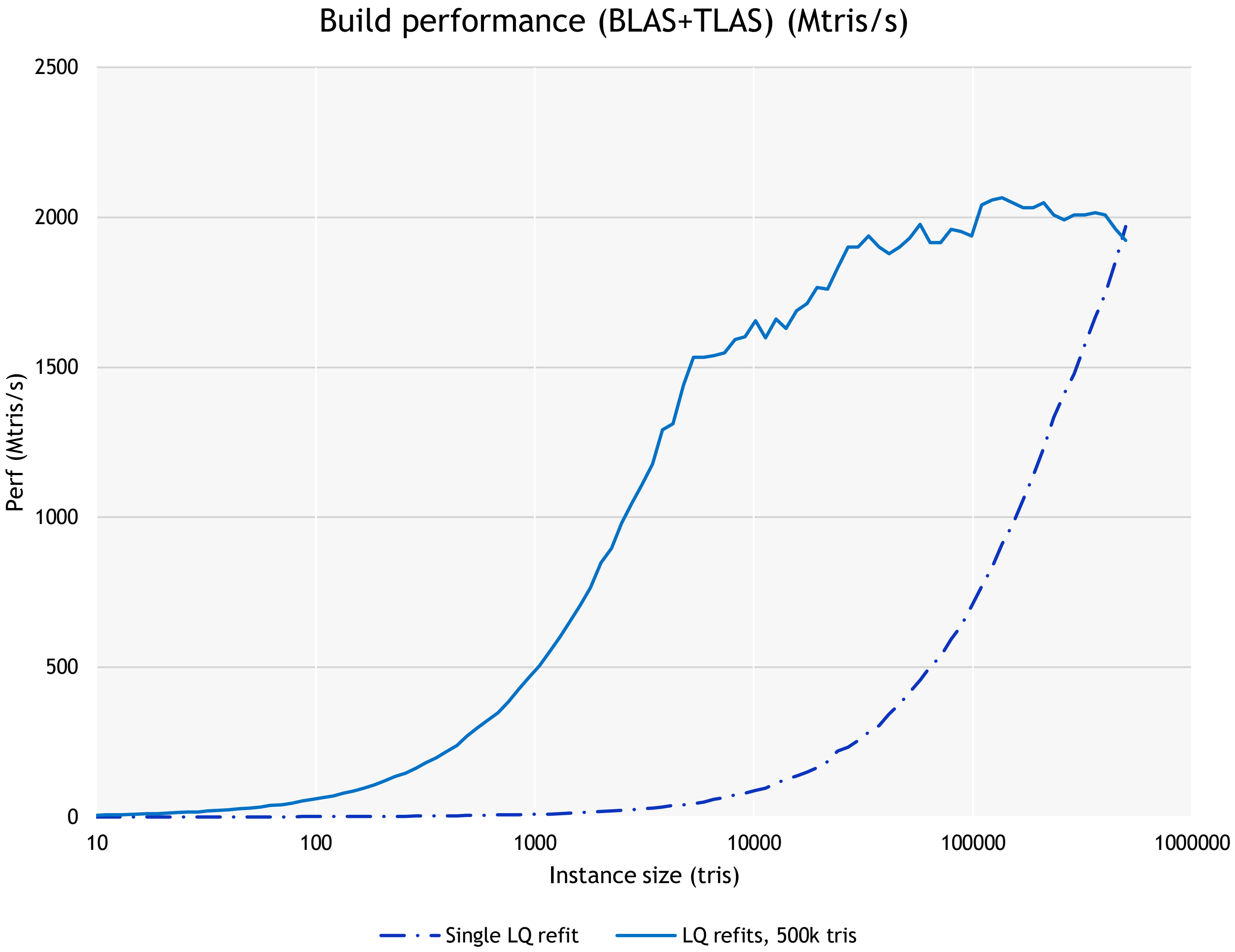


RTX BUILDER PERFORMANCE

Toy benchmark

A single build needs to be very large to utilize the GPU

Groups of smaller BLAS builds stay efficient down to ~1000tri instances (~5000 for refits). With smaller instances, performance starts to fall off.

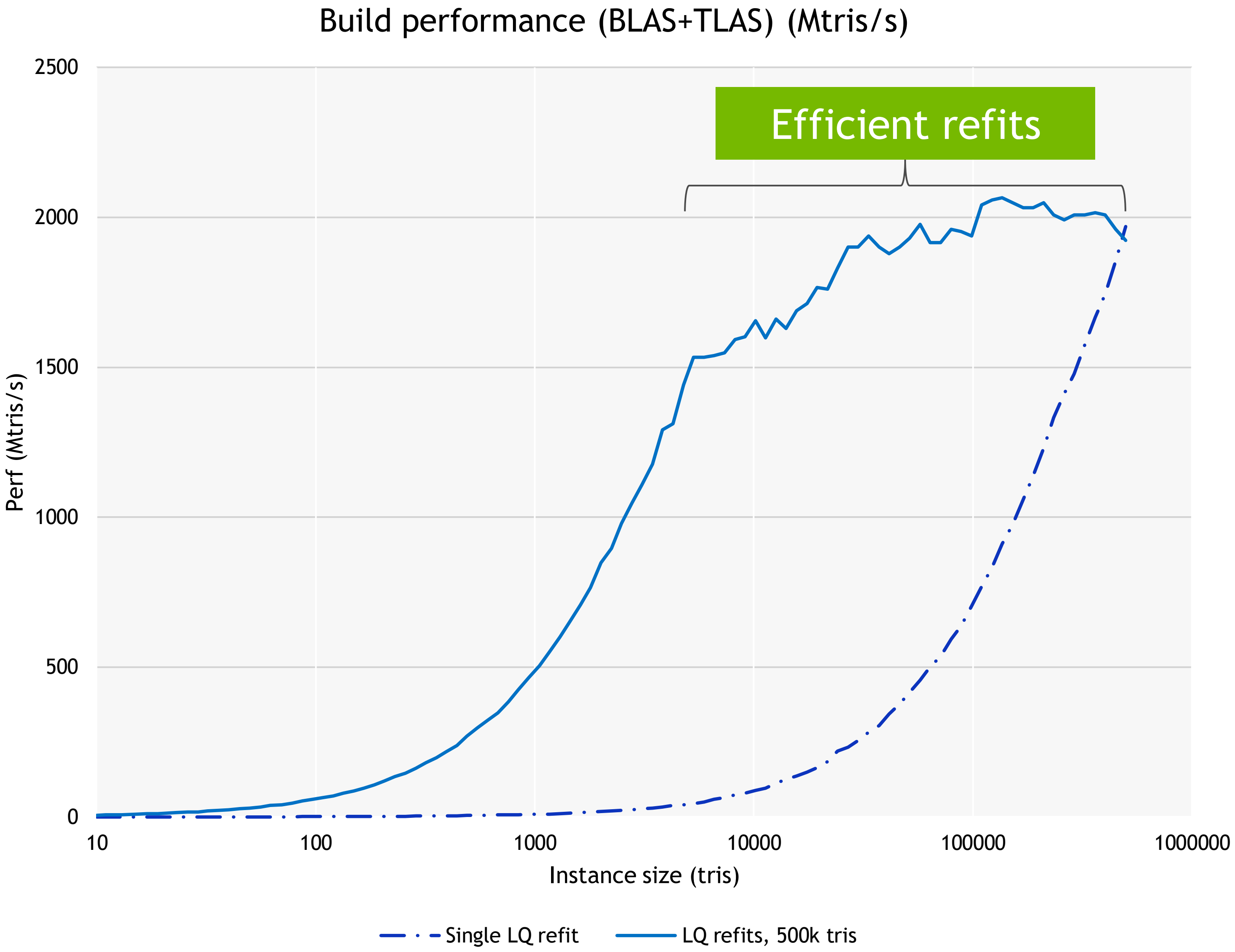


RTX BUILDER PERFORMANCE

Toy benchmark

A single build needs to be very large to utilize the GPU

Groups of smaller BLAS builds stay efficient down to ~1000tri instances (~5000 for refits). With smaller instances, performance starts to fall off.

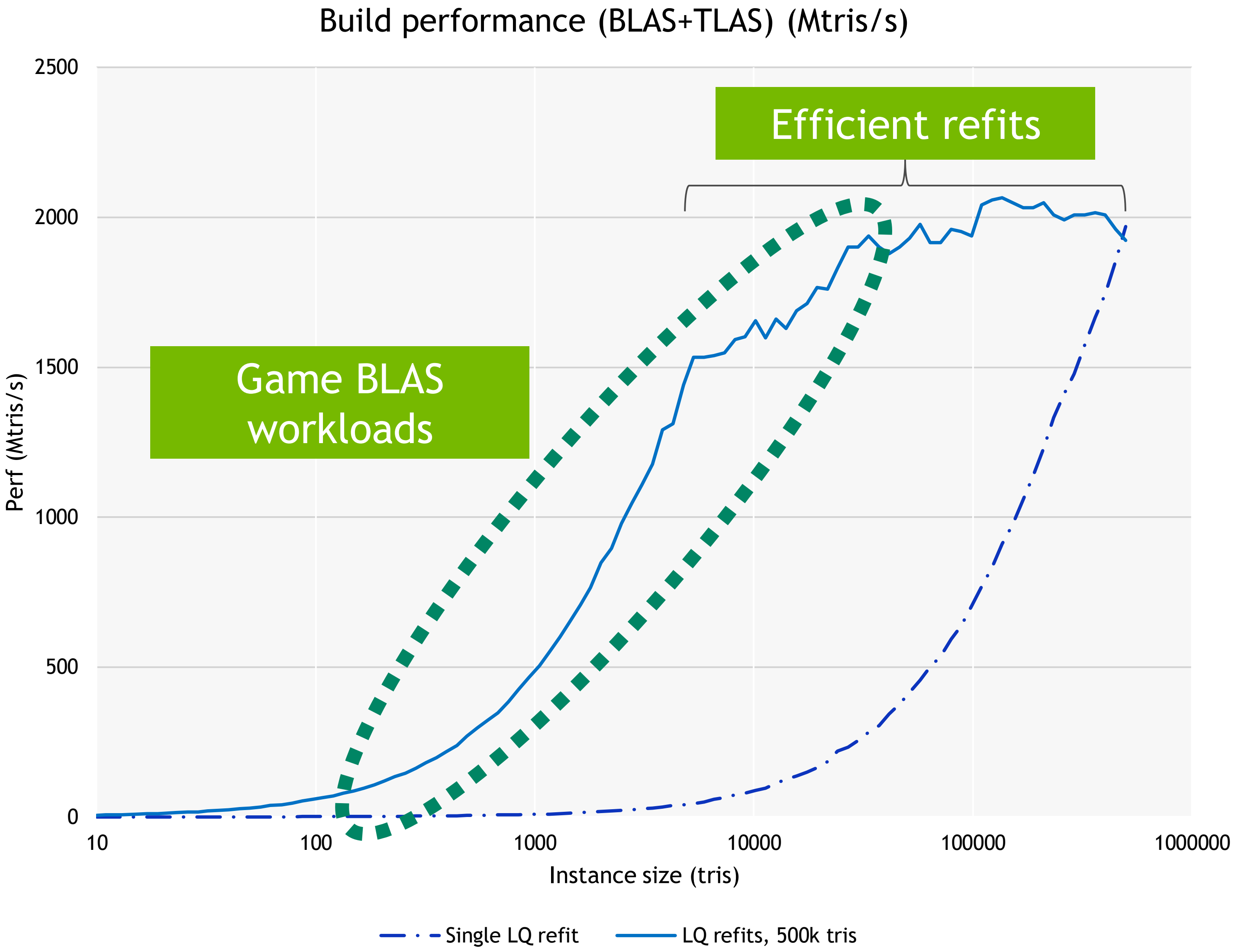


RTX BUILDER PERFORMANCE

Toy benchmark

A single build needs to be very large to utilize the GPU

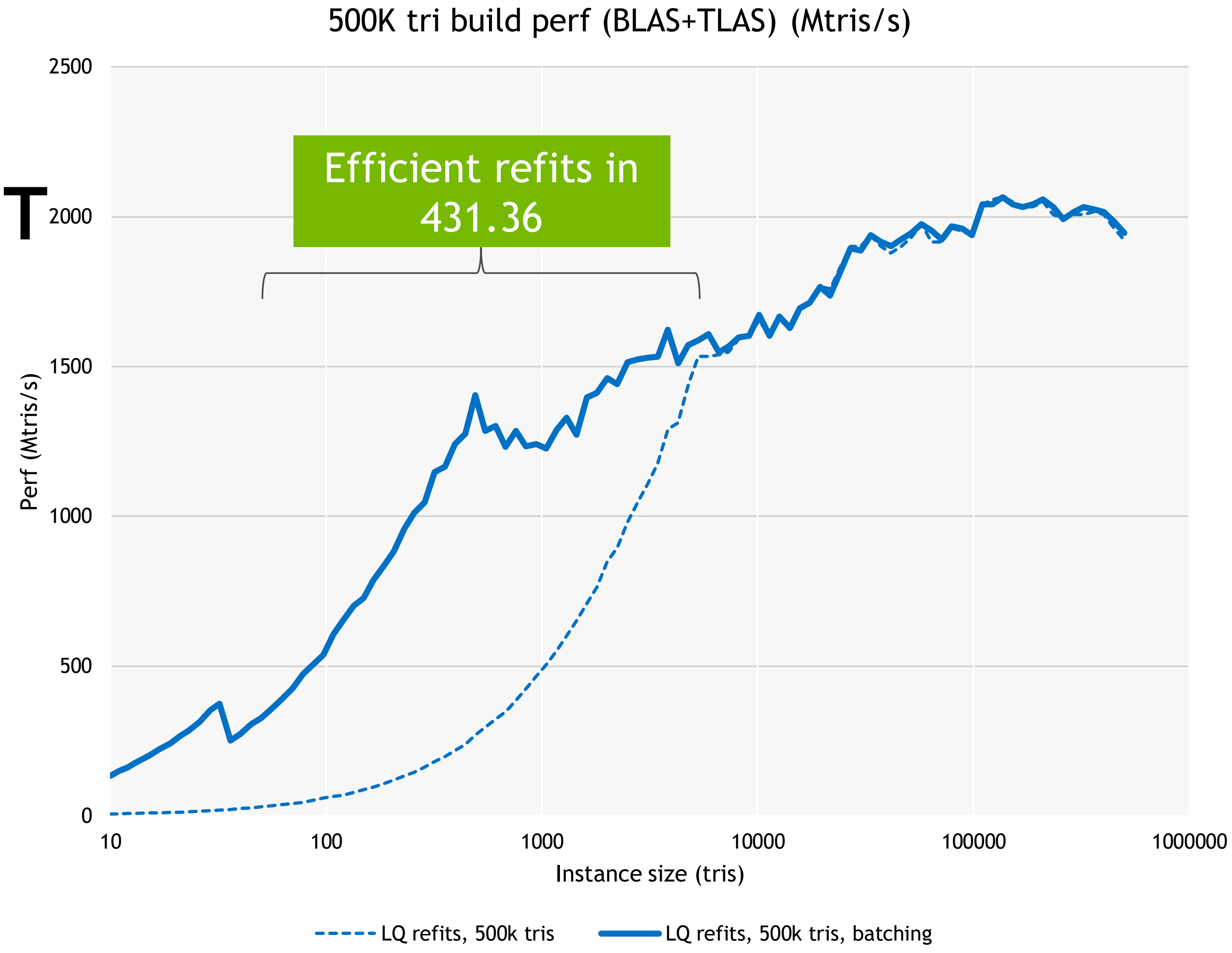
Groups of smaller BLAS builds stay efficient down to ~1000tri instances (~5000 for refits). With smaller instances, performance starts to fall off.



SMALL BUILD/REFIT OPTIMIZATIONS

New optimizations in recent driver (431.36) : heavily improved performance on batches of small builds.

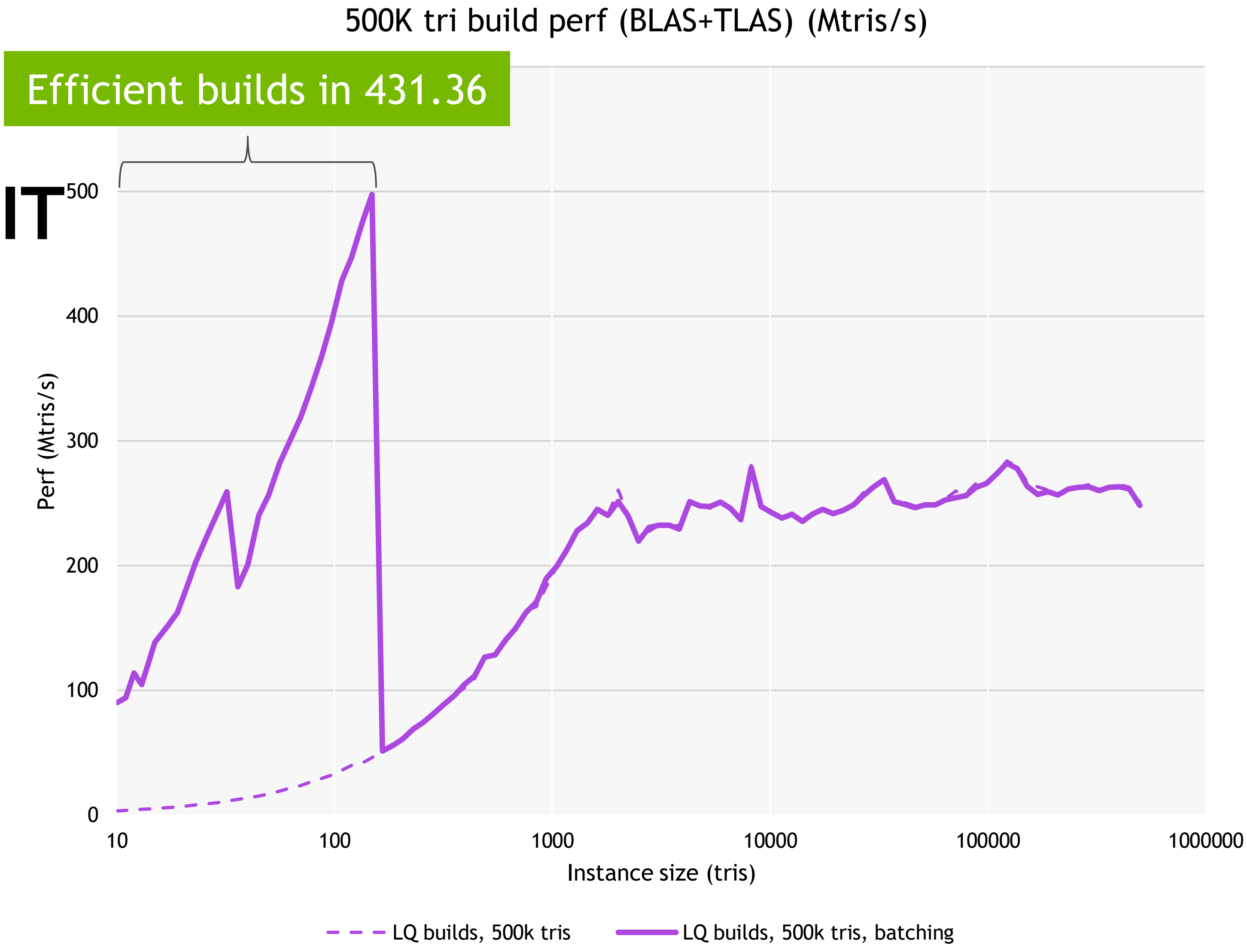
So far supports FAST_BUILD refits



SMALL BUILD/REFIT OPTIMIZATIONS

New optimizations in recent driver (431.36) : heavily improved performance on batches of small builds.

So far supports FAST_BUILD refits (and builds below a size threshold)



Efficient builds in 431.36

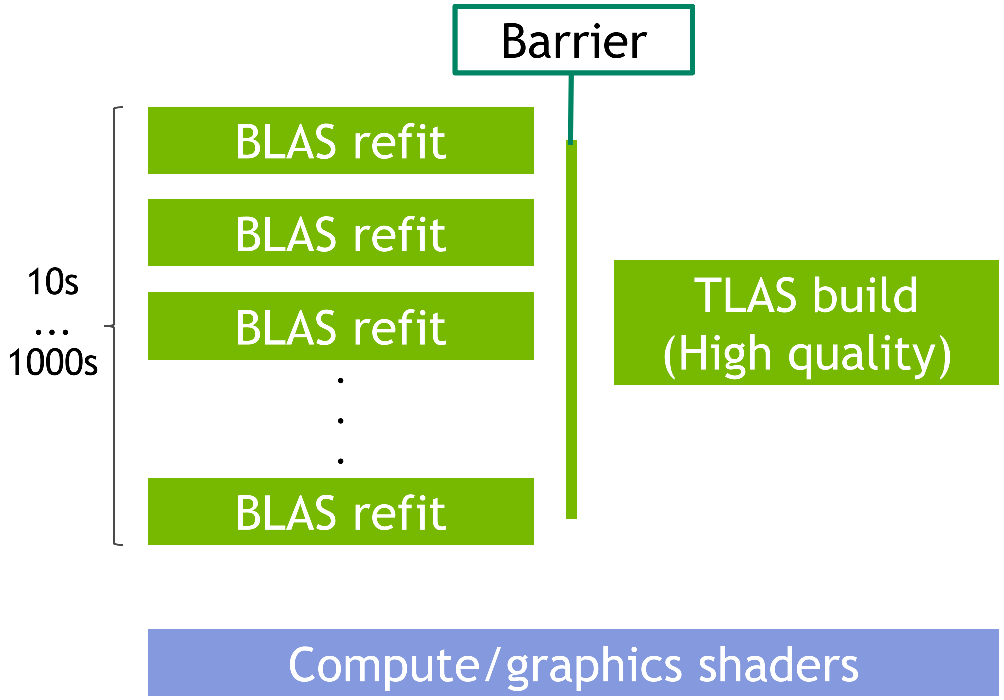
OVERLAPPING

BVH builds have low utilization

Overlap asynchronous compute and graphics work to hide BVH maintenance

- Used in e.g. BFV, Metro

Improved in Turing: can run graphics and compute concurrently in the same SM



RTX BVH MANAGEMENT

Difficult cases

INSTANCE GROUPING

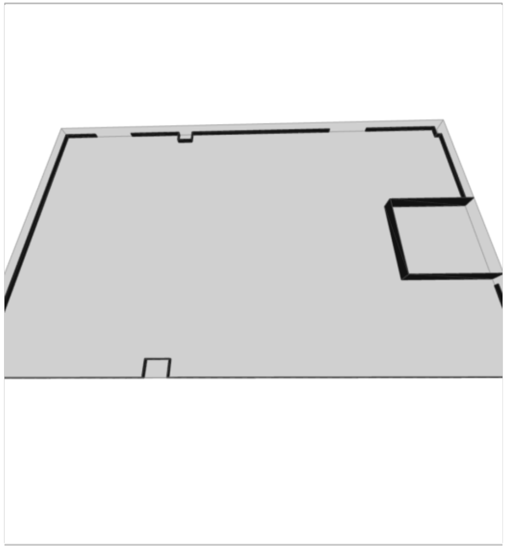
How to group geometry into BVH instances?

Adding RT effects to a rasterized game → convert drawcalls to instances? Both have geometry + shader program.

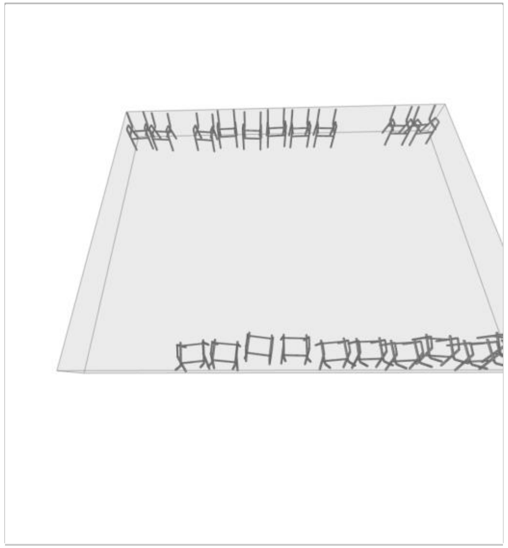
→1 instance per material shader?

INSTANCE PARTITIONING

Group by material, example instances



Instance 1



Instance 2

Instance 3

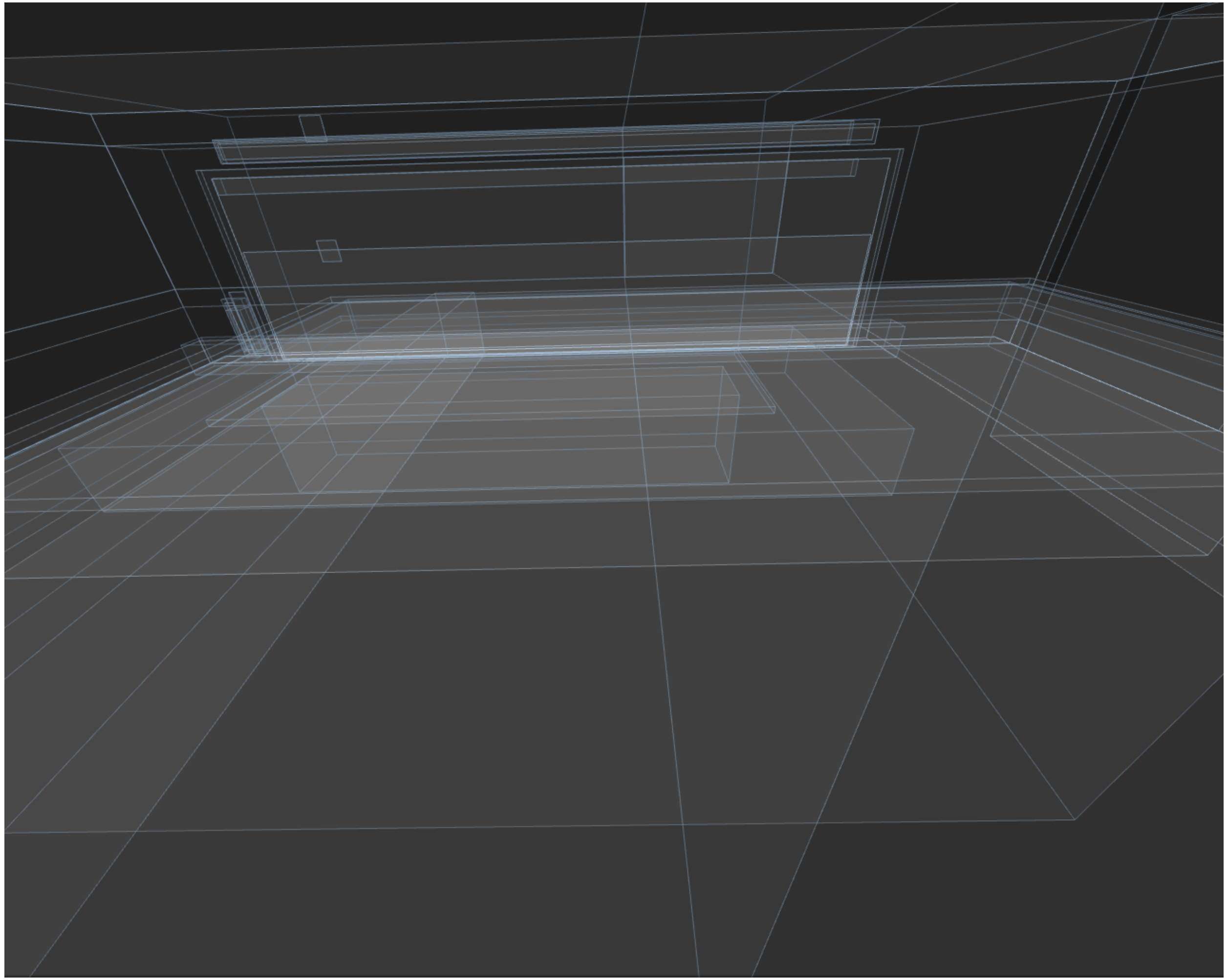
Instance 4

INSTANCE GROUPING

Group by material

Large, overlapping instance bounds with much empty space

- Many instance hits (have to transform ray to object coordinates for each hit instance)
- Bad TLAS quality



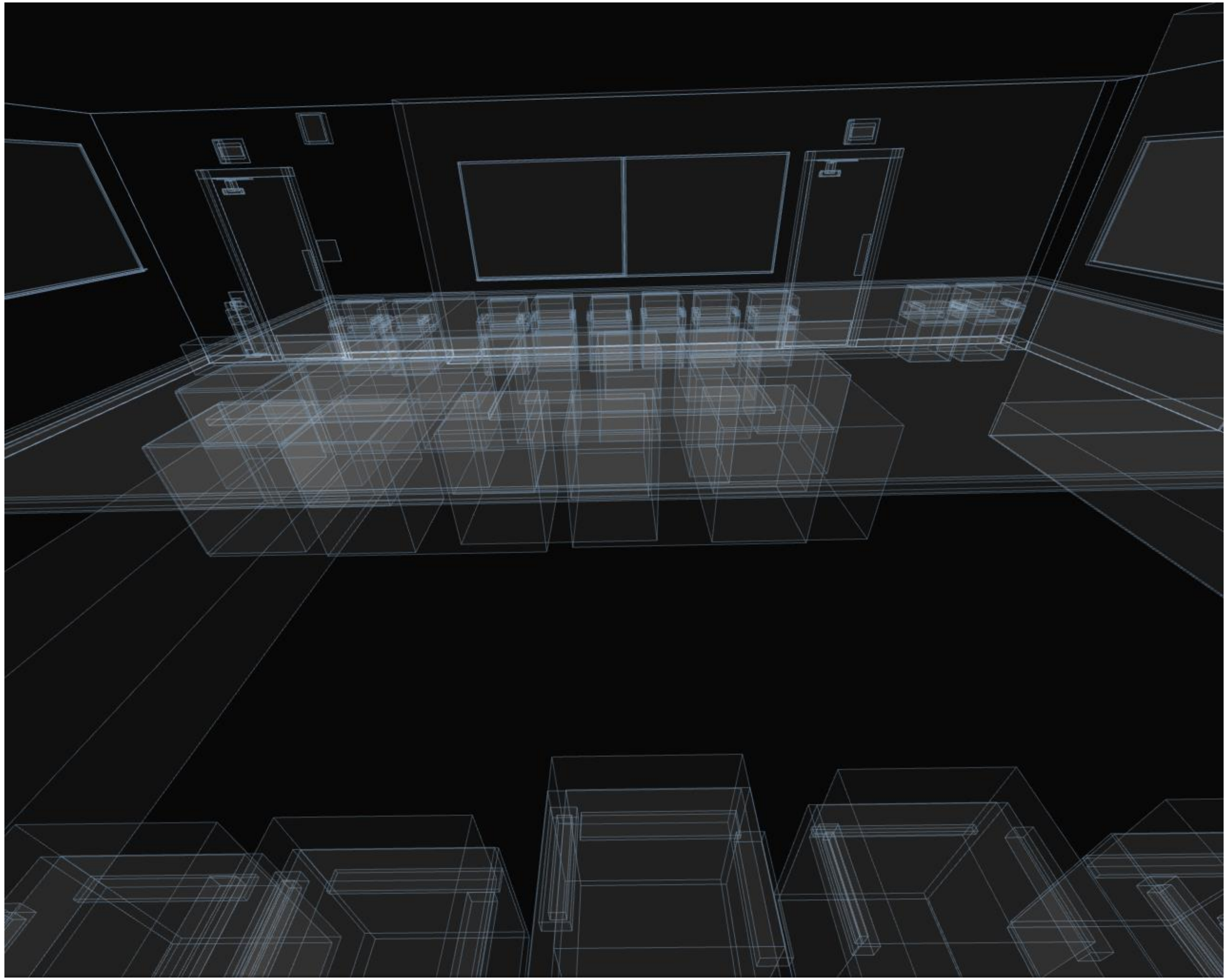
INSTANCE GROUPING

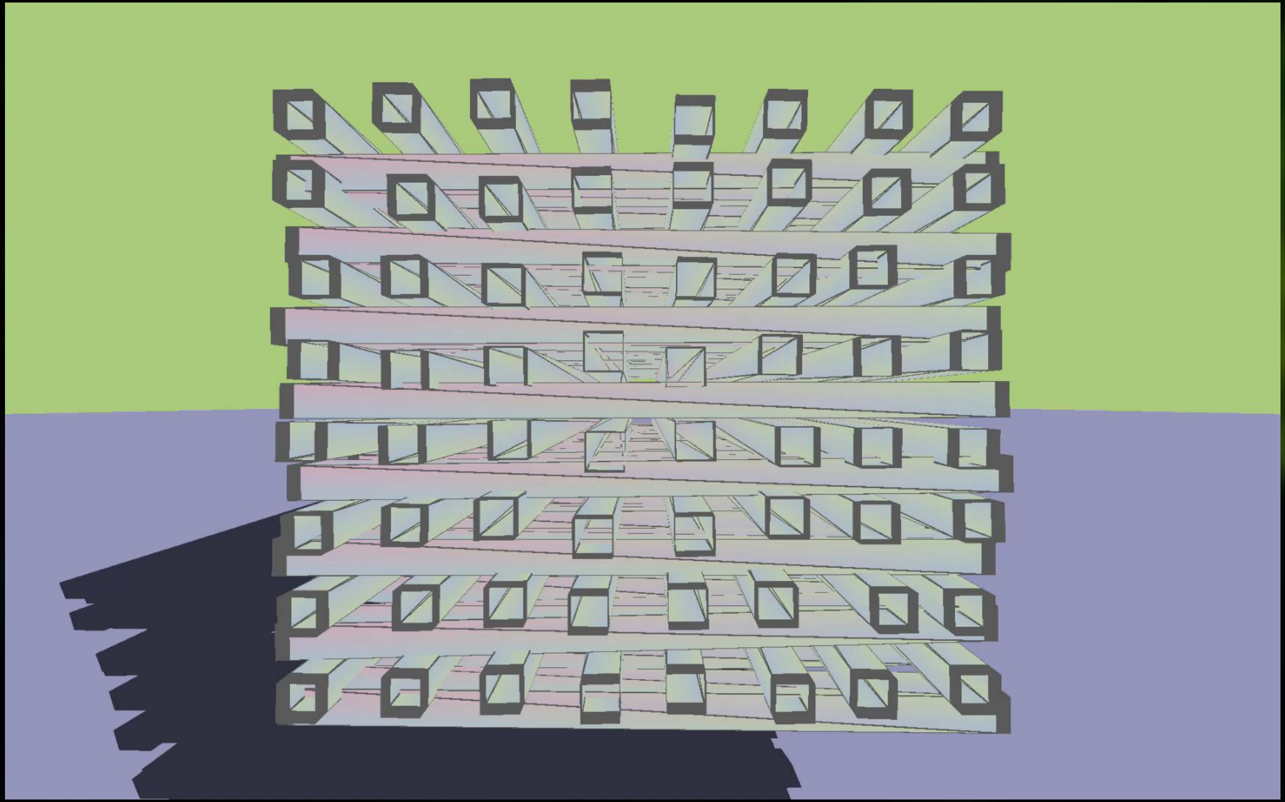
Group by locality

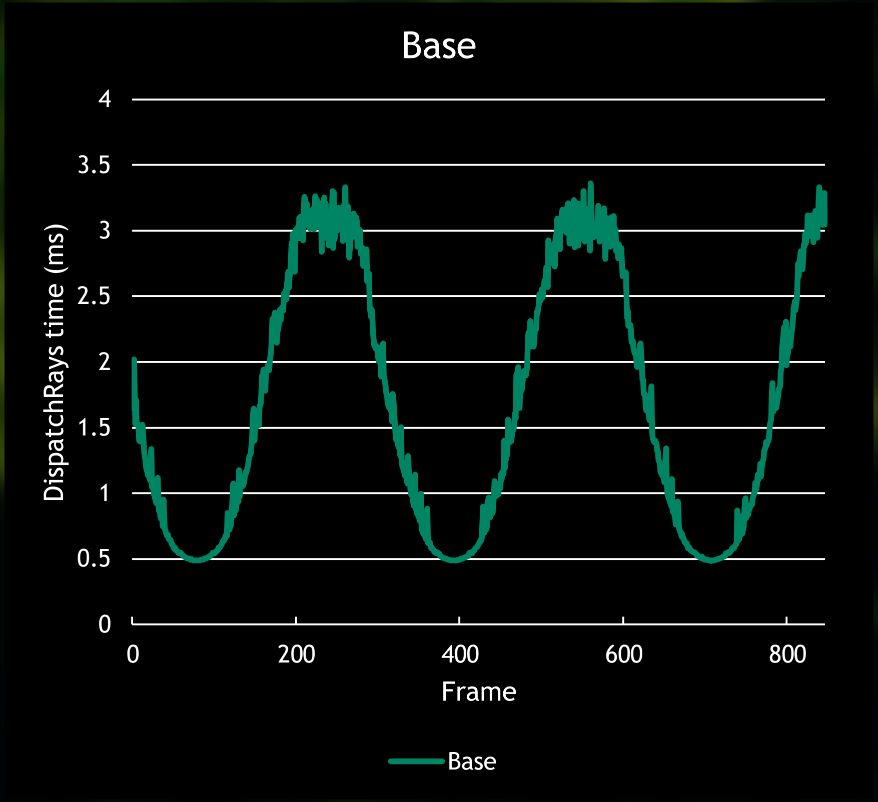
BVH instancing works better with discrete physical objects as instances

Note: Can still have multiple geometries in a BLAS with different material shaders

(Open question: how to make a builder robust to instance grouping)







SHARP TRIANGLES, DISPATCHRAYS() 0.5MS..3MS

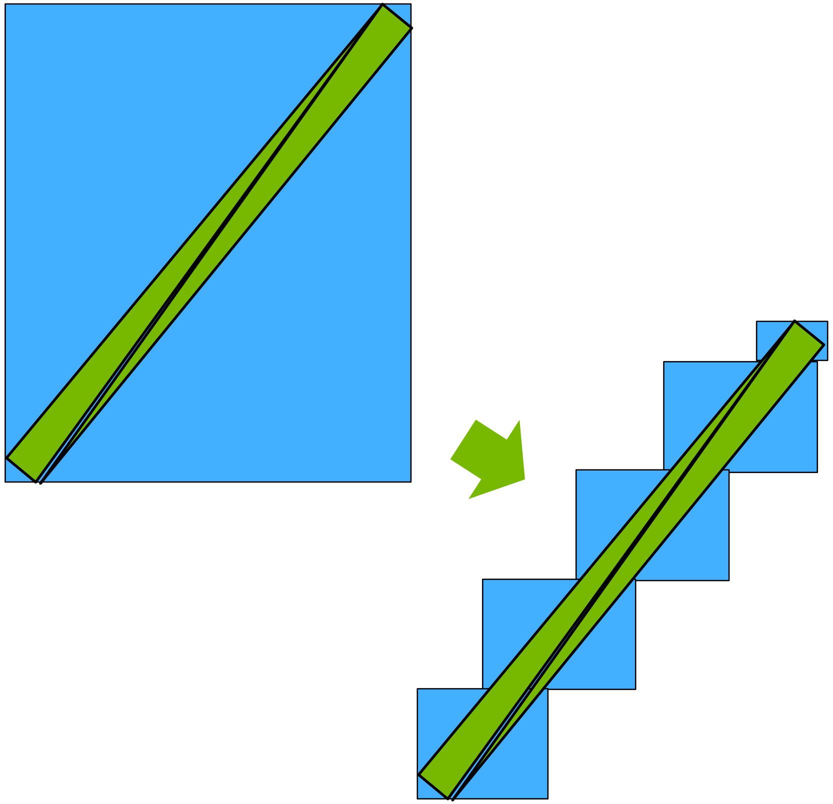
LONG, NARROW TRIANGLES

When not axis-aligned, long, narrow triangles have large AABBs that catch many false-positive rays

→ A BLAS with enough such triangles can hurt RT perf

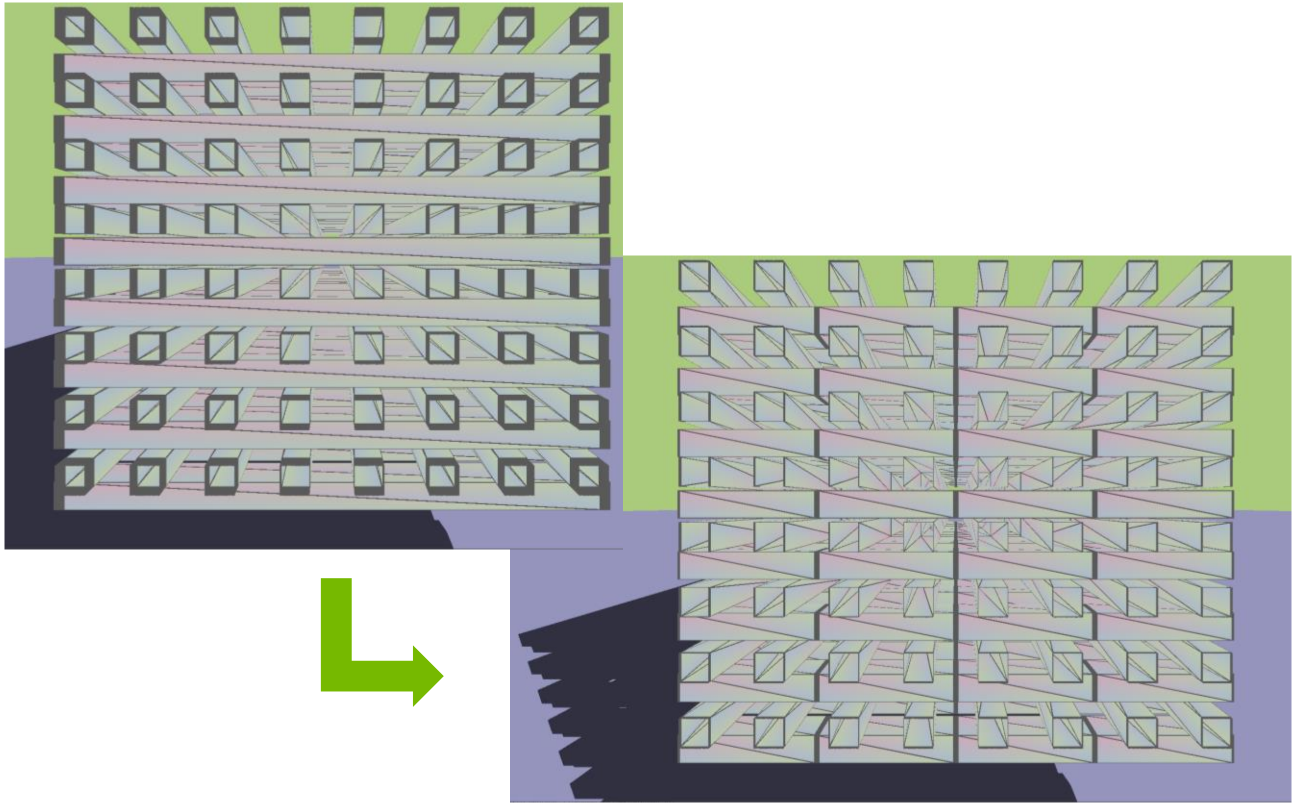
Can be mitigated by triangle splitting – but limited split budget, so too many sharp triangles overload the mechanism

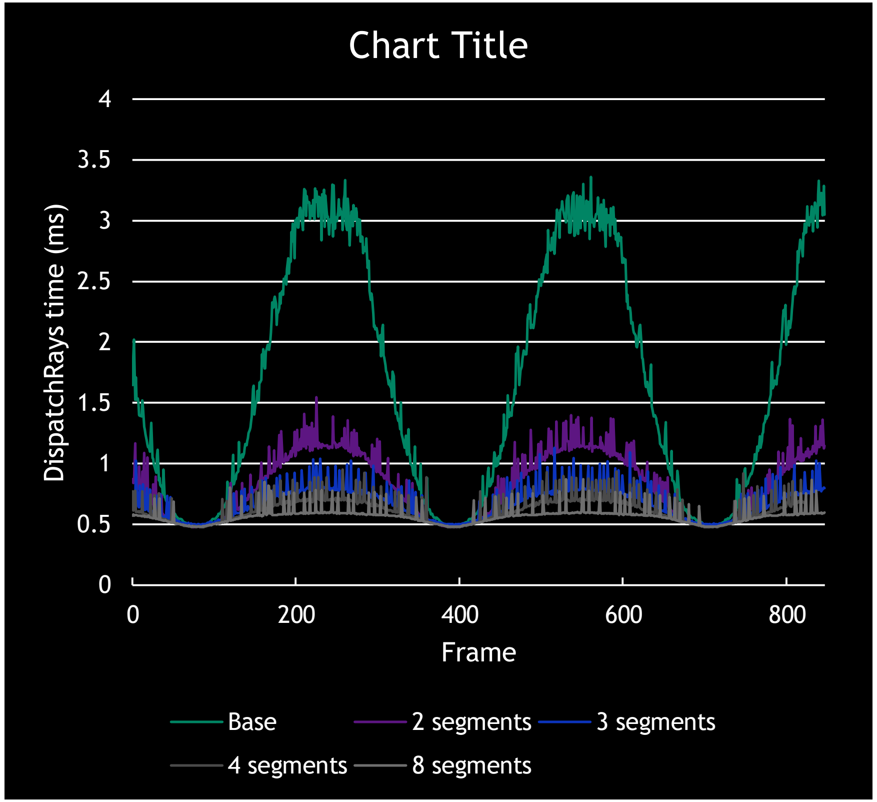
(Very rare corner case, but hit in one real game workload)



LONG, NARROW TRIANGLES

Mitigation via app side splitting





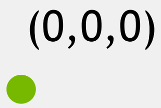
REFIT FROM DEGENERATES

Extra difficult case of heavily deformed refitting: build degenerate geometry with no spatial information whatsoever, then refit

Often shows up in game particle effects

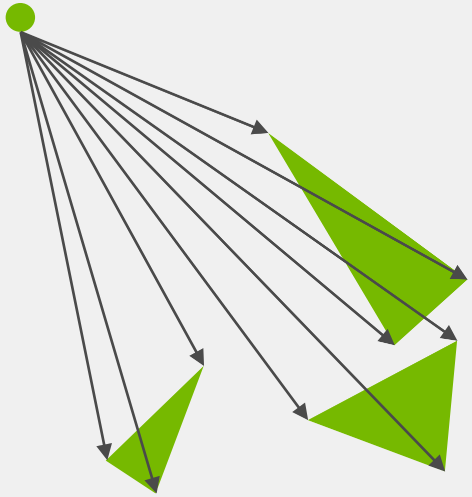
Works OK with ~tens of triangles

FRAME 0

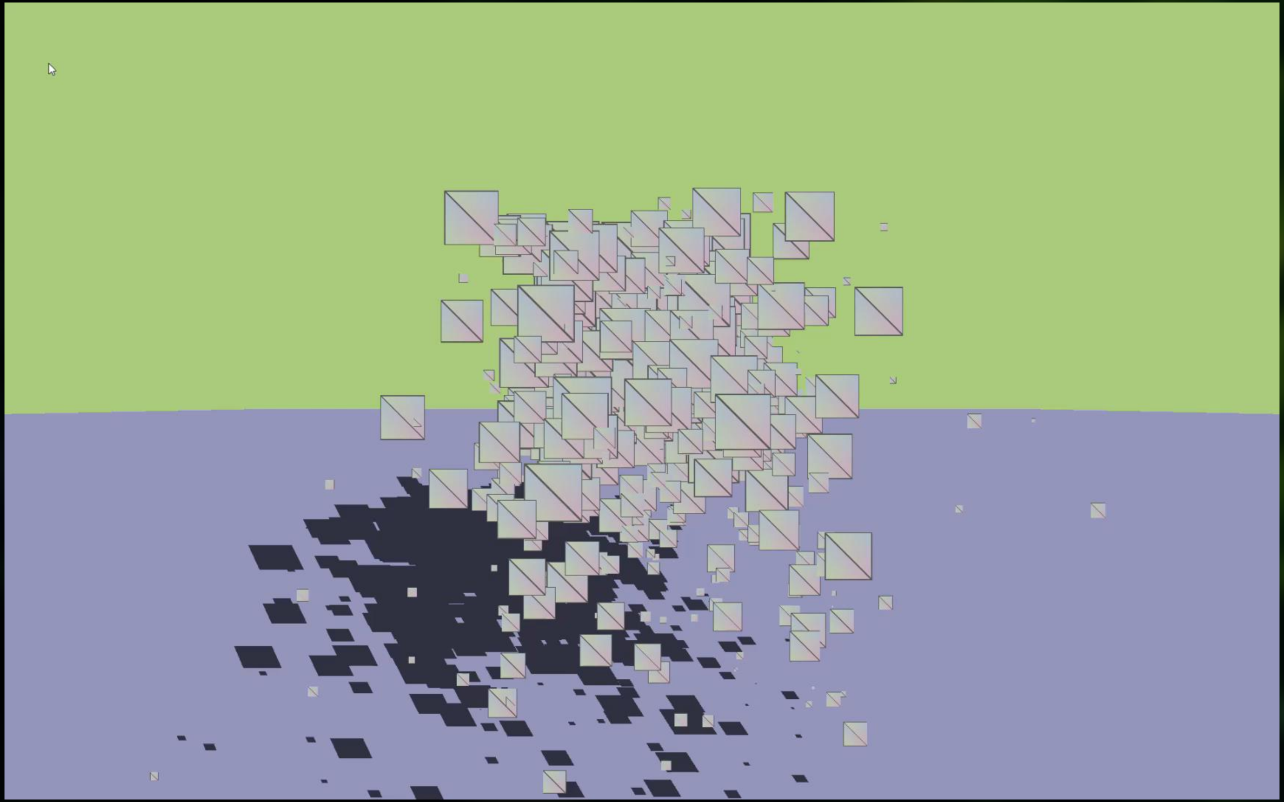


Build degenerate geometry

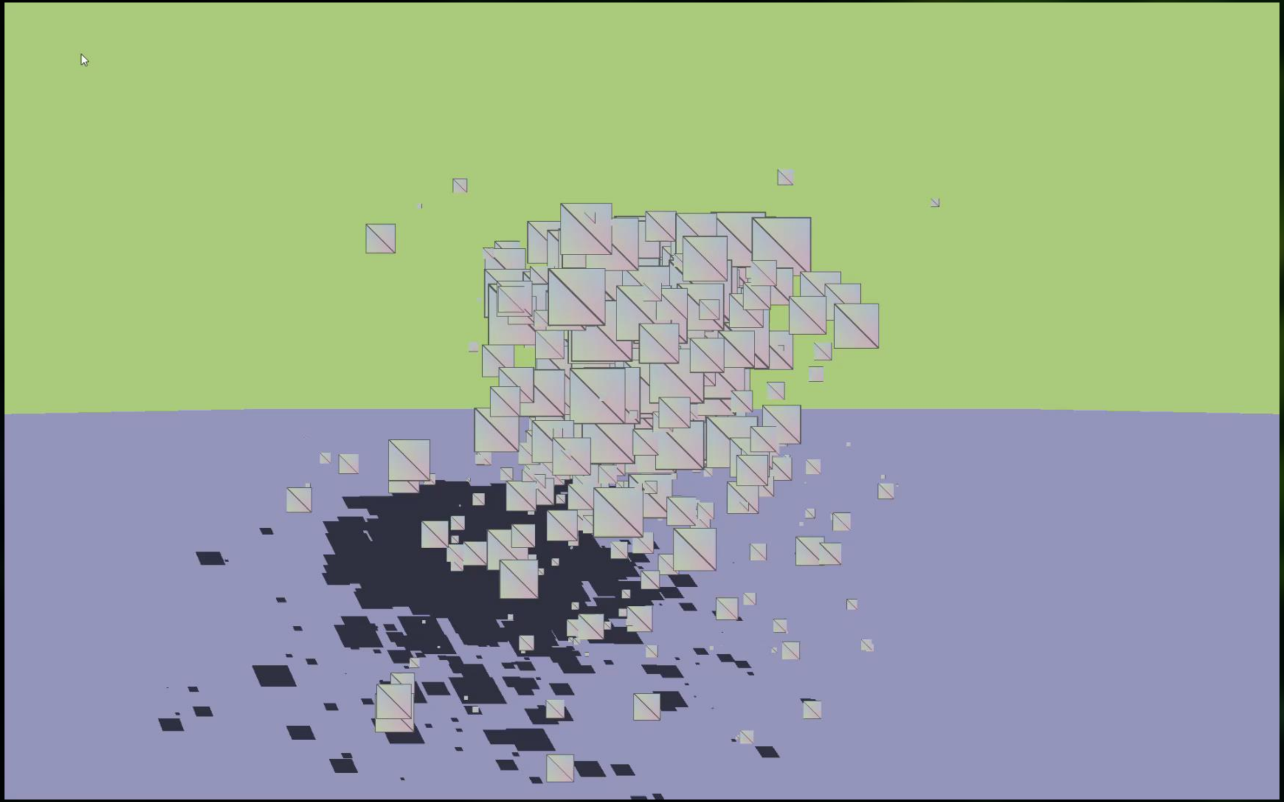
FRAME N



Refit into visible triangles



2000 PARTICLES, REBUILD EVERY FRAME, TRACERAYS() 0.5MS



2000 PARTICLES, REFIT EVERY FRAME, TRACERAYS() 0.5 → 8MS

RTX BVH SUMMARY

The RTX BVH builder is powering ray tracing in AAA games, and has been improving rapidly

- Builds are often almost free due to asynch overlapping

Limitations:

- Application side optimization needed; can’t rebuild all geometry every frame
- Some corner cases must be currently worked around application side
  - (Some might in the future be handled by the driver)

→Not done yet!

BVH CONSTRUCTION HARDWARE

MOTIVATION

Would always be nice to have much more raw build performance

Fixed-function accelerators can be 2—3 orders of magnitude faster (in perf per silicon area) and more energy-efficient than SW on a general-purpose processor (Hameed 2010)

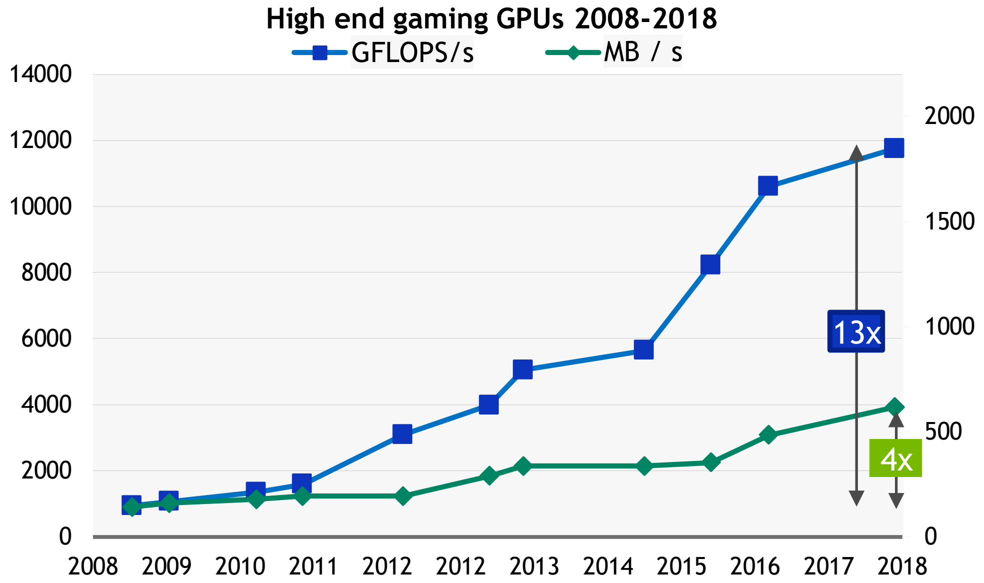
- ...But we are comparing against GPU SW and running a memory-intensive algorithm, so not going to get that much
- If HW accelerating a memory-intensive algorithm, might get more efficient on-chip computation but the same memory accesses → maybe no gains at all
- → All recent research on ray tracing HW revolves around optimizing DRAM traffic

DRAM ACCESS COST

Energy and bandwidth usage

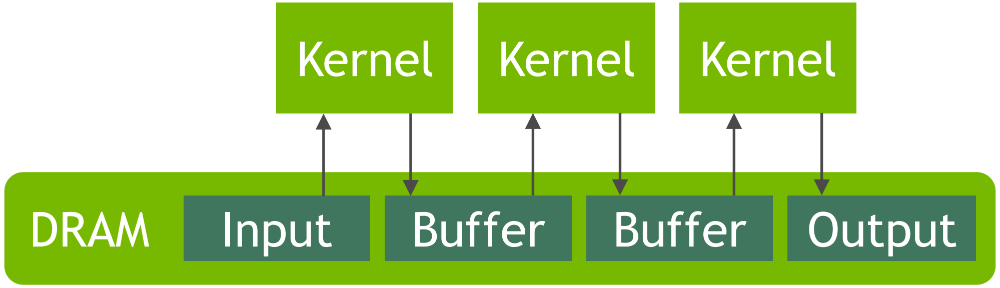
Operation	Energy
64-bit multiply-add	64 pJ
Read/store register data	6 pJ
Read 64 bits from DRAM	4200 pJ
Read 32 bits from DRAM	2100 pJ

S. Borkar, Intel, 32nm technology ca. 2010

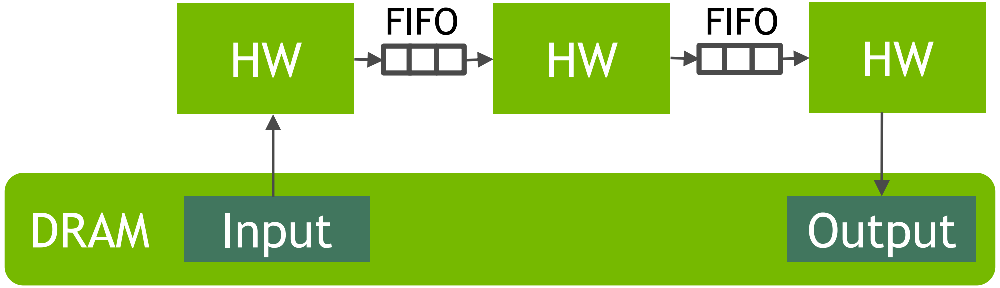


BANDWIDTH-SAVING HARDWARE DESIGN

A CUDA program often has multiple kernel launches which communicate through intermediate data buffers.



In HW, maybe the same algorithm can be expressed as serial HW pipelines communicating through on-chip FIFOs, saving DRAM traffic.



(Note: often the CUDA program can be improved in the same way)

TREE UPDATE HARDWARE

Small field, ~10 papers

- k-D tree builders (Nah, 2014; Liu, 2015)
- Refitter units (Nah, 2015; Woop, 2006)
- Imagination Technologies SHG (McCombe 2014)
- **Binned SAH sweep unit (Doyle, 2013)**
- **MergeTree (Viitanen, 2015)**
- **PLOCTree (Viitanen, 2018)**

TREE UPDATE HARDWARE

Small field, ~10 papers

- k-D tree builders (Nah, 2014; Liu, 2015)
- Refitter units (Nah, 2015; Woop, 2006)
- Imagination Technologies SHG (McCombe 2014)
- **Binned SAH sweep unit (Doyle, 2013)**
- **MergeTree (Viitanen, 2015)**
- **PLOCTree (Viitanen, 2018)**

k-D tree builds are too expensive

Refitters are interesting, but not described in much detail – parts of larger RT systems

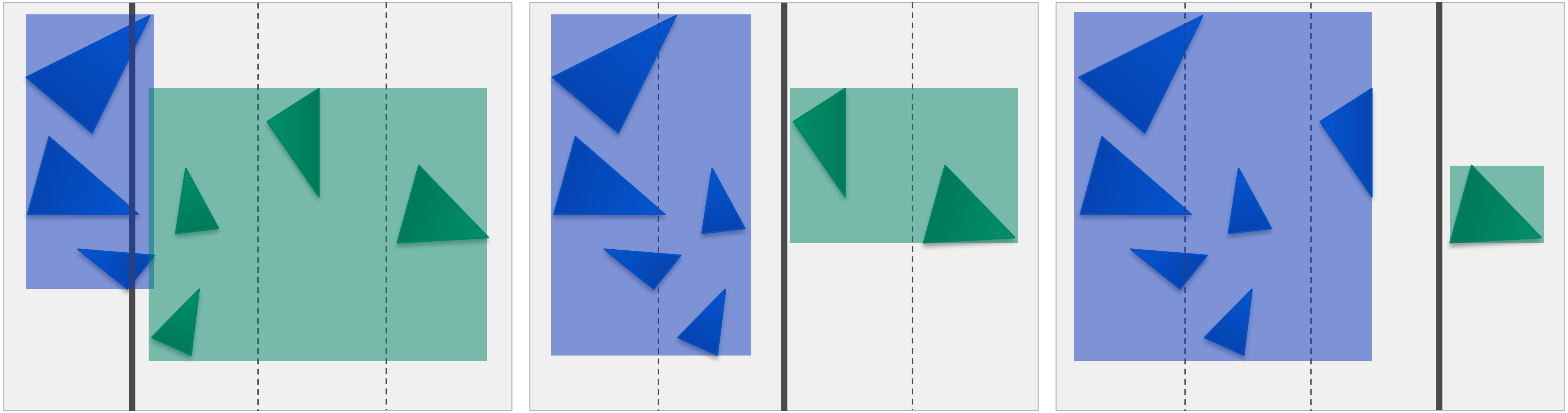
Very interesting and exotic architecture by a GPU vendor – but not much information out

We'll look at these

HARDWARE TREE BUILDERS: BINNED SAH (DOYLE, 2013)

BINNED SAH SWEEP

(4 bins)



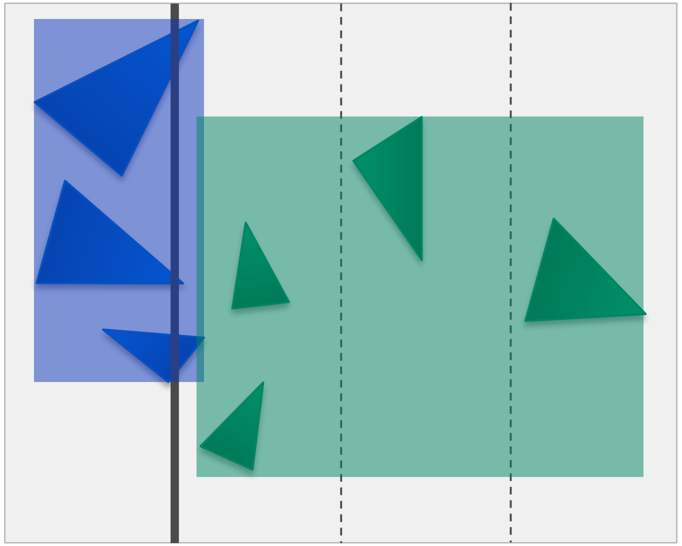
Split candidate 1

Split candidate 2

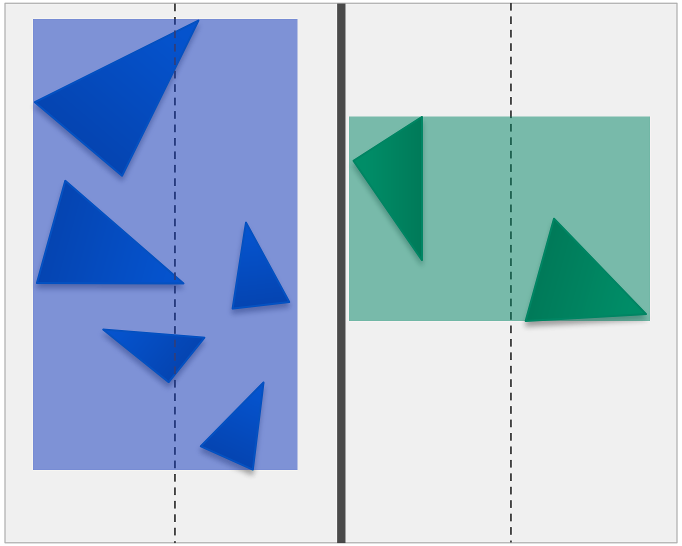
Split candidate 3

BINNED SAH SWEEP

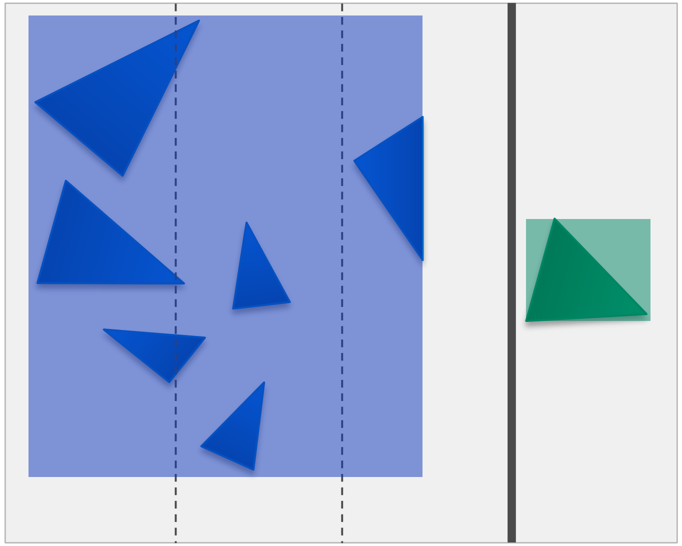
(4 bins)



Split candidate 1



Split candidate 2

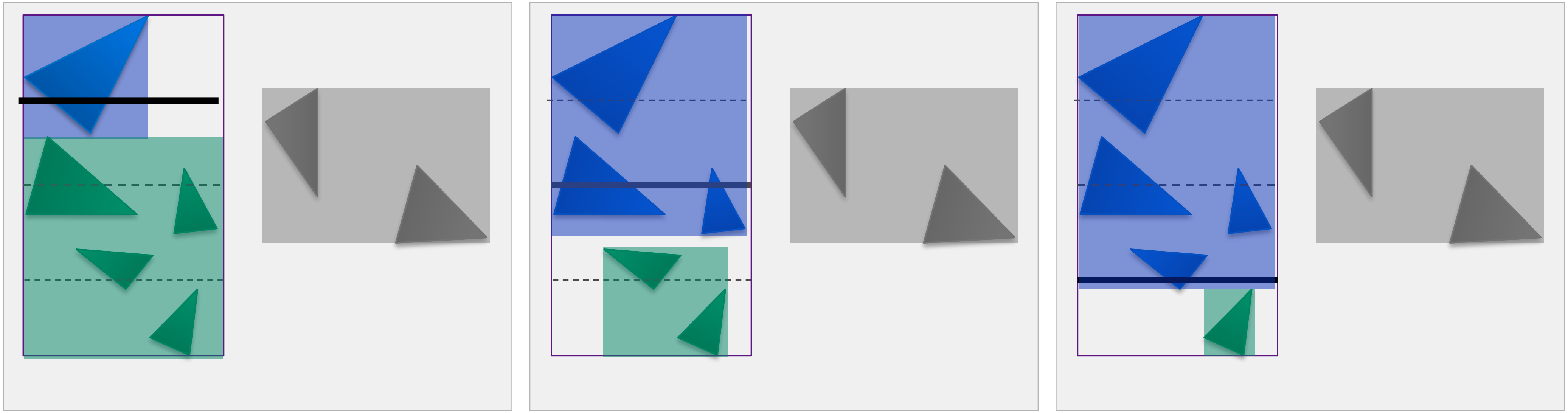


Split candidate 3

Best split!

BINNED SAH SWEEP

(4 bins)



Split candidate 1

Split candidate 2

Split candidate 3

BINNED SAH HW

(Doyle, 2013)

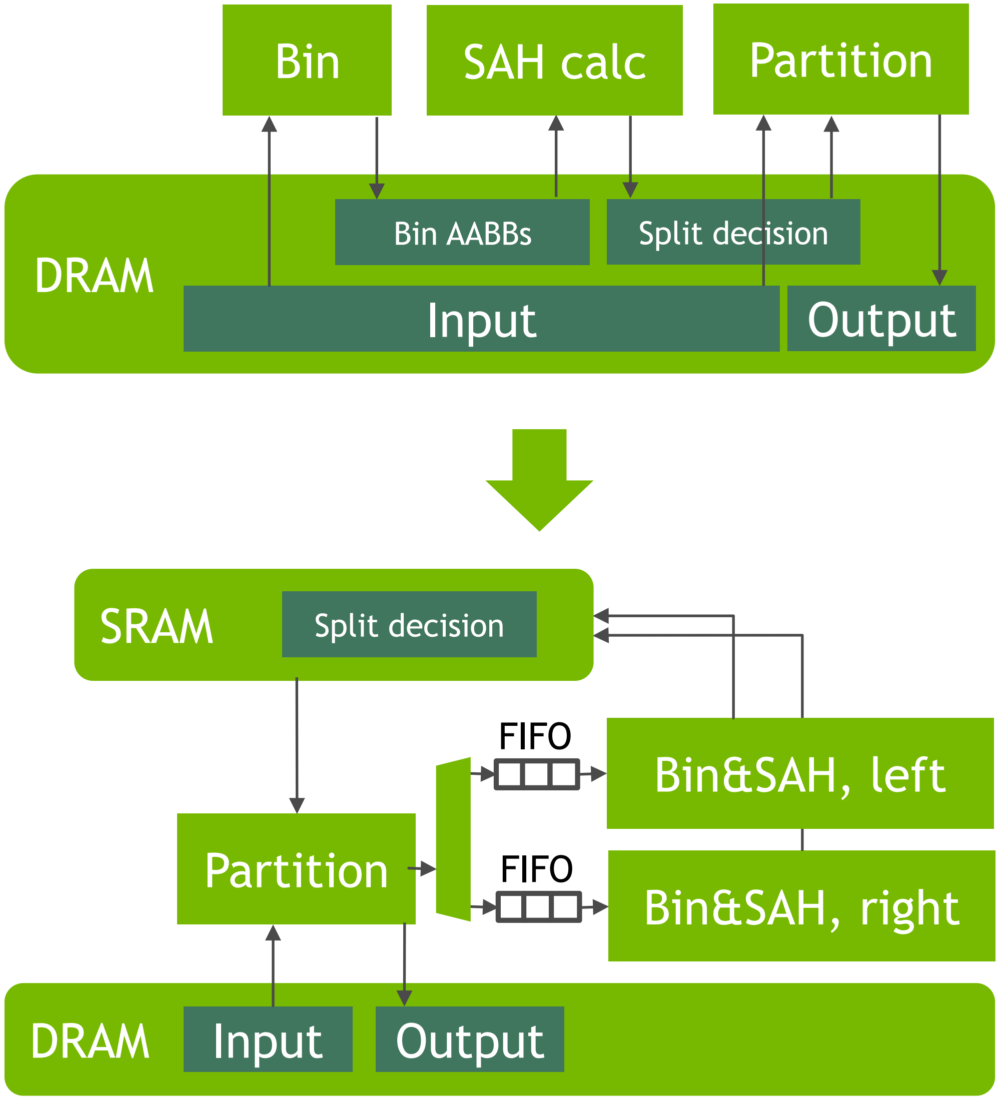
Memory optimizations:

Pipeline partitioning with binning and SAH computation for the **child partitions** (one pass over input data instead of two)

When partition size drops small enough, handle it completely in on-chip memory

Mem traffic 2-3x less than HLBVH, a cheaper algorithm; far faster than GPU binned SAH

Downside: expensive, many FPUs



HARDWARE TREE BUILDERS: MERGETREE (VIITANEN, 2015)

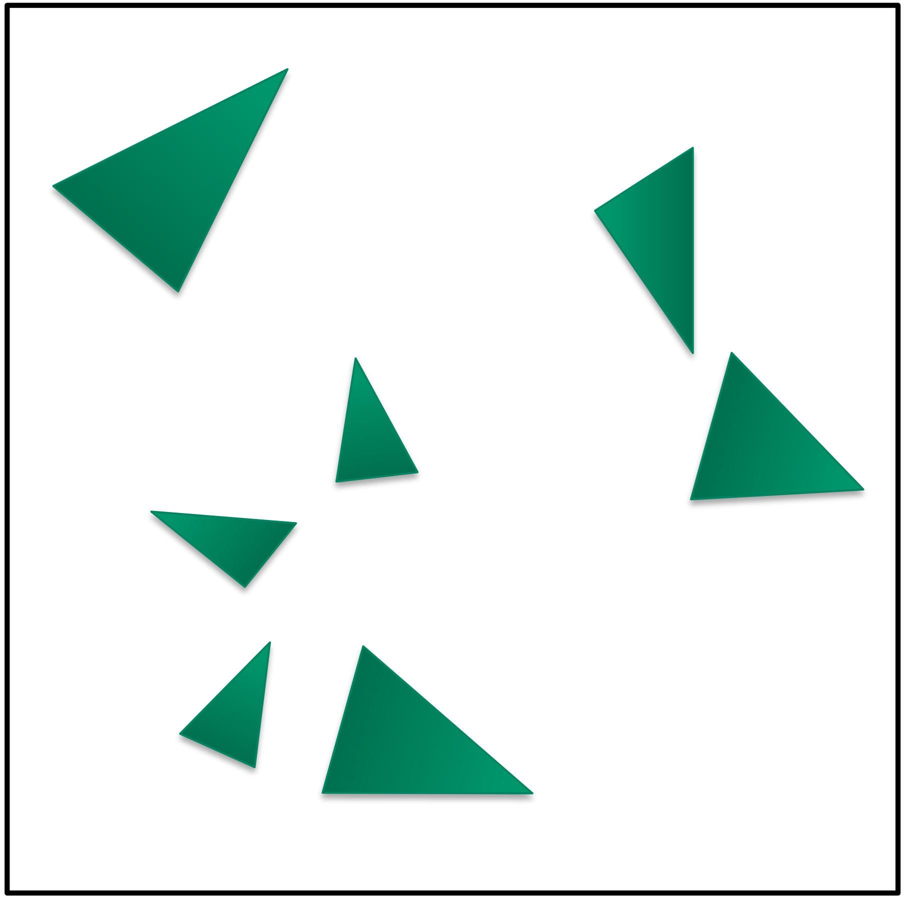
LBVH ALGORITHM (LAUTERBACH ET AL. 2009)

Morton code computation

Sorting

Hierarchy emission

AABB computation



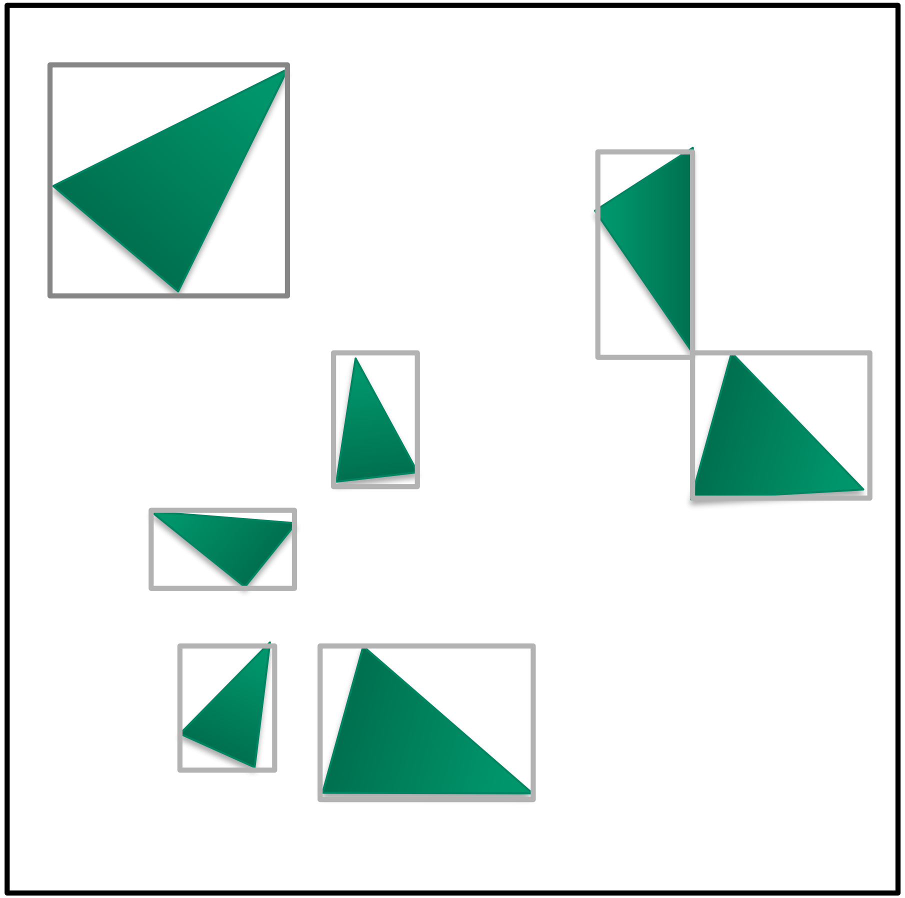
LBVH ALGORITHM (LAUTERBACH ET AL. 2009)

Morton code computation

Sorting

Hierarchy emission

AABB computation



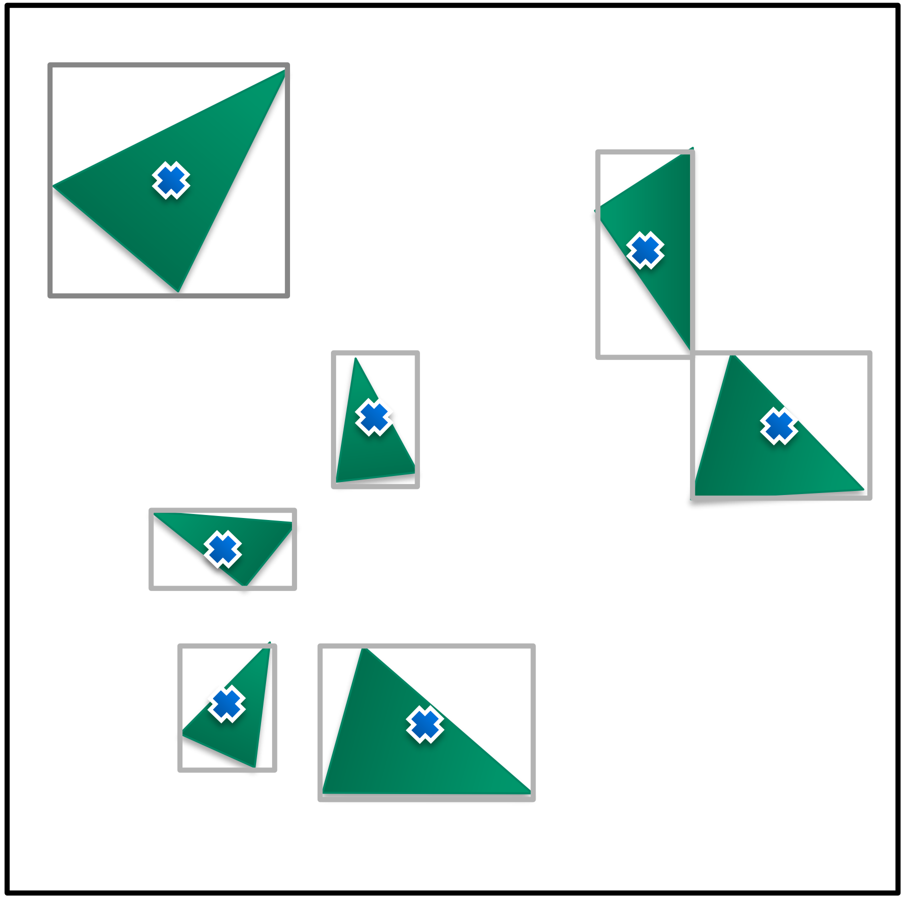
LBVH ALGORITHM (LAUTERBACH ET AL. 2009)

Morton code computation

Sorting

Hierarchy emission

AABB computation



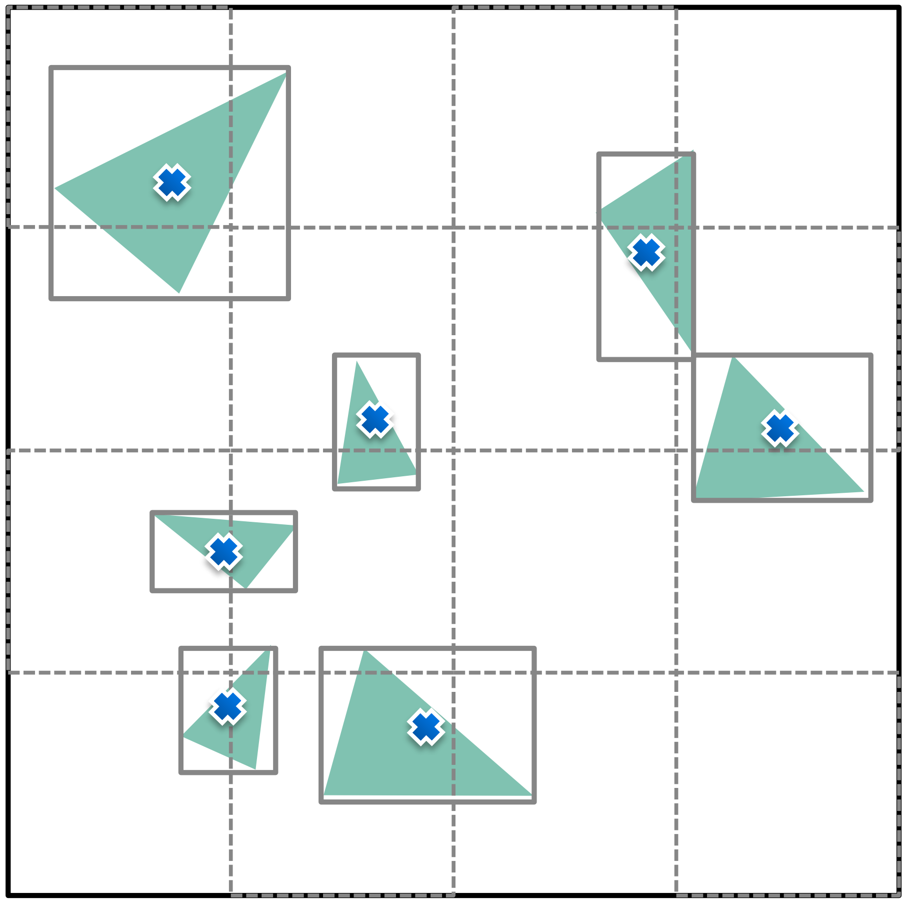
LBVH ALGORITHM (LAUTERBACH ET AL. 2009)

Morton code computation

Sorting

Hierarchy emission

AABB computation



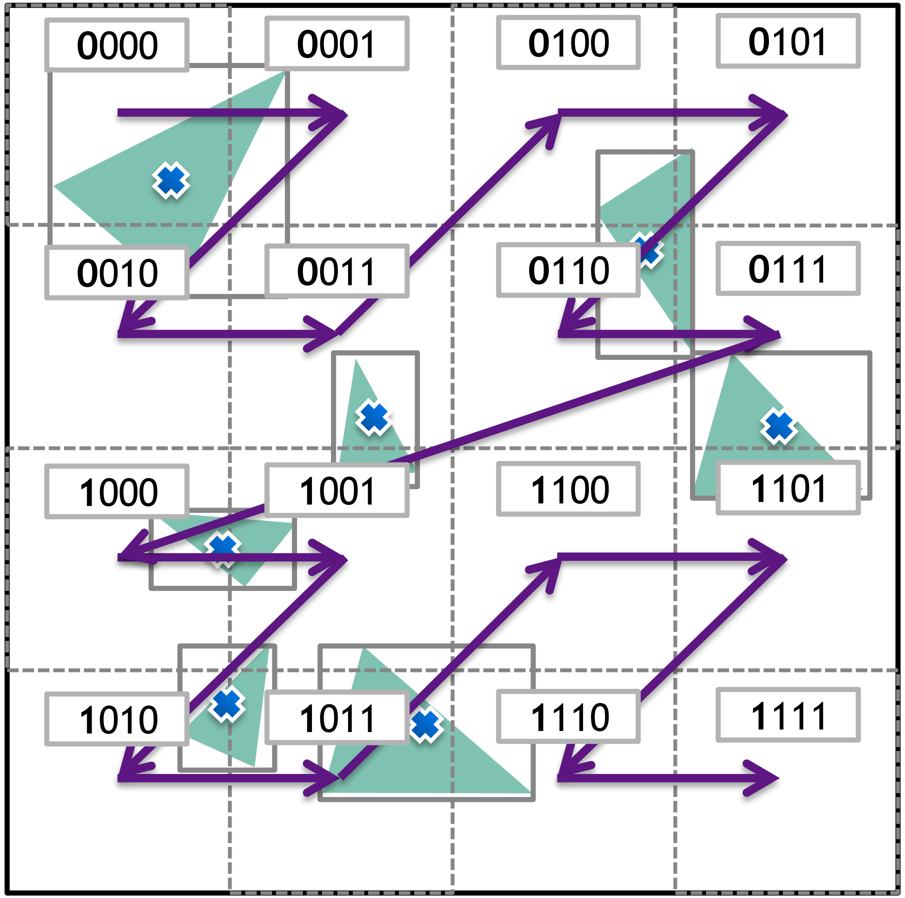
LBVH ALGORITHM (LAUTERBACH ET AL. 2009)

Morton code computation

Sorting

Hierarchy emission

AABB computation



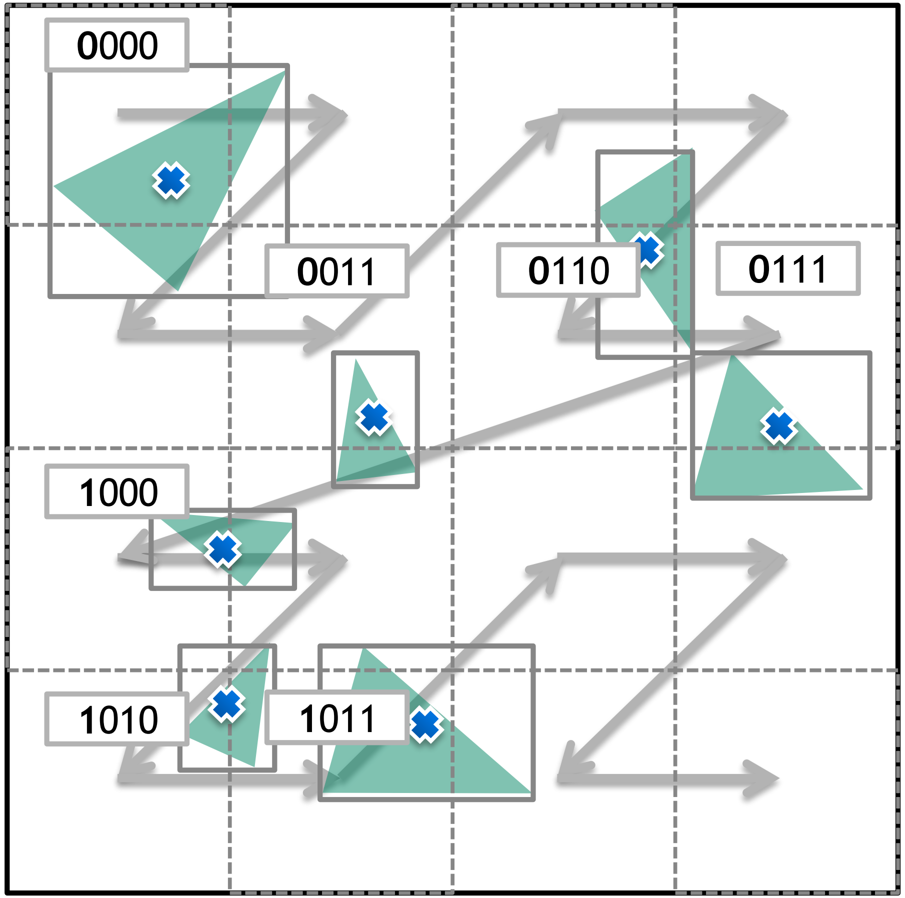
LBVH ALGORITHM (LAUTERBACH ET AL. 2009)

Morton code computation

Sorting

Hierarchy emission

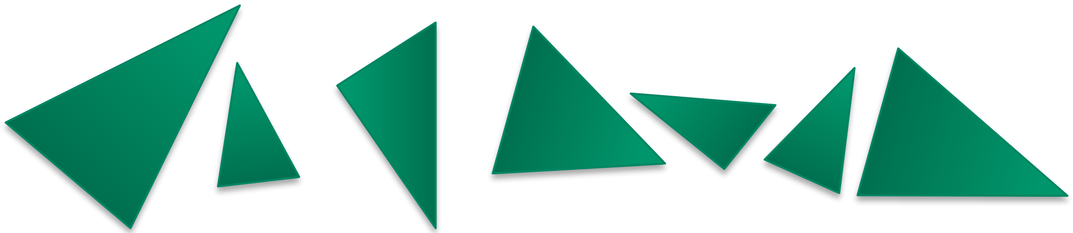
AABB computation

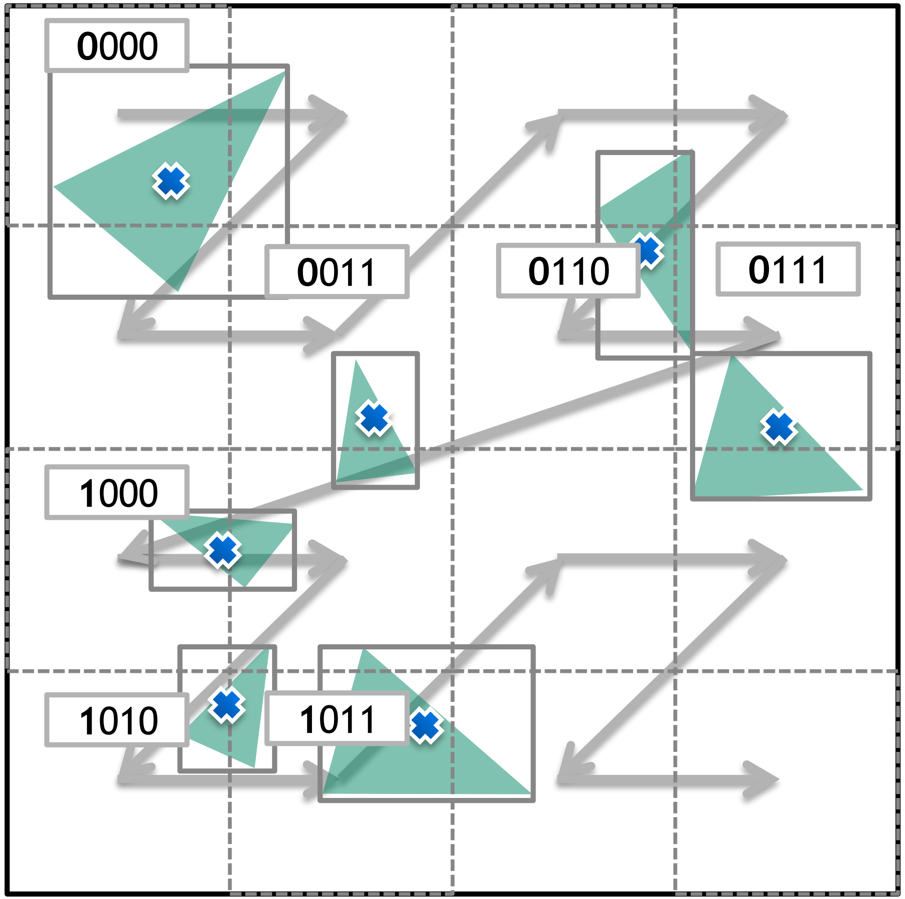


LBVH ALGORITHM (LAUTERBACH ET AL. 2009)

Hierarchy emission

0	0	0	0	1	1	1
0	0	1	1	0	0	0
0	1	1	1	0	1	1
0	1	0	1	0	0	1

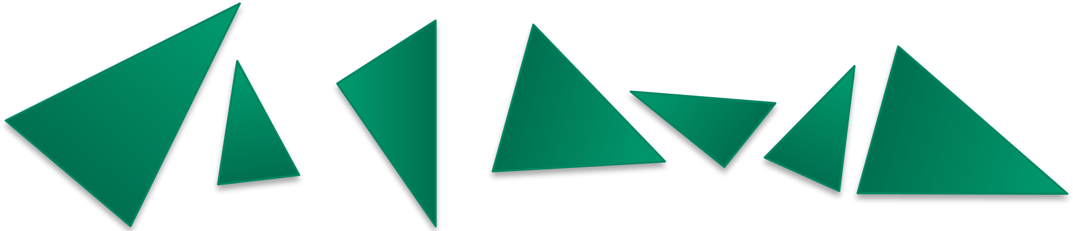


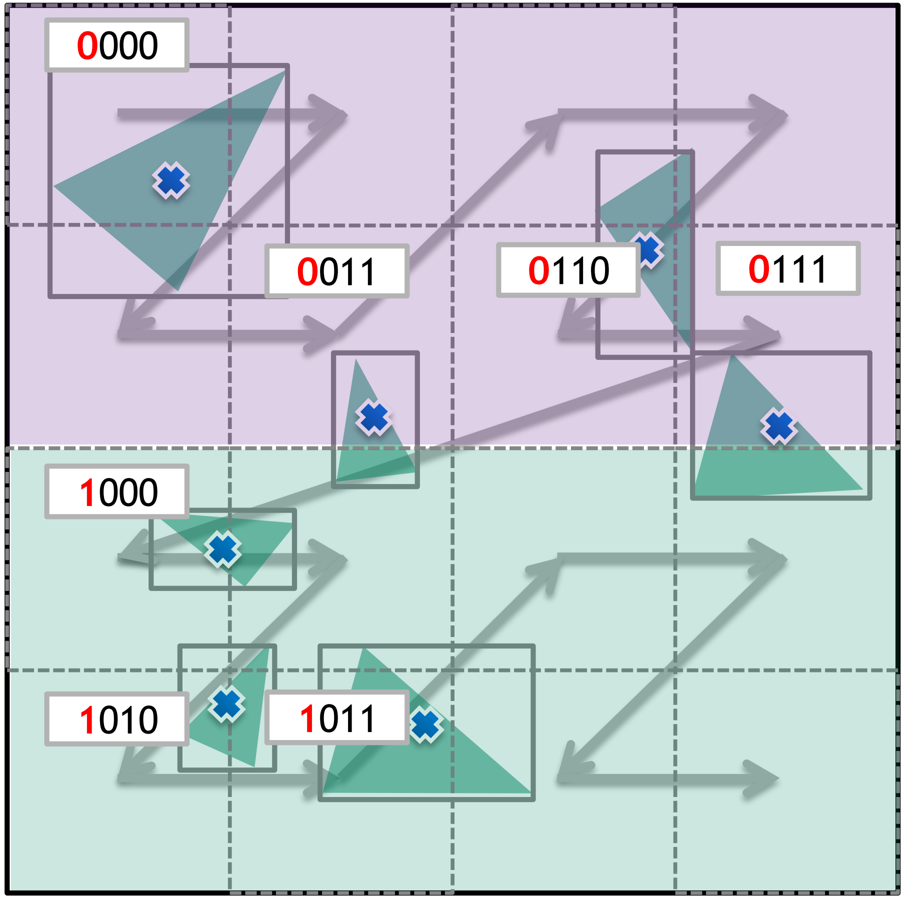


LBVH ALGORITHM (LAUTERBACH ET AL. 2009)

Hierarchy emission

0	0	0	0	1	1	1
0	0	1	1	0	0	0
0	1	1	1	0	1	1
0	1	0	1	0	0	1

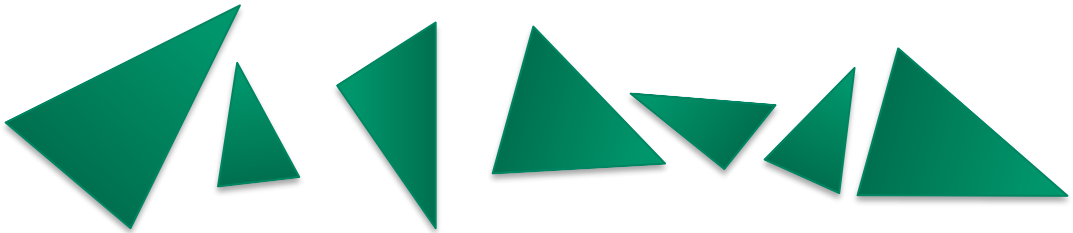


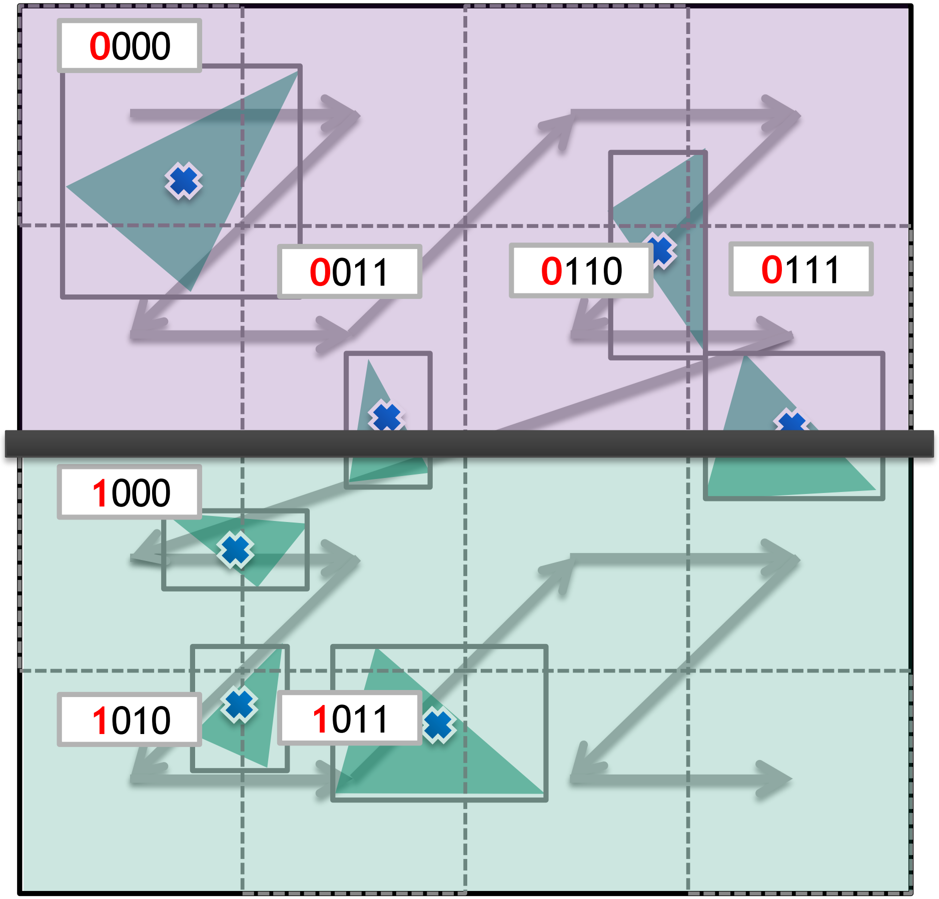


LBVH ALGORITHM (LAUTERBACH ET AL. 2009)

Hierarchy emission

0	0	0	0 ■	1	1	1
0	0	1	1	0	0	0
0	1	1	1	0	1	1
0	1	0	1	0	0	1

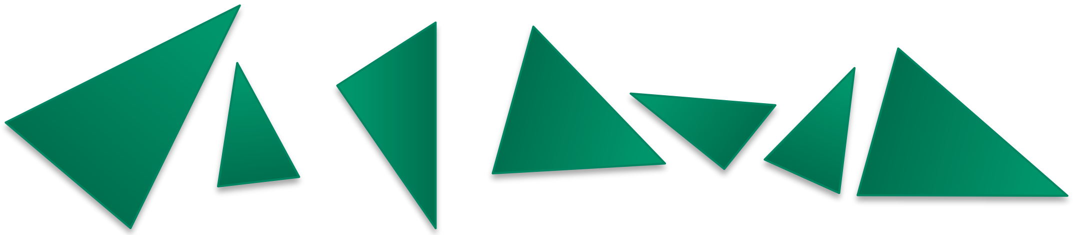


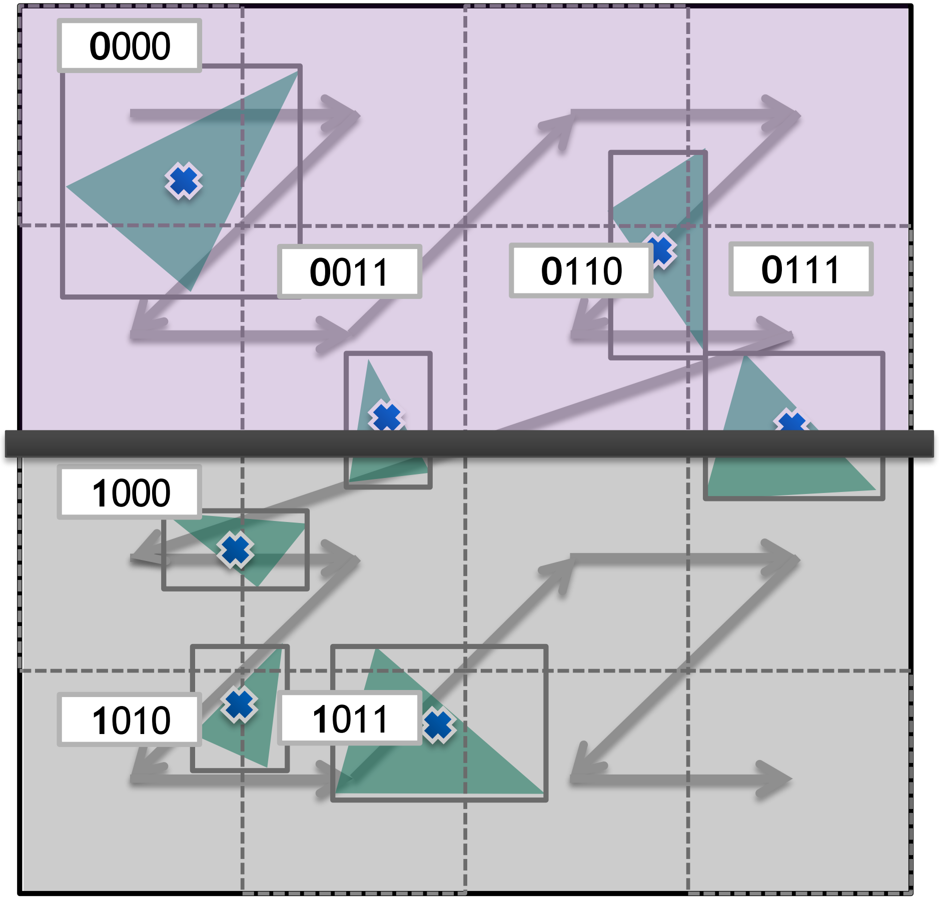


LBVH ALGORITHM (LAUTERBACH ET AL. 2009)

Hierarchy emission

0	0	0	0 ■	1	1	1
0	**0**	**1**	**1**	0	0	0
0	1	1	1	0	1	1
0	1	0	1	0	0	1

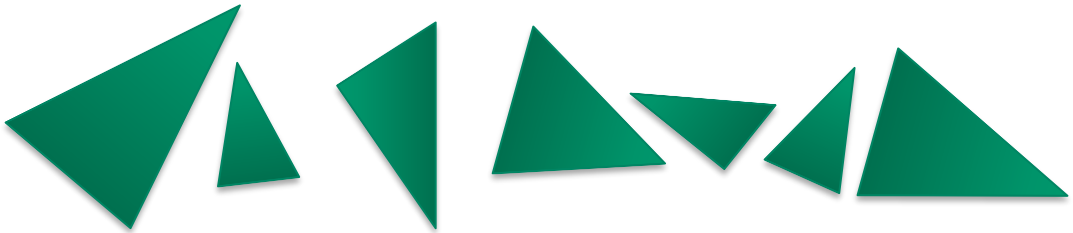


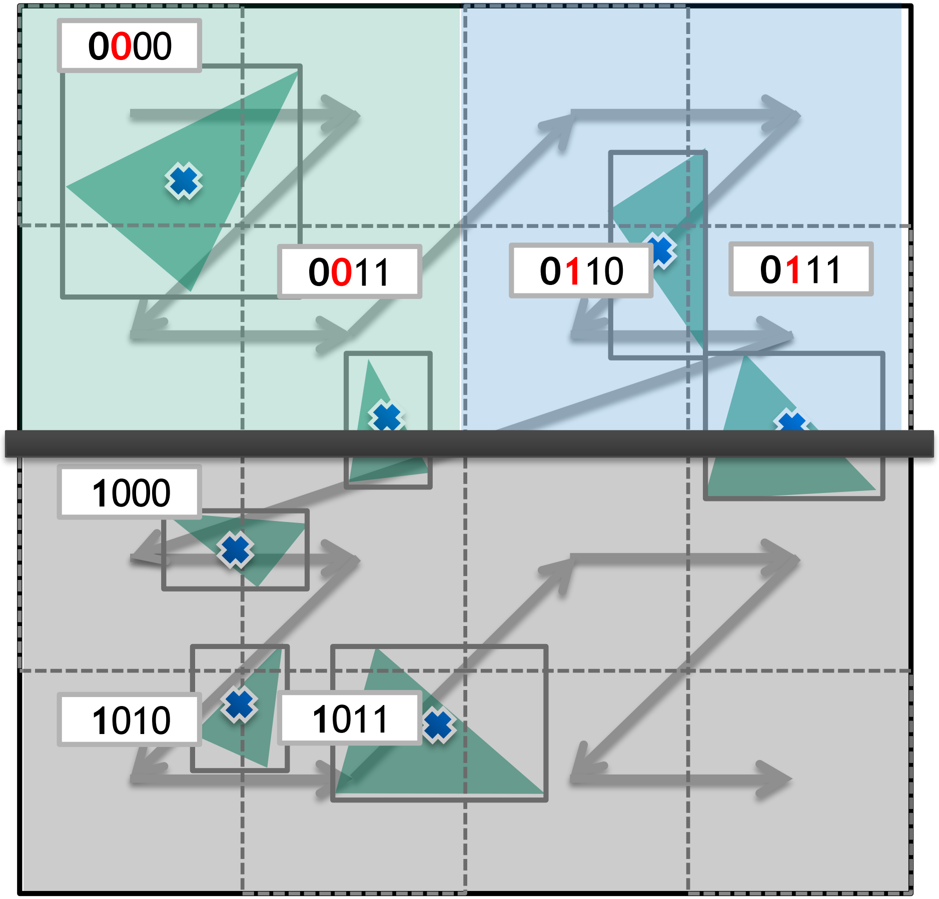


LBVH ALGORITHM (LAUTERBACH ET AL. 2009)

Hierarchy emission

0	0	0	0	1	1	1
0	0	1	1	0	0	0
0	1	1	1	0	1	1
0	1	0	1	0	0	1

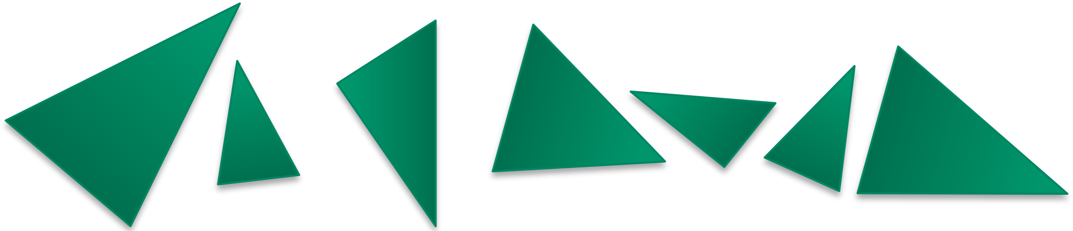


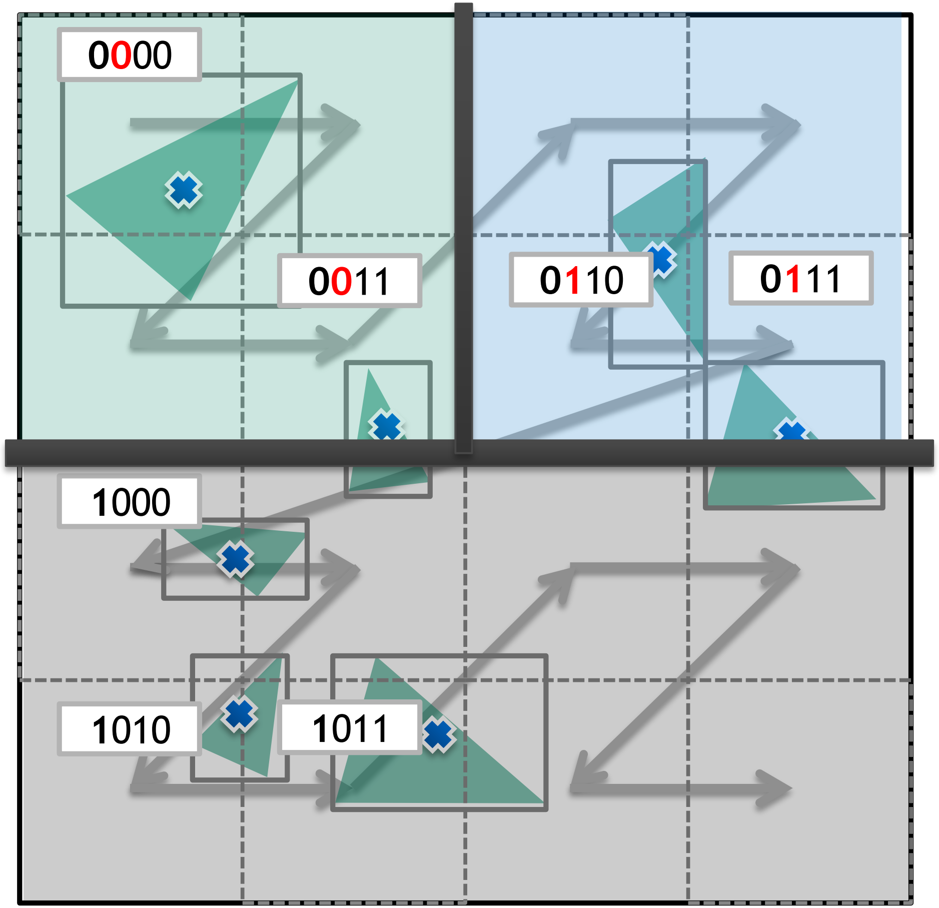


LBVH ALGORITHM (LAUTERBACH ET AL. 2009)

Hierarchy emission

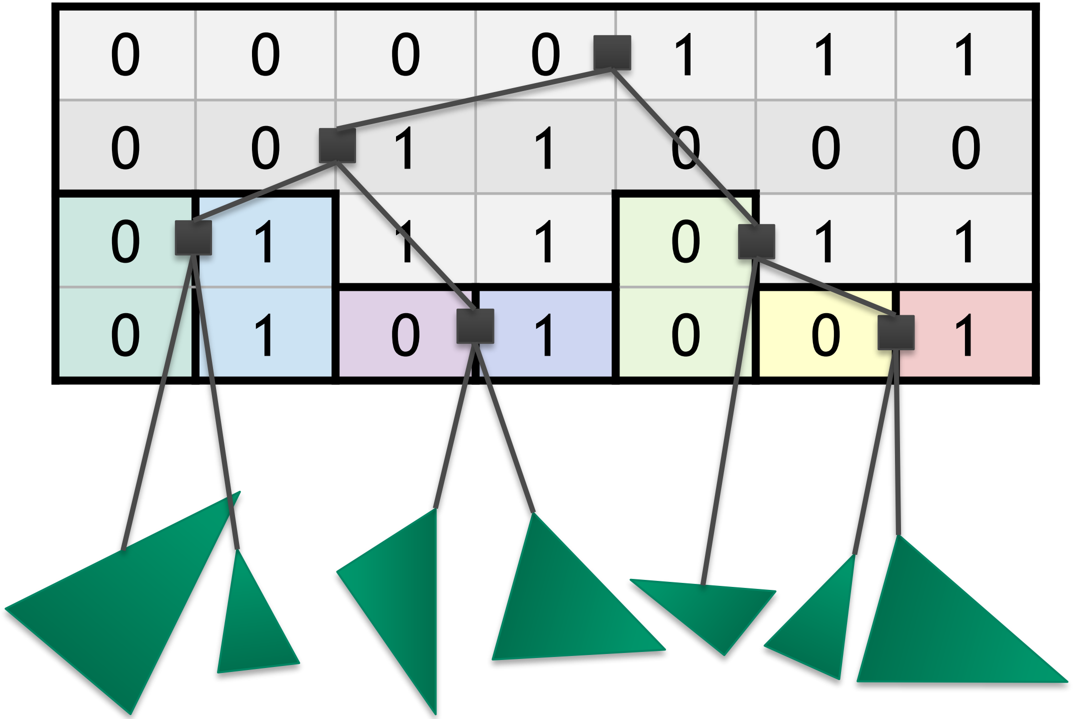
0	0	0	0	1	1	1
0	0	1	1	0	0	0
0	1	1	1	0	1	1
0	1	0	1	0	0	1



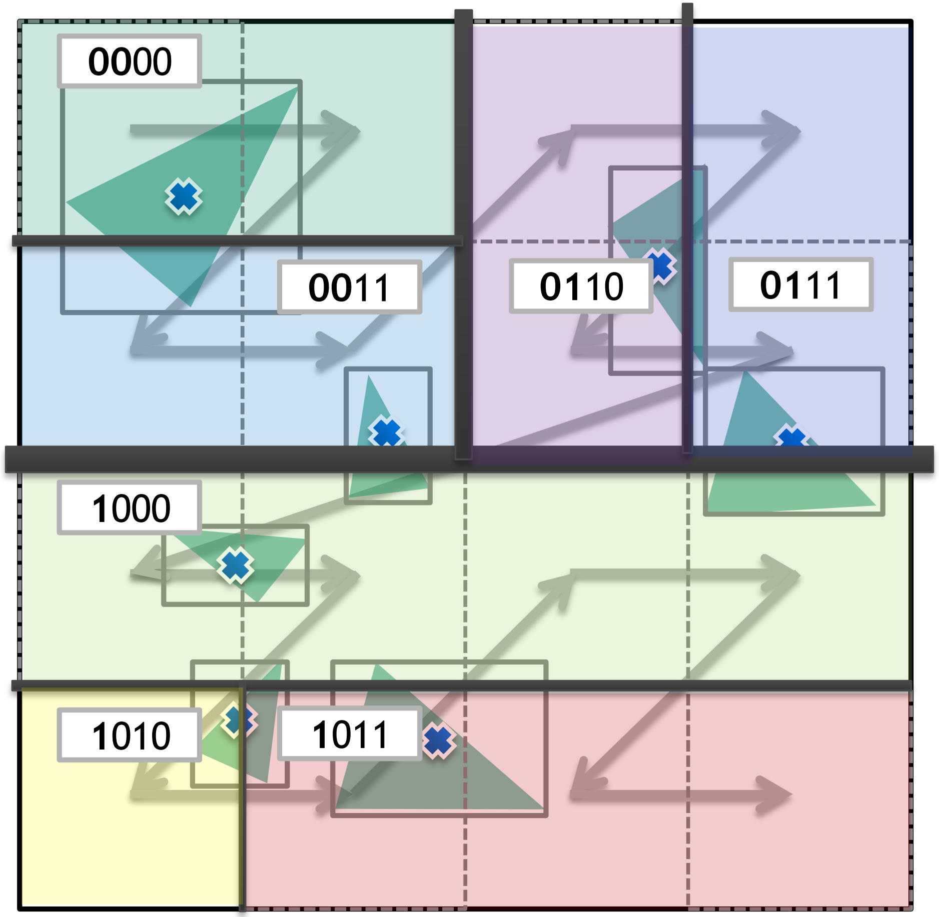


LBVH ALGORITHM (LAUTERBACH ET AL. 2009)

Hierarchy emission

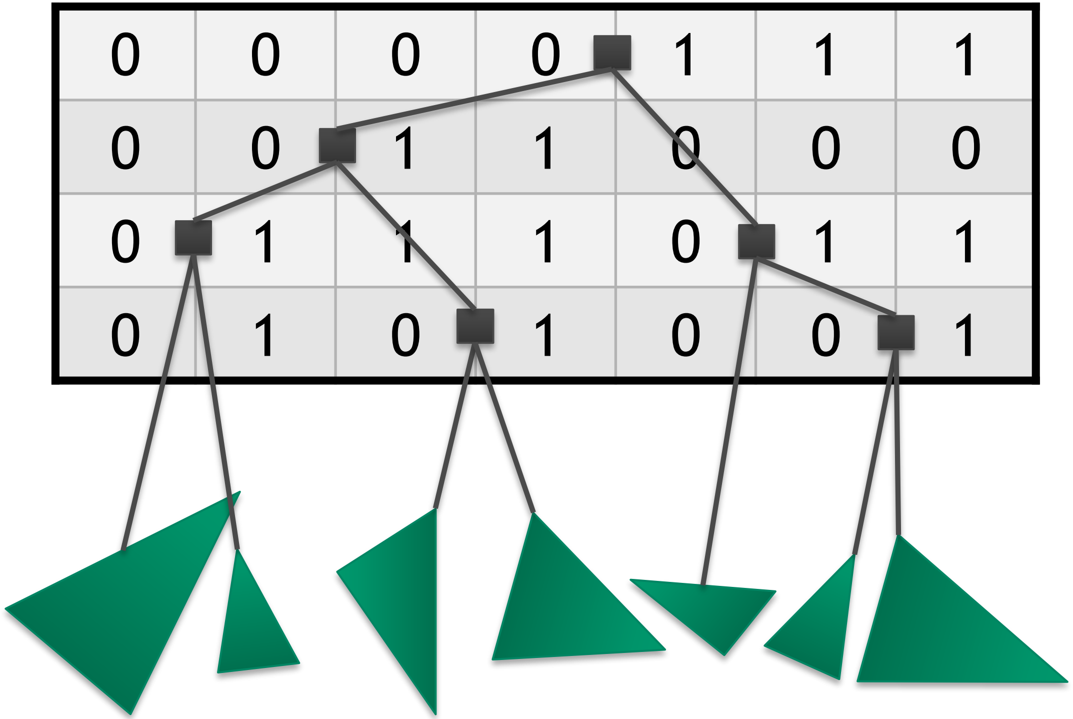


0	0	0	0	1	1	1
0	0	1	1	0	0	0
0	1	1	1	0	1	1
0	1	0	1	0	0	1

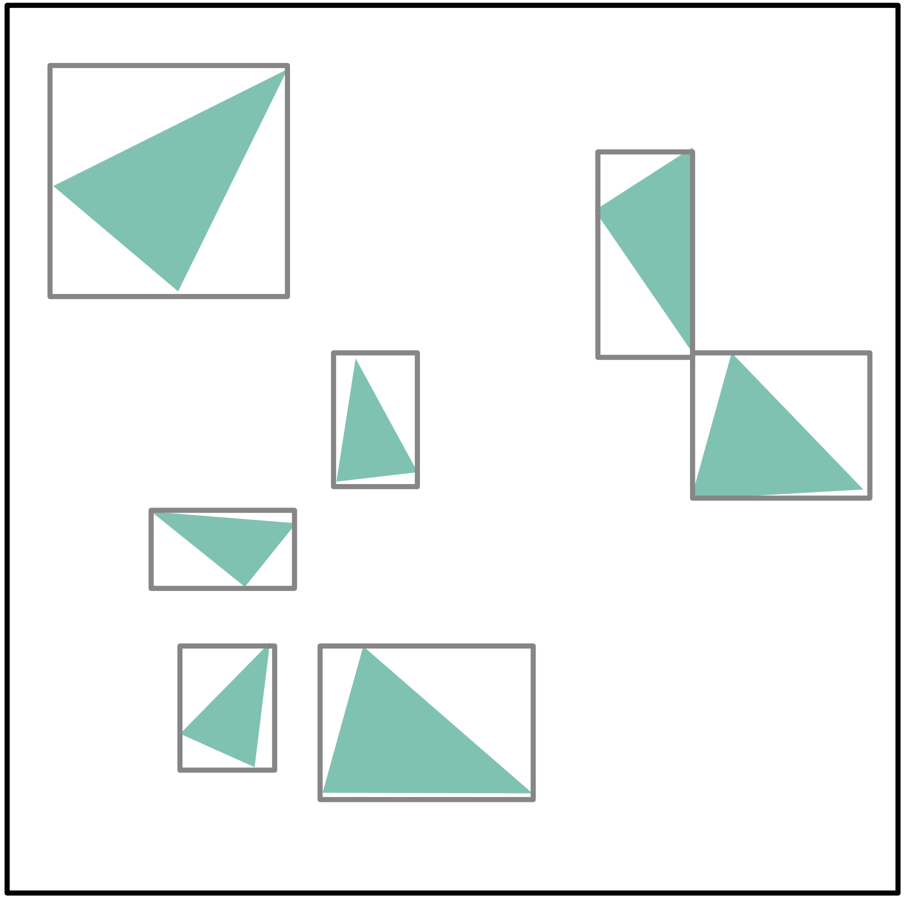


LBVH ALGORITHM (LAUTERBACH ET AL. 2009)

AABB computation

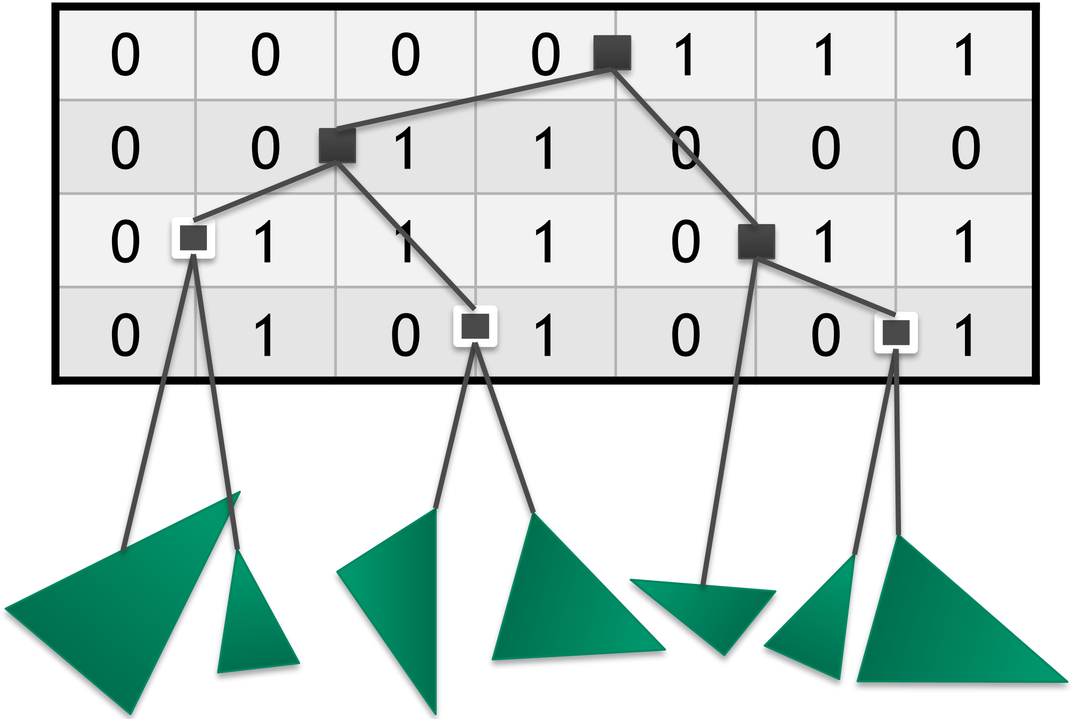


0	0	0	0	1	1	1
0	0	1	1	0	0	0
0	1	1	1	0	1	1
0	1	0	1	0	0	1

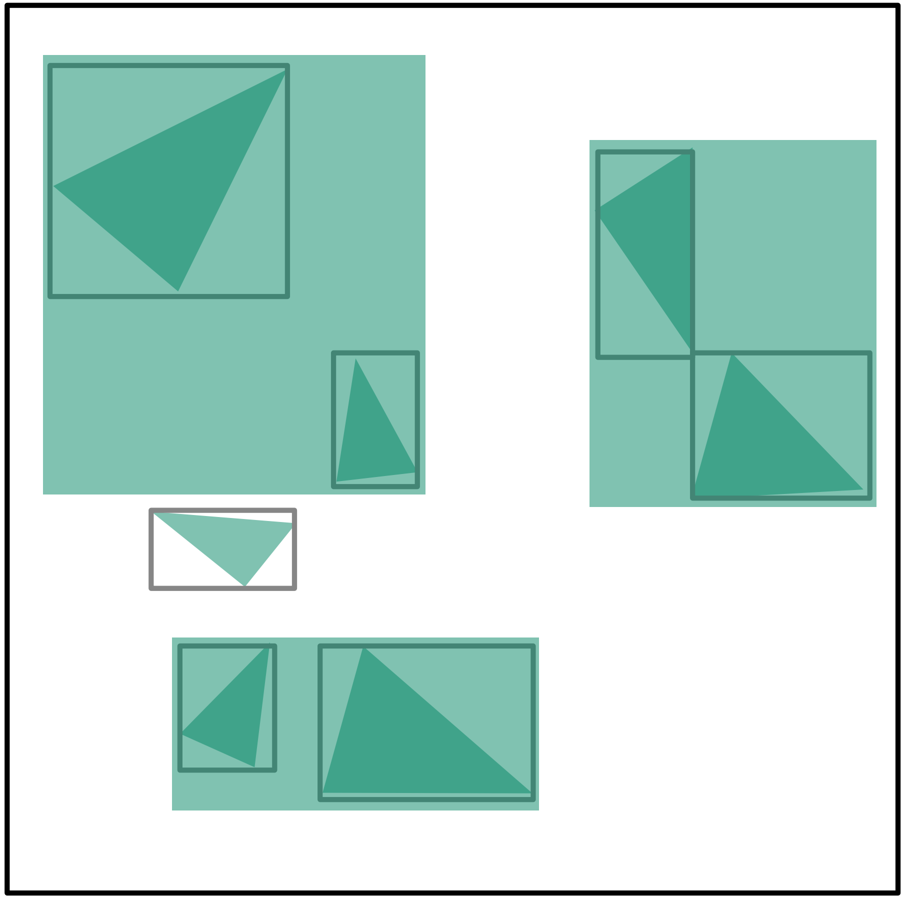


LBVH ALGORITHM (LAUTERBACH ET AL. 2009)

AABB computation

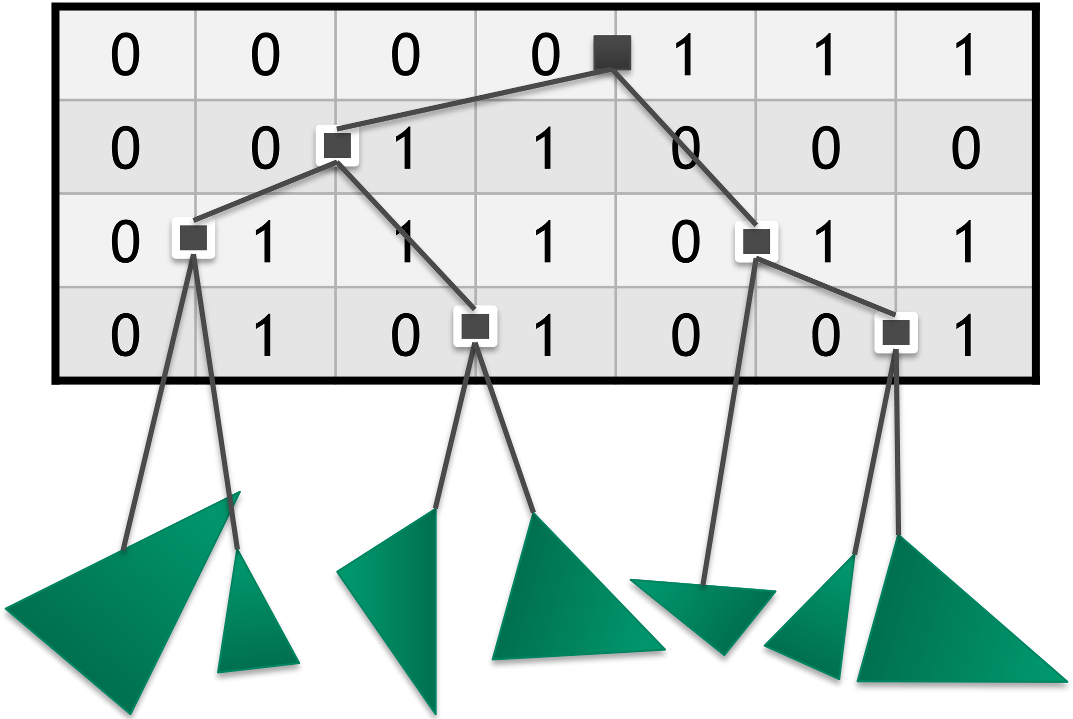


0	0	0	0	1	1	1
0	0	1	1	0	0	0
0	1	1	1	0	1	1
0	1	0	1	0	0	1

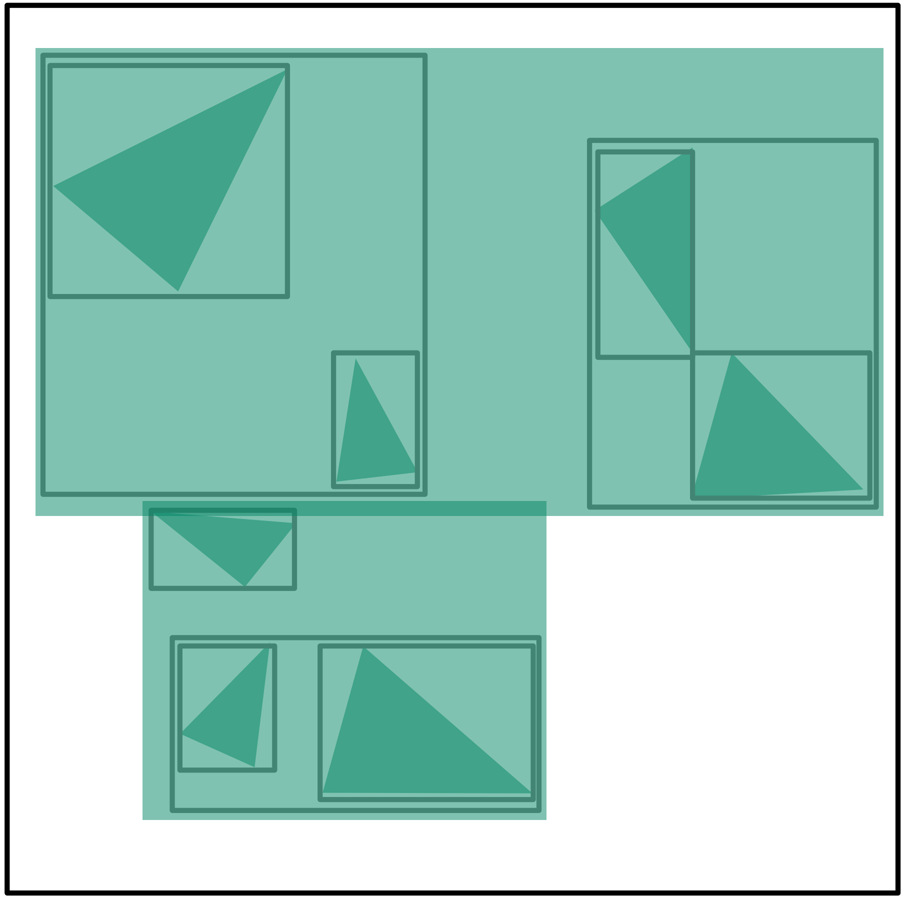


LBVH ALGORITHM (LAUTERBACH ET AL. 2009)

AABB computation

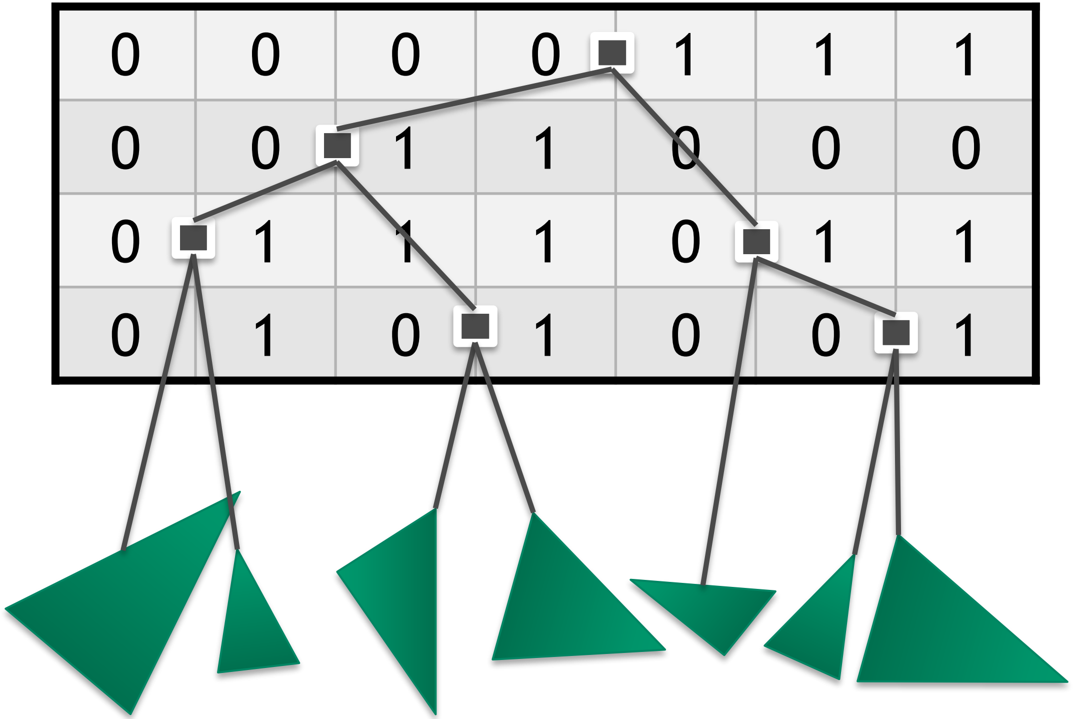


0	0	0	0	1	1	1
0	0	1	1	0	0	0
0	1	1	1	0	1	1
0	1	0	1	0	0	1

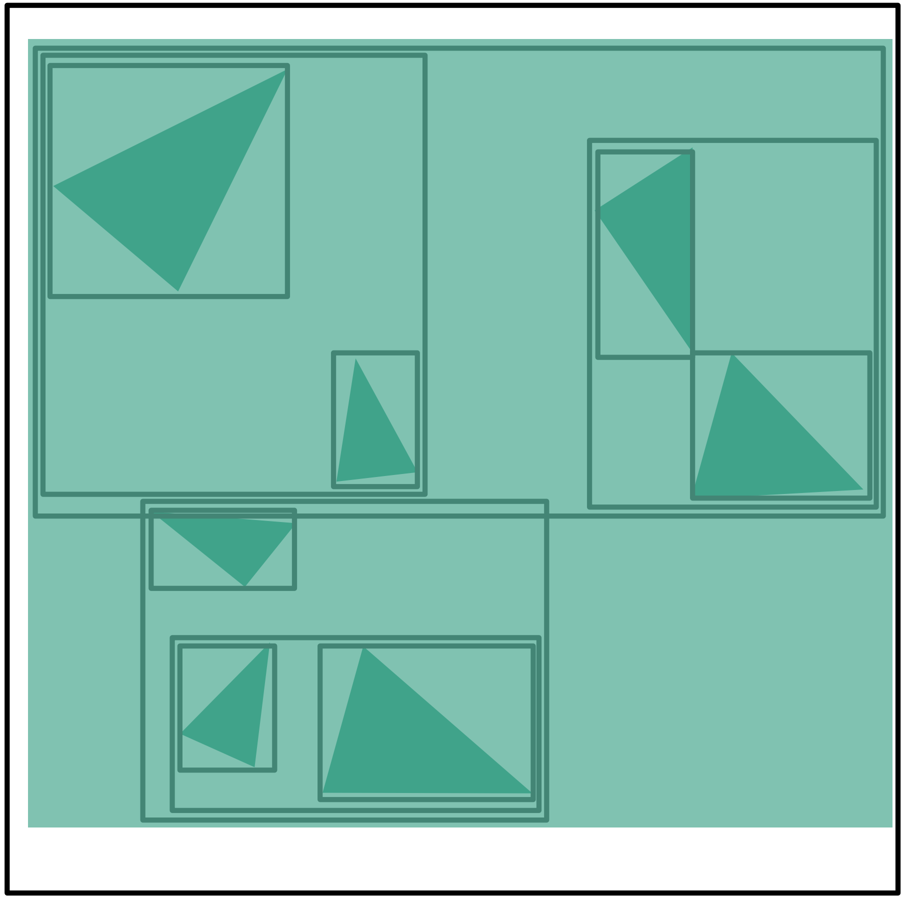


LBVH ALGORITHM (LAUTERBACH ET AL. 2009)

AABB computation



0	0	0	0	1	1	1
0	0	1	1	0	0	0
0	1	1	1	0	1	1
0	1	0	1	0	0	1

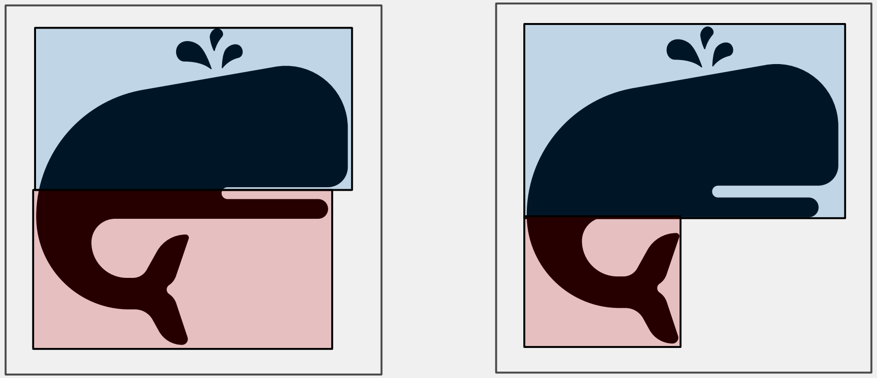


LBVH

Some reasons LBVH has low quality

FIXED SPLIT LOCATIONS

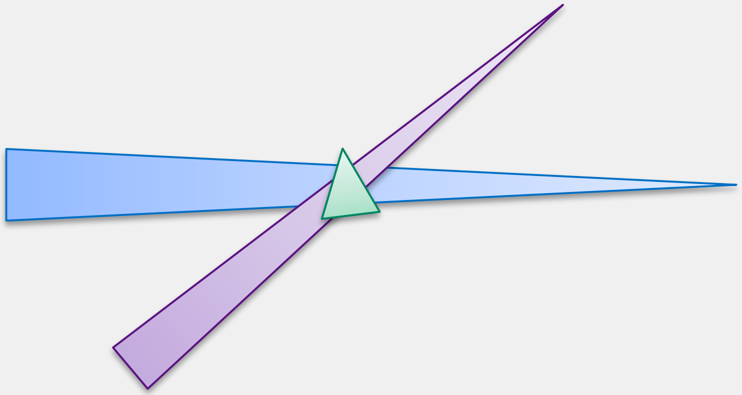
A halfway split along a predetermined axis is probably not the best one



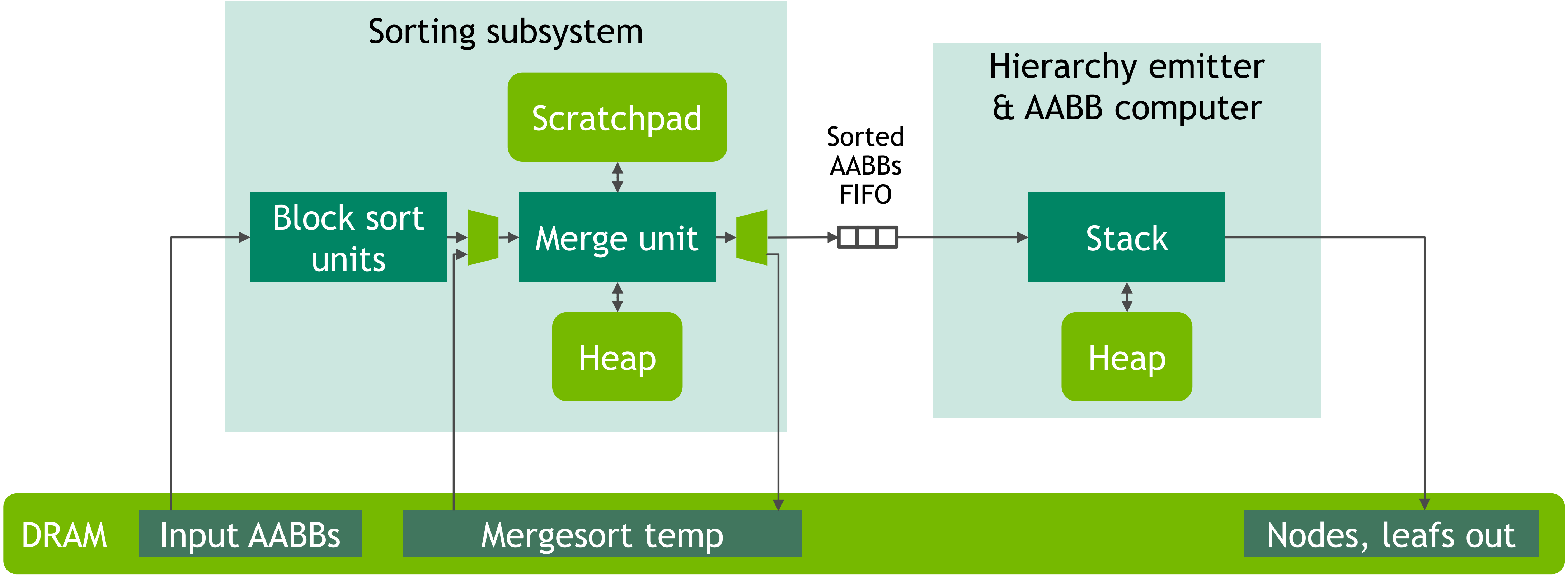
SCALE INSENSITIVITY

LBVH only looks at triangle centroids and has no idea of their shape: the triangles below are treated as identical

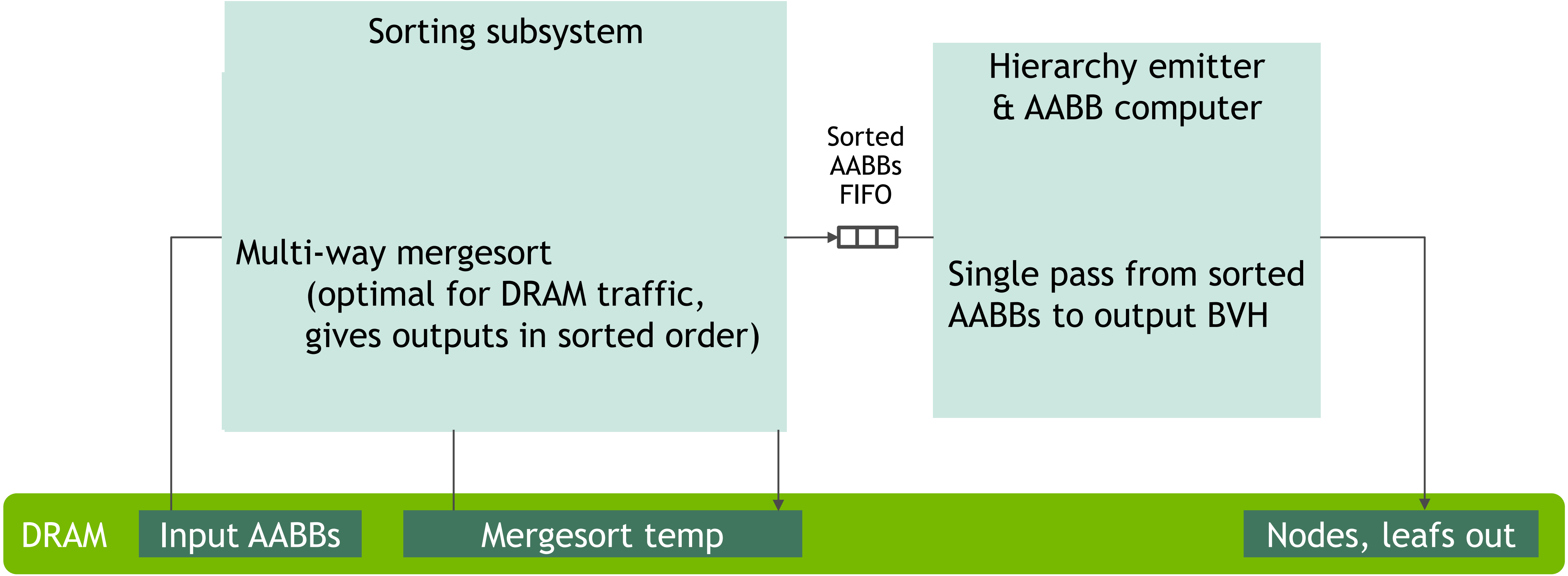
(One attempt to help this: extended Morton codes (Vinkler et al. 2017))



MERGETREE ARCHITECTURE



MERGETREE ARCHITECTURE



Sorting subsystem

Multi-way mergesort
(optimal for DRAM traffic,
gives outputs in sorted order)

Sorted
AABBs
FIFO

Hierarchy emitter
& AABB computer

Single pass from sorted
AABBs to output BVH

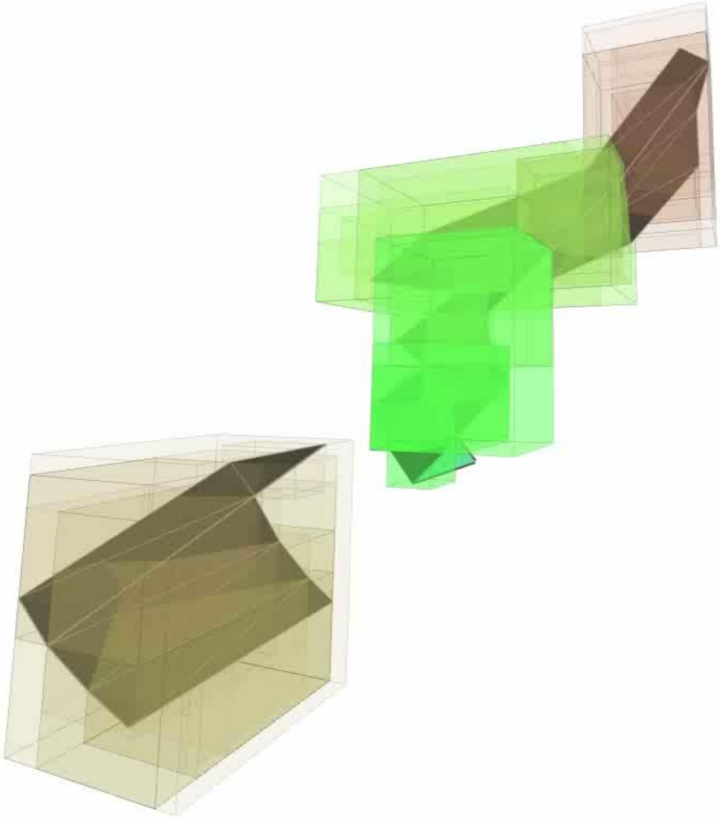
DRAM

Input AABBs

Mergesort temp

Nodes, leafs out

Streaming Hierarchy Emission



MERGETREE

~10x smaller than silicon area binned SAH; ~5x faster builds, ~3x less DRAM traffic

...But quality is much worse

Straight HW implementation of GPU algorithm would have ~2.5x more traffic

Single pipeline doesn’t quite catch up to a high/end GPU running SW LBVH but comes close (0.68x speed)

HARDWARE TREE BUILDERS: PLOCTREE

MODERN GPU BVH BUILDERS

Binned SAH is high-quality but expensive

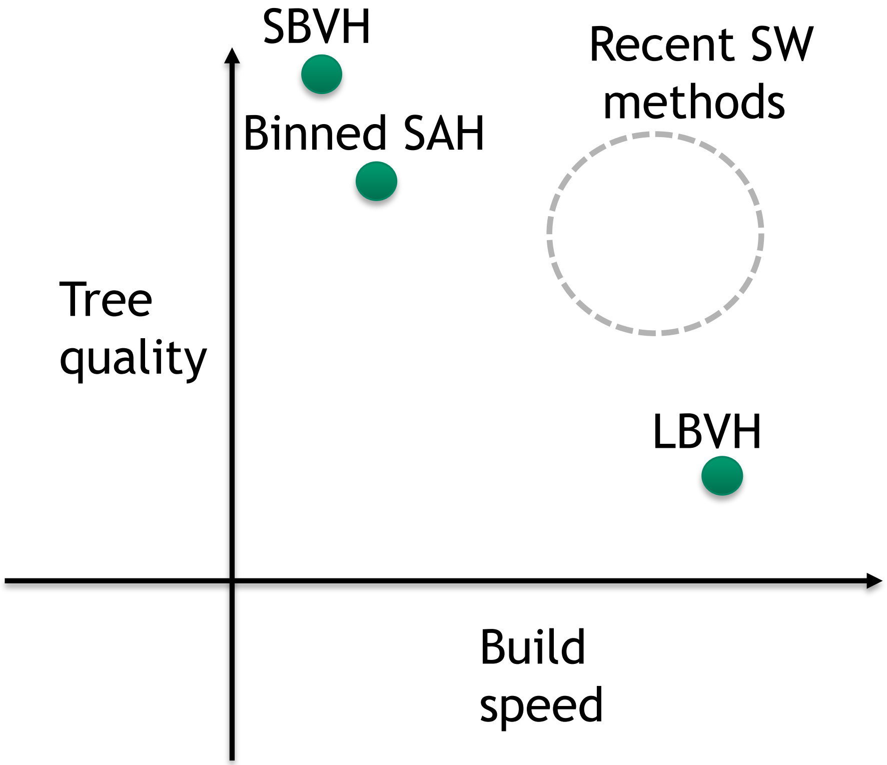
LBVH is cheap but low-quality

Recent GPU builders often try to start with Morton code sorting or full LBVH, and then improve quality

E.g. HLBVH (Pantaleoni and Luebke 2010), TRBVH (Karras 2013), ATRBVH (Domingues and Pedrini 2015), PLOC (Meister and Bittner 2018).

PLOC looks suitable for HW implementation

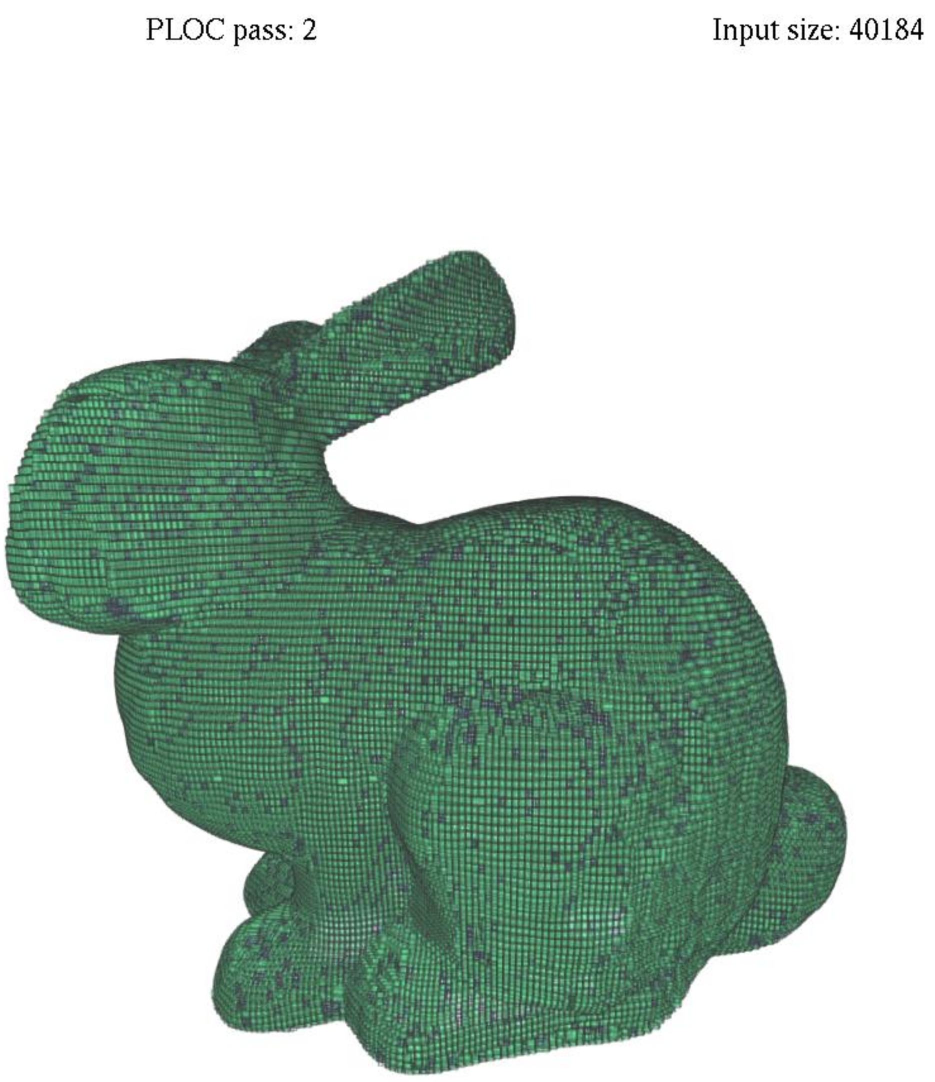
→Adapt to a HW architecture, PLOCTree



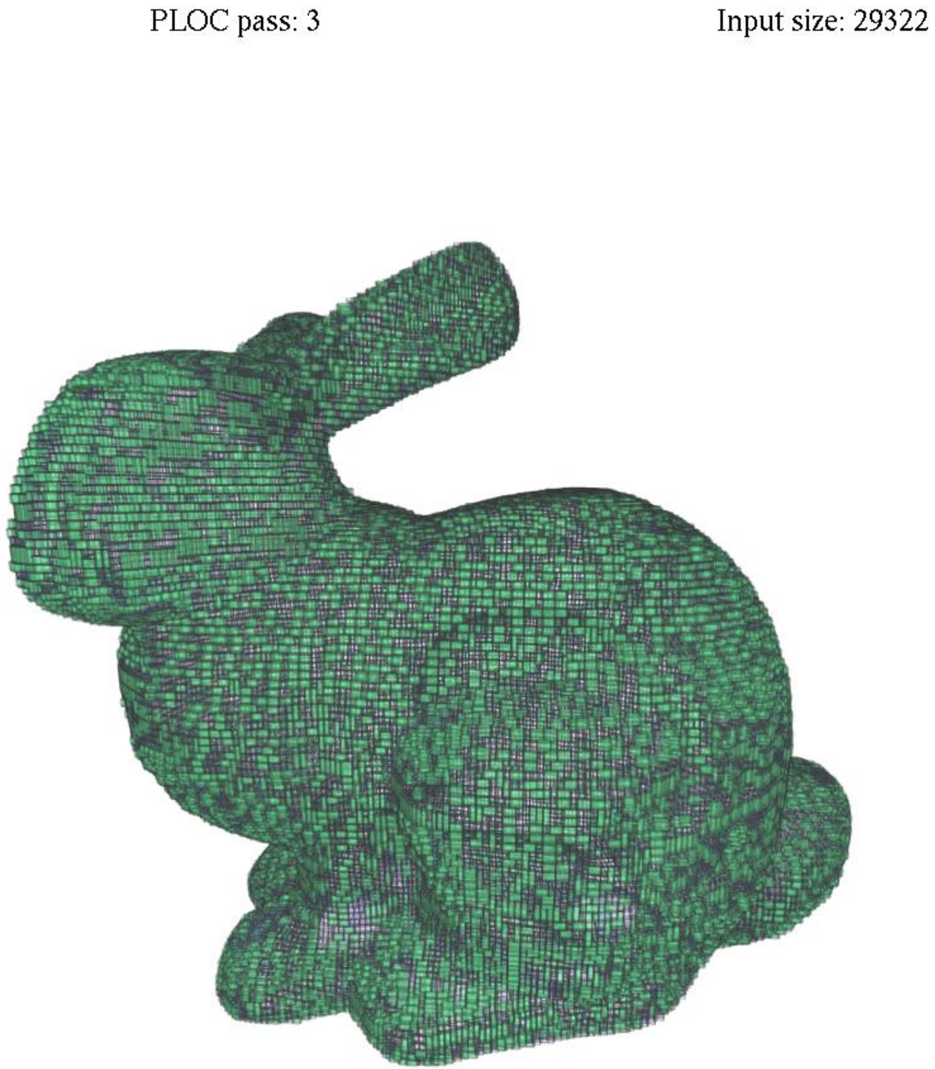
PLOC

PLOC pass: 1
Input size: 69666

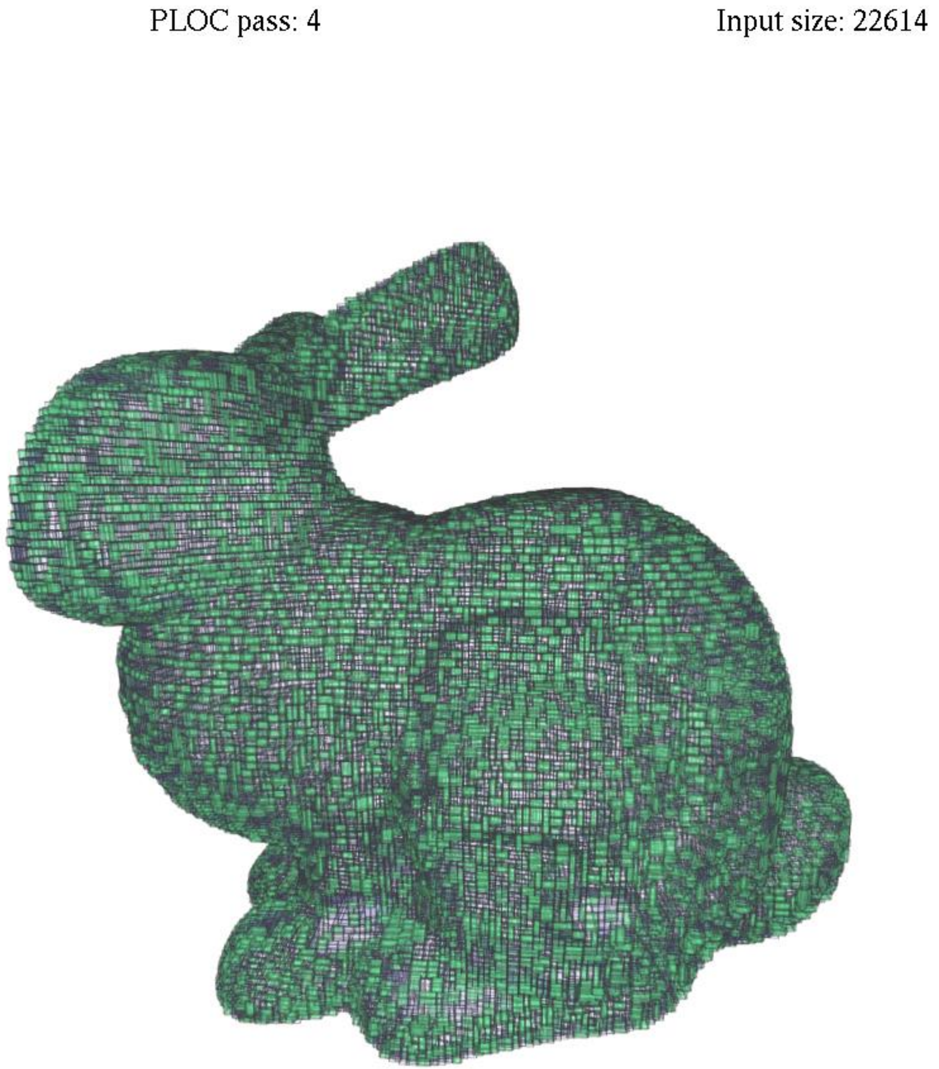
PLOC



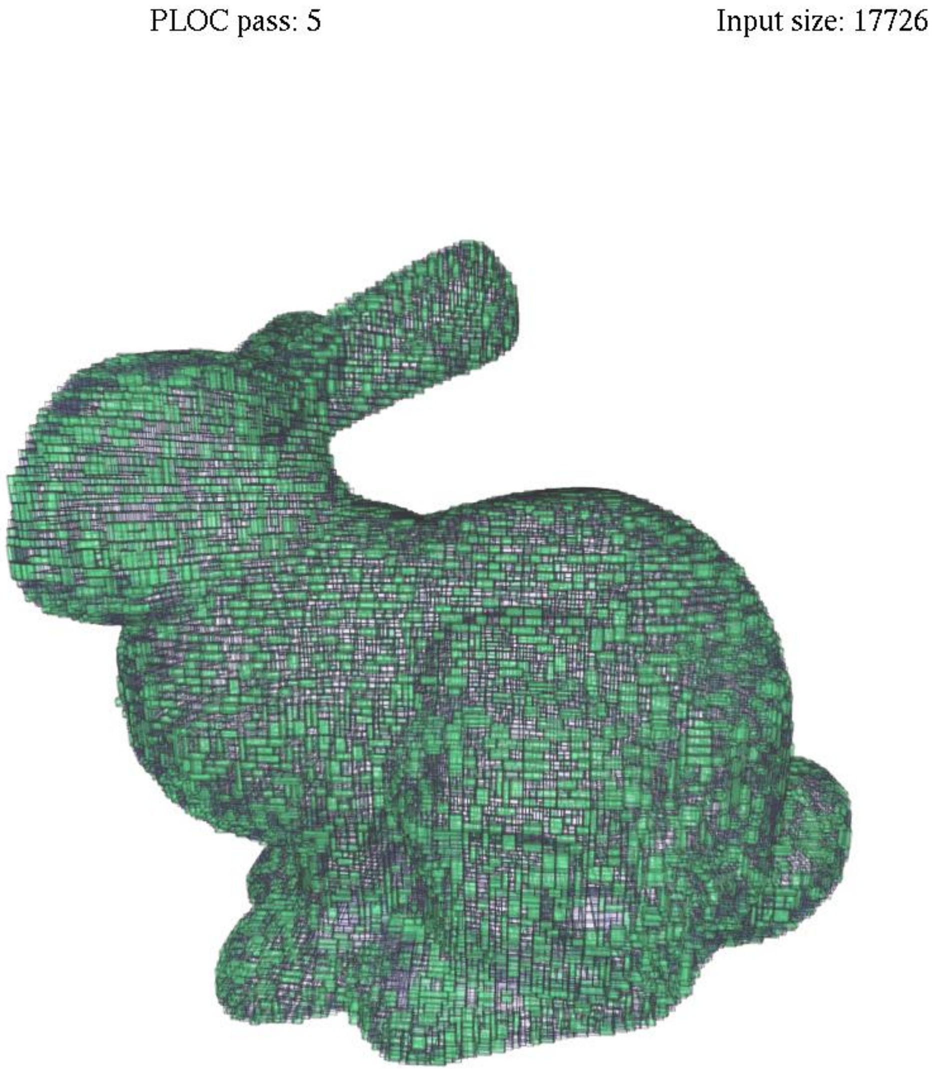
PLOC



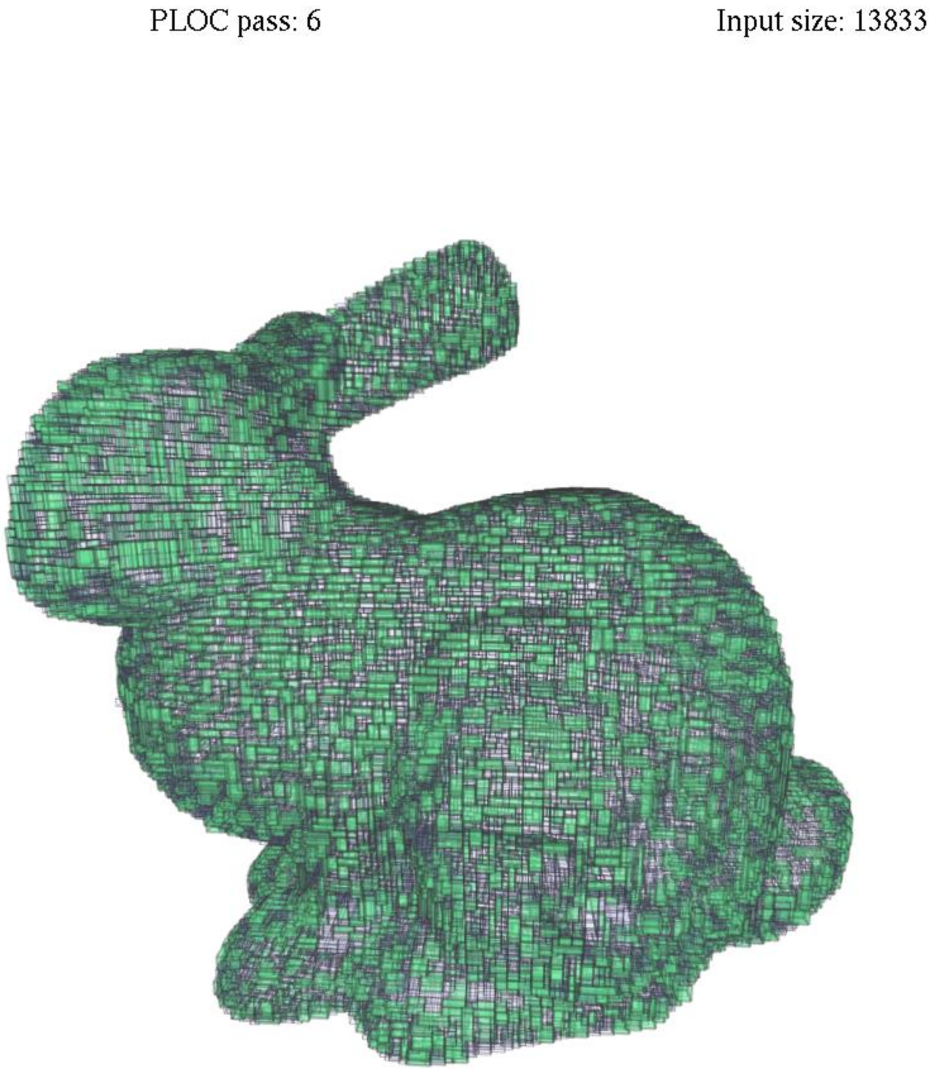
PLOC



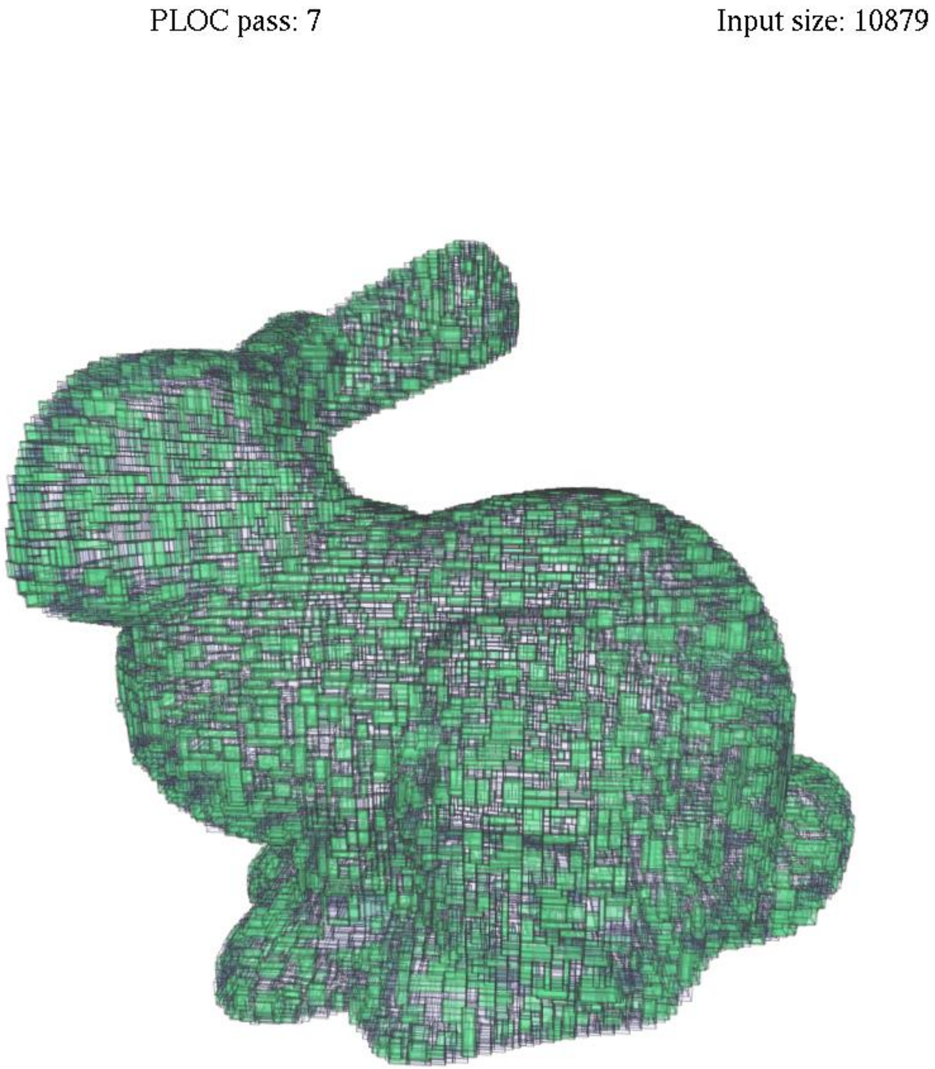
PLOC



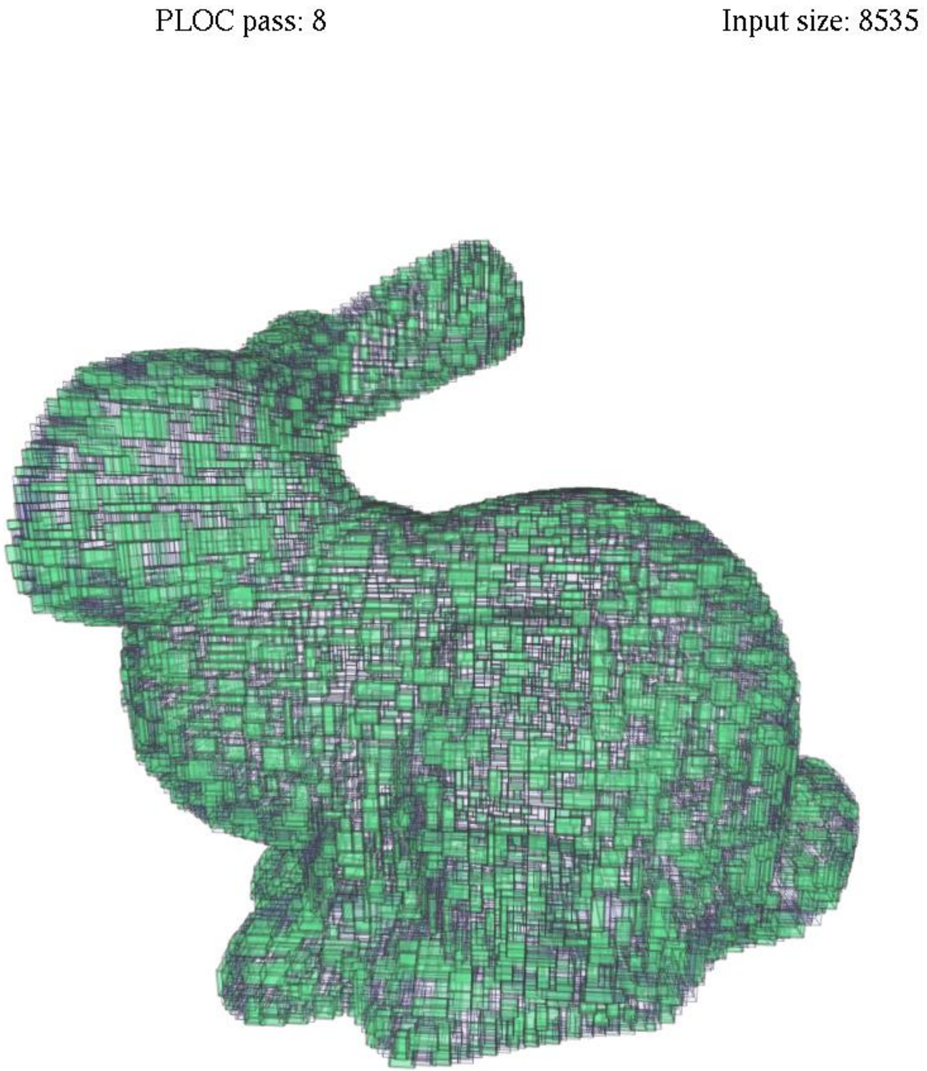
PLOC



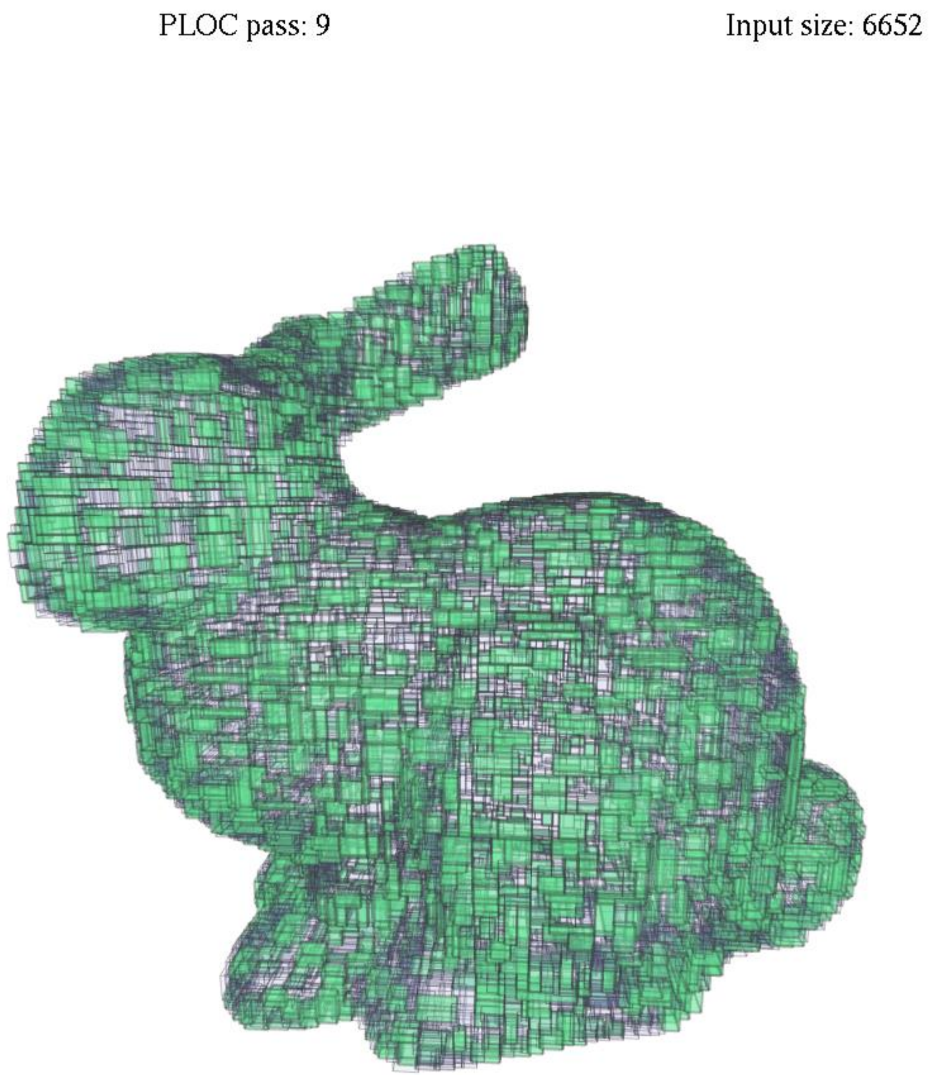
PLOC



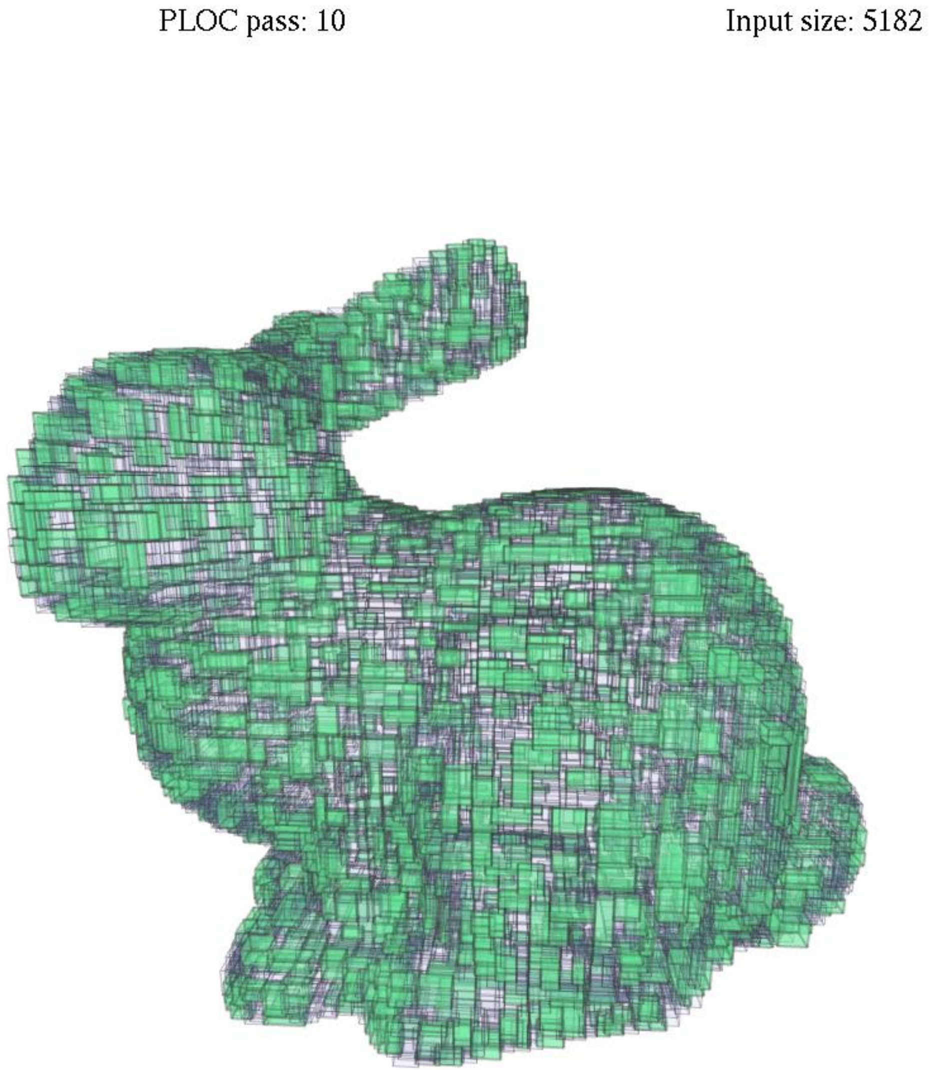
PLOC



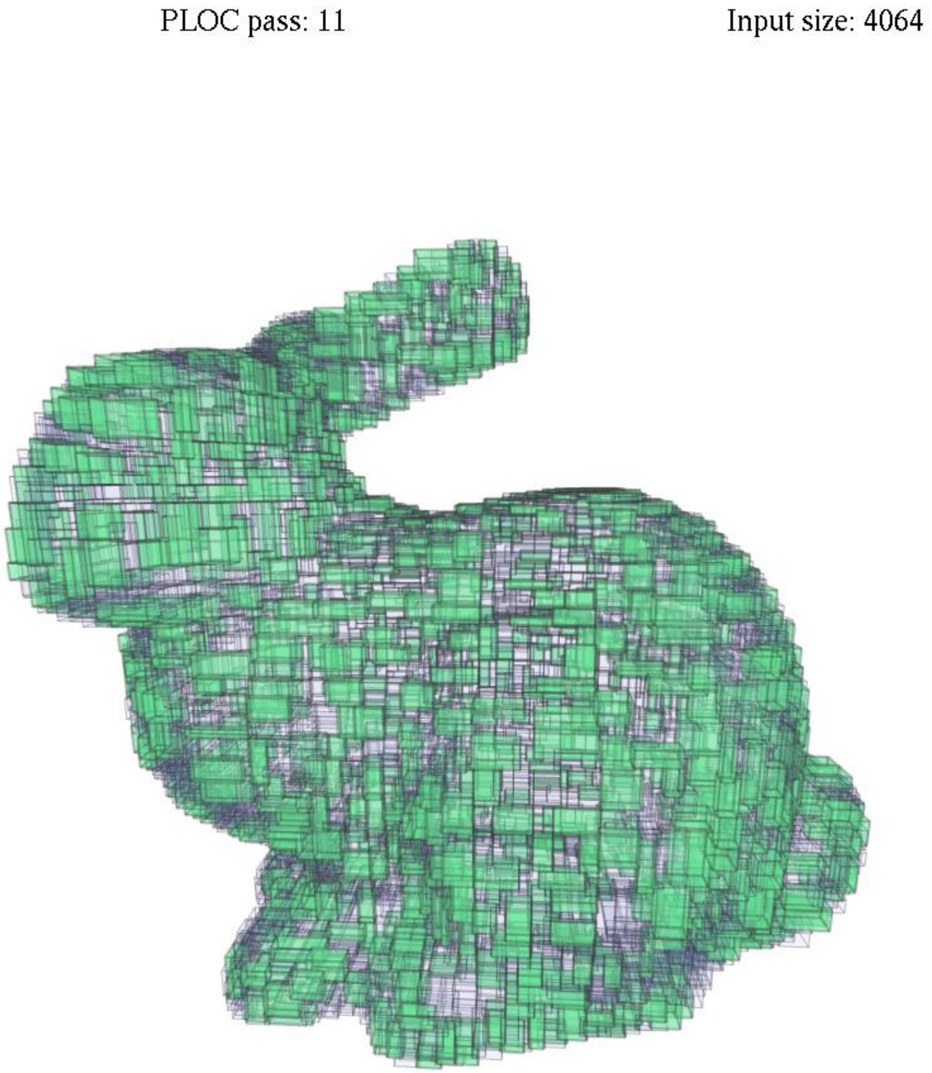
PLOC



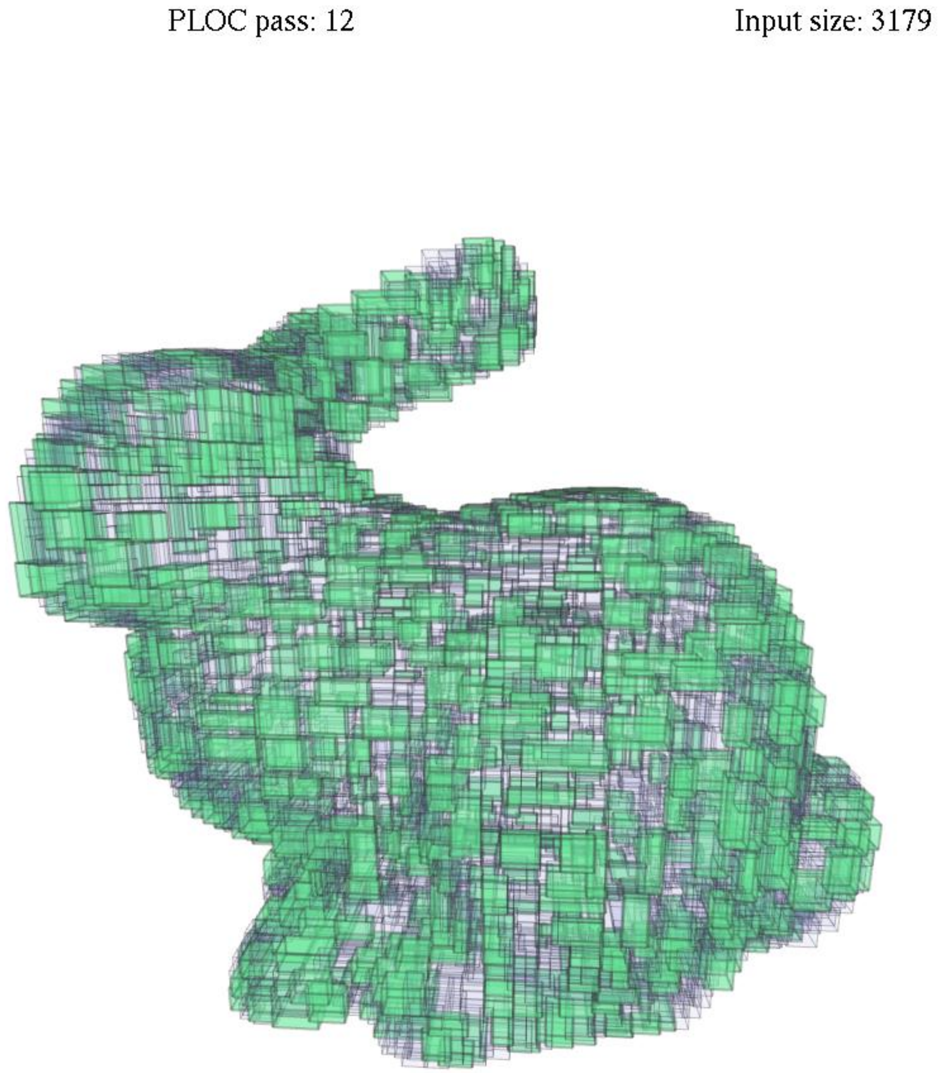
PLOC



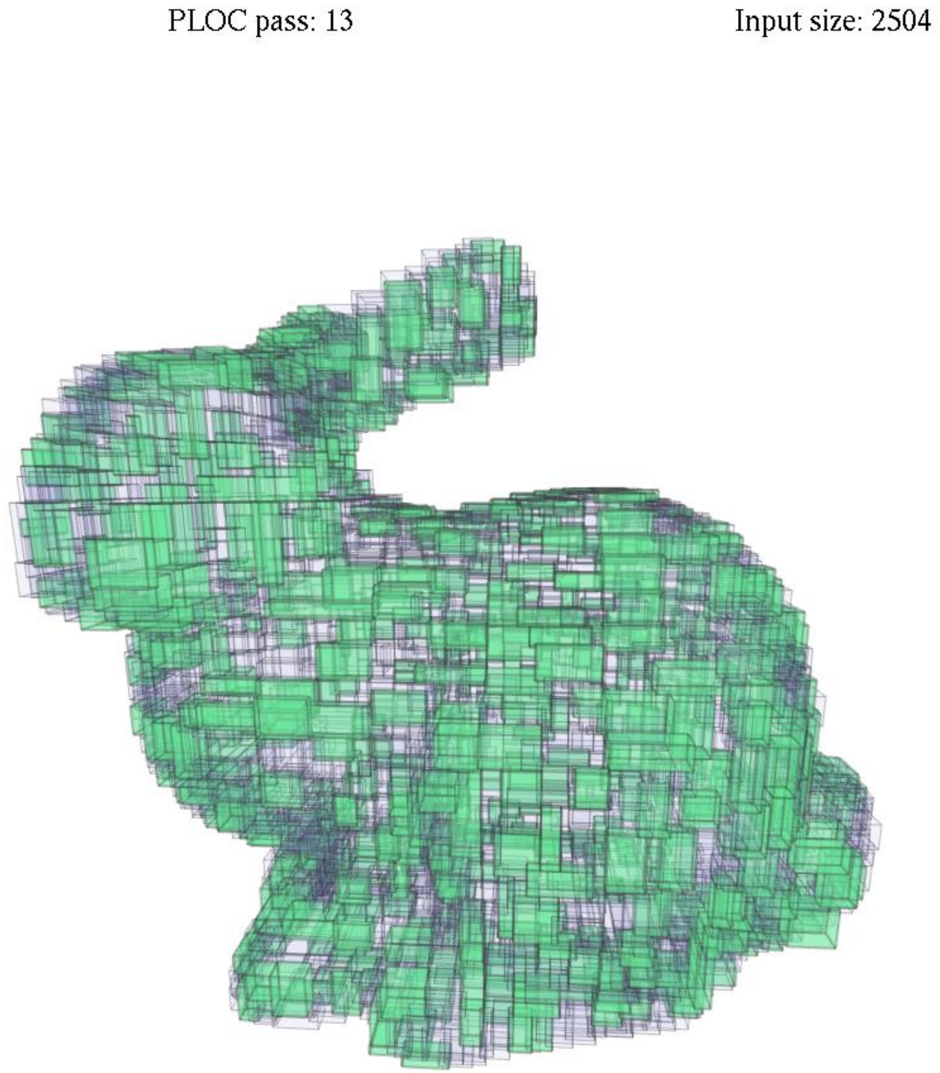
PLOC



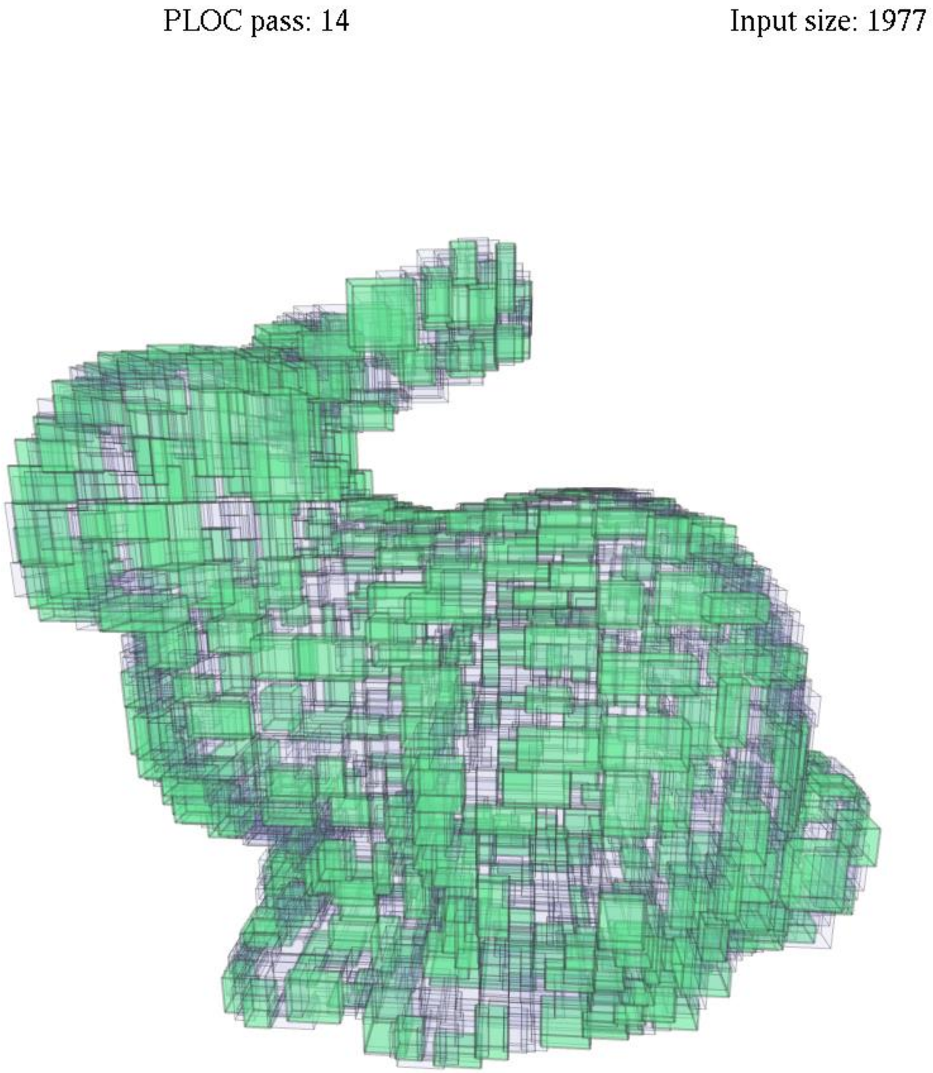
PLOC



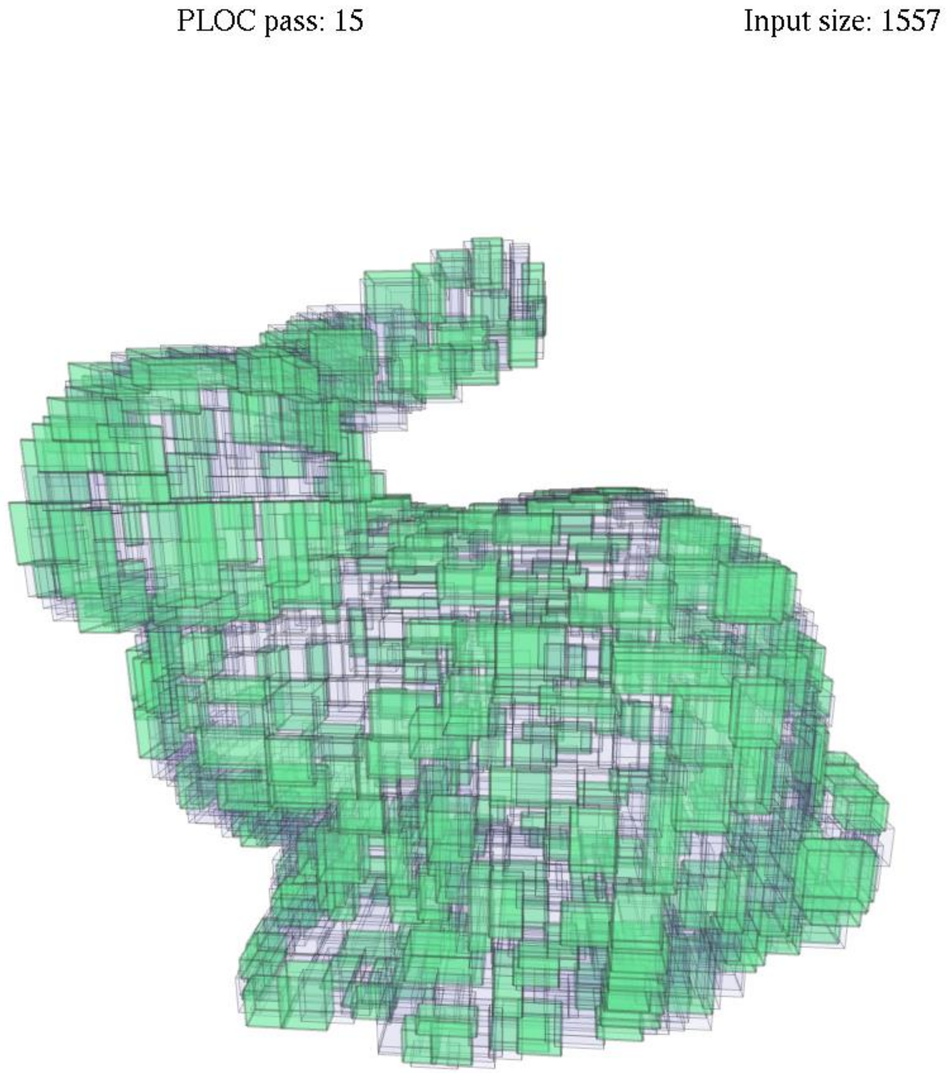
PLOC



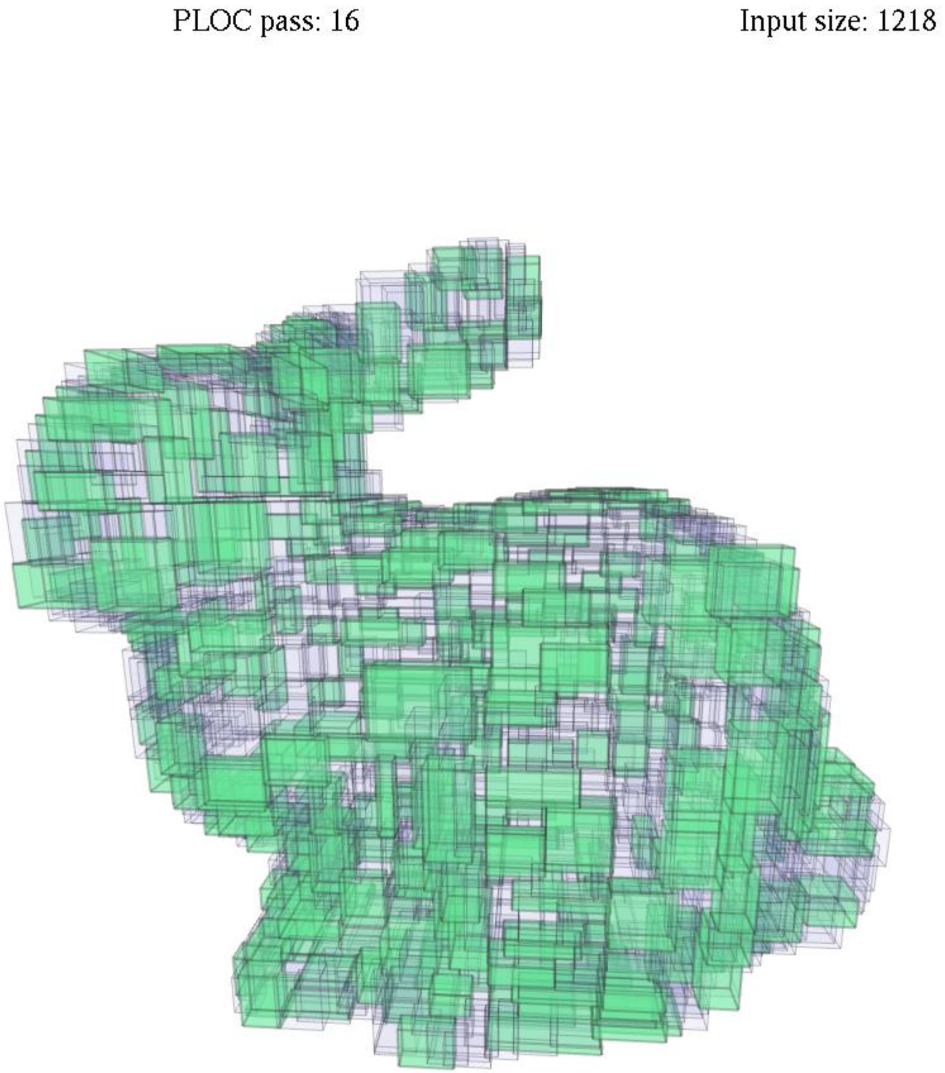
PLOC



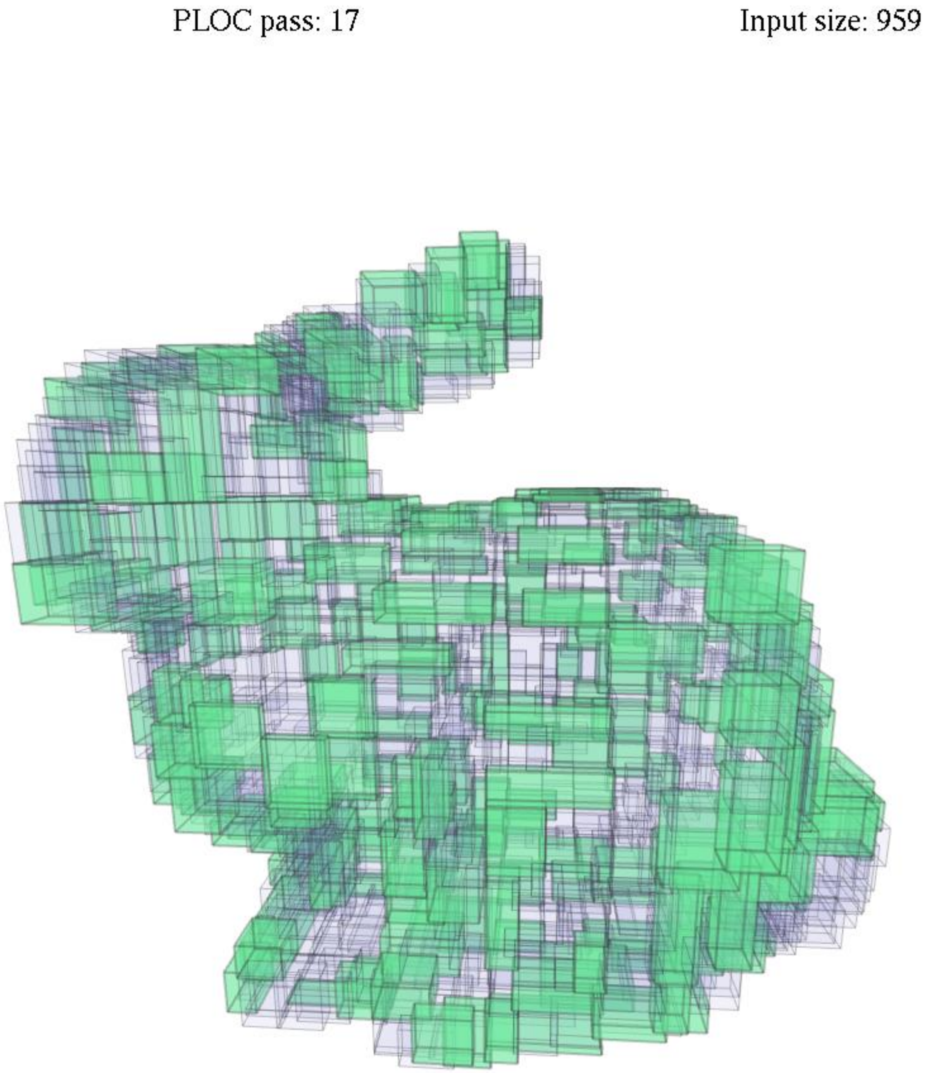
PLOC



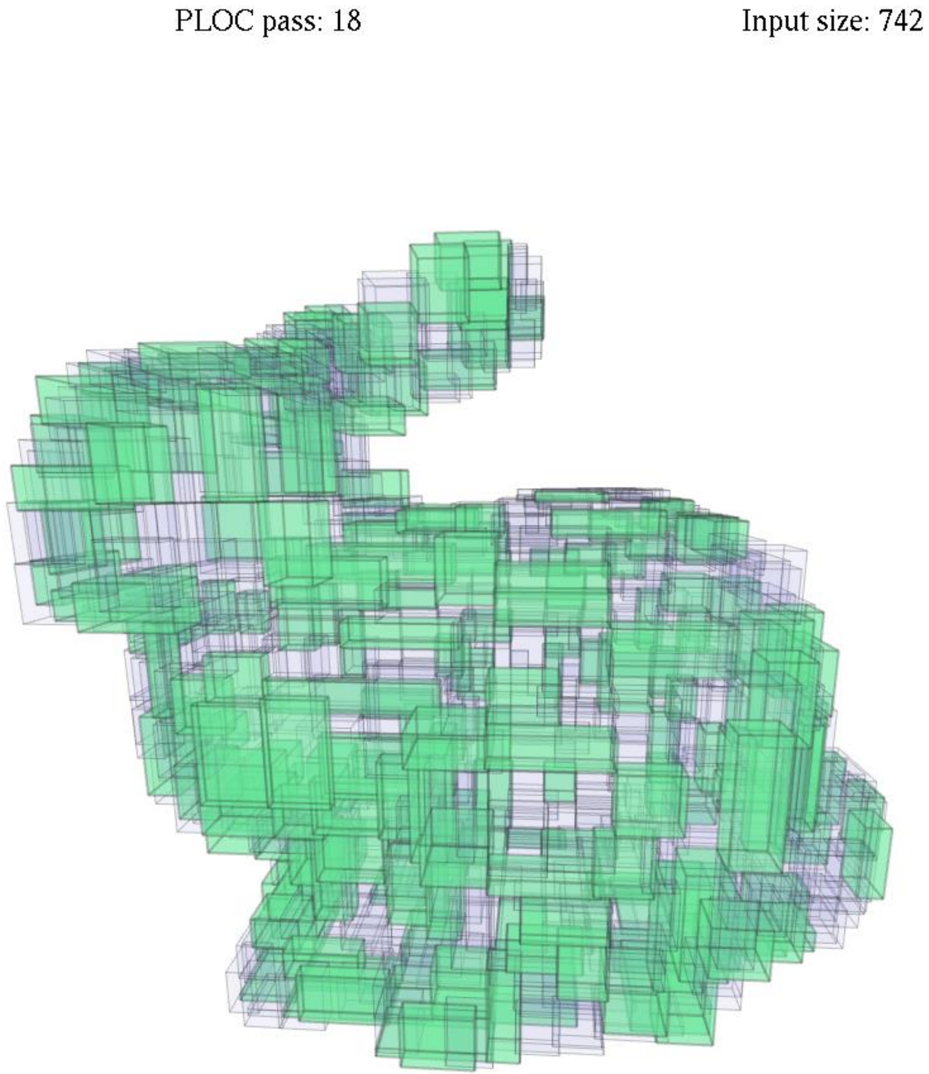
PLOC



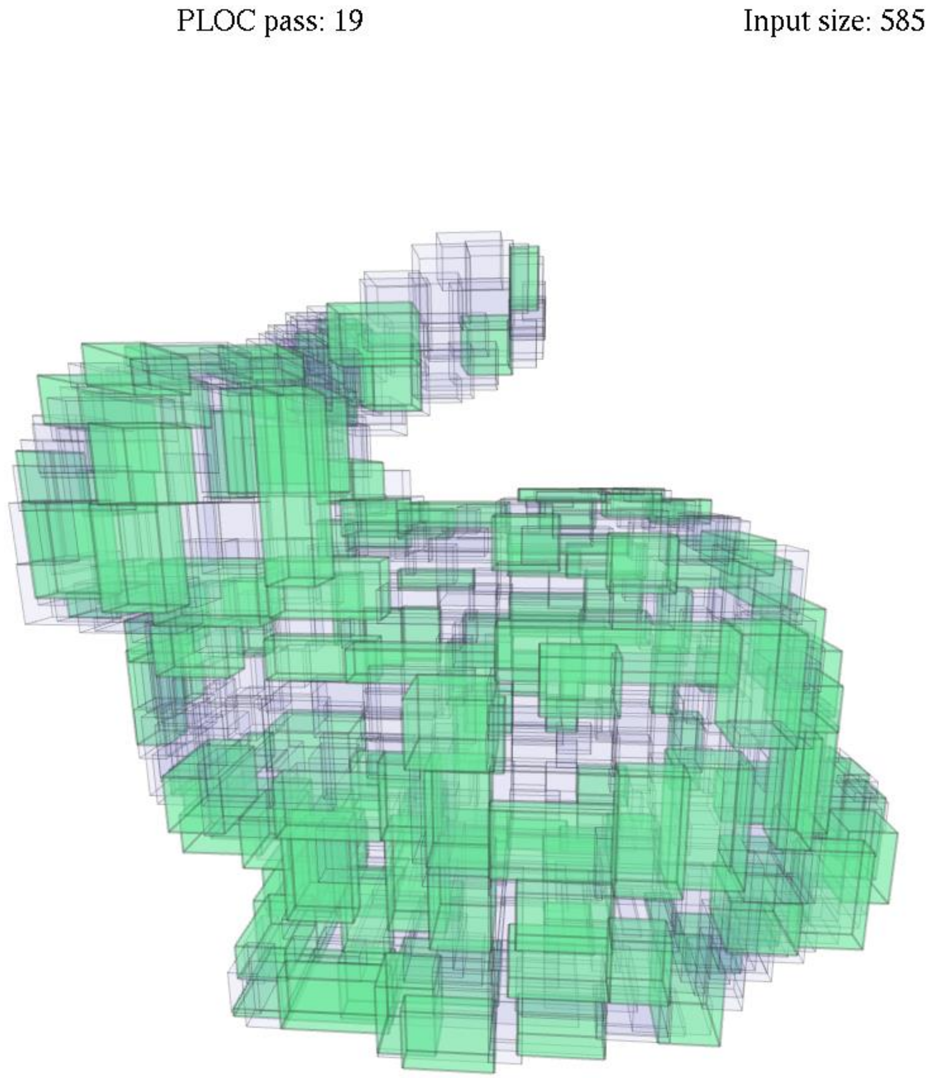
PLOC



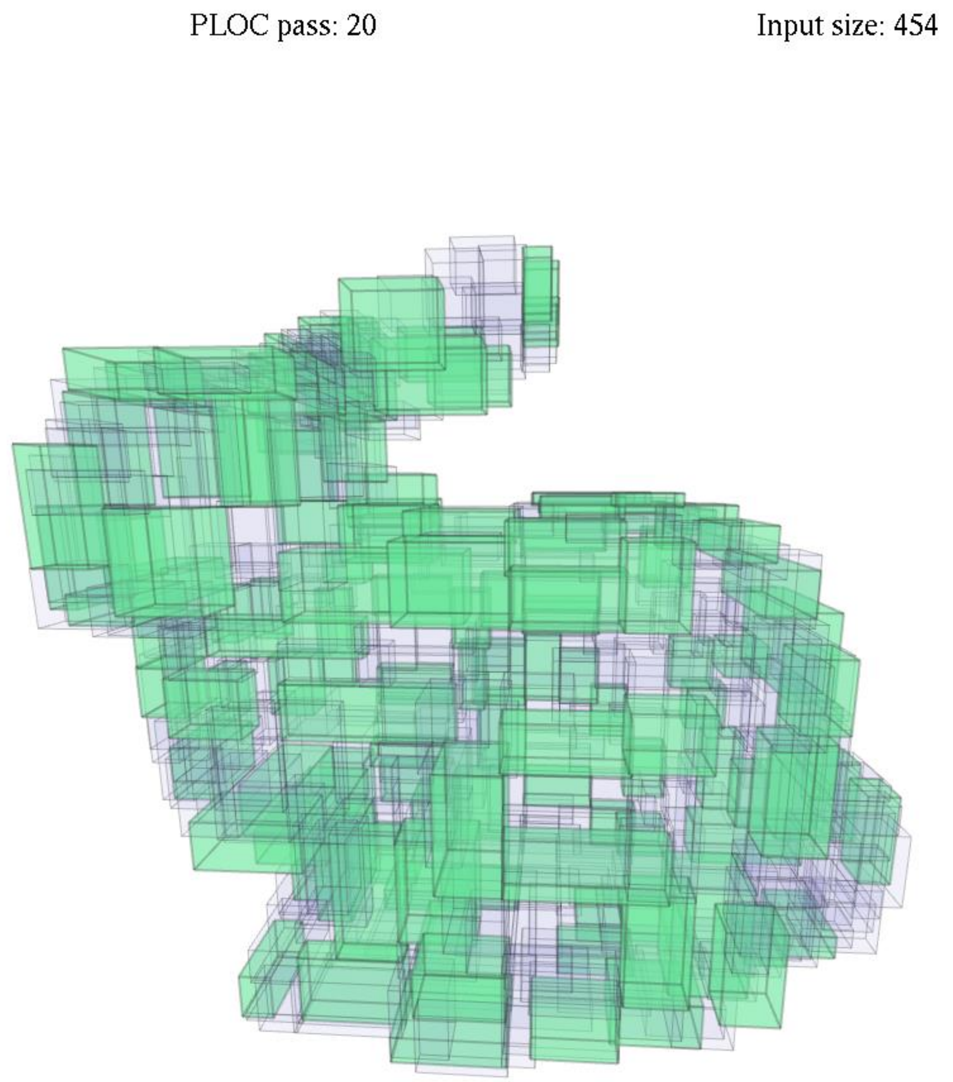
PLOC



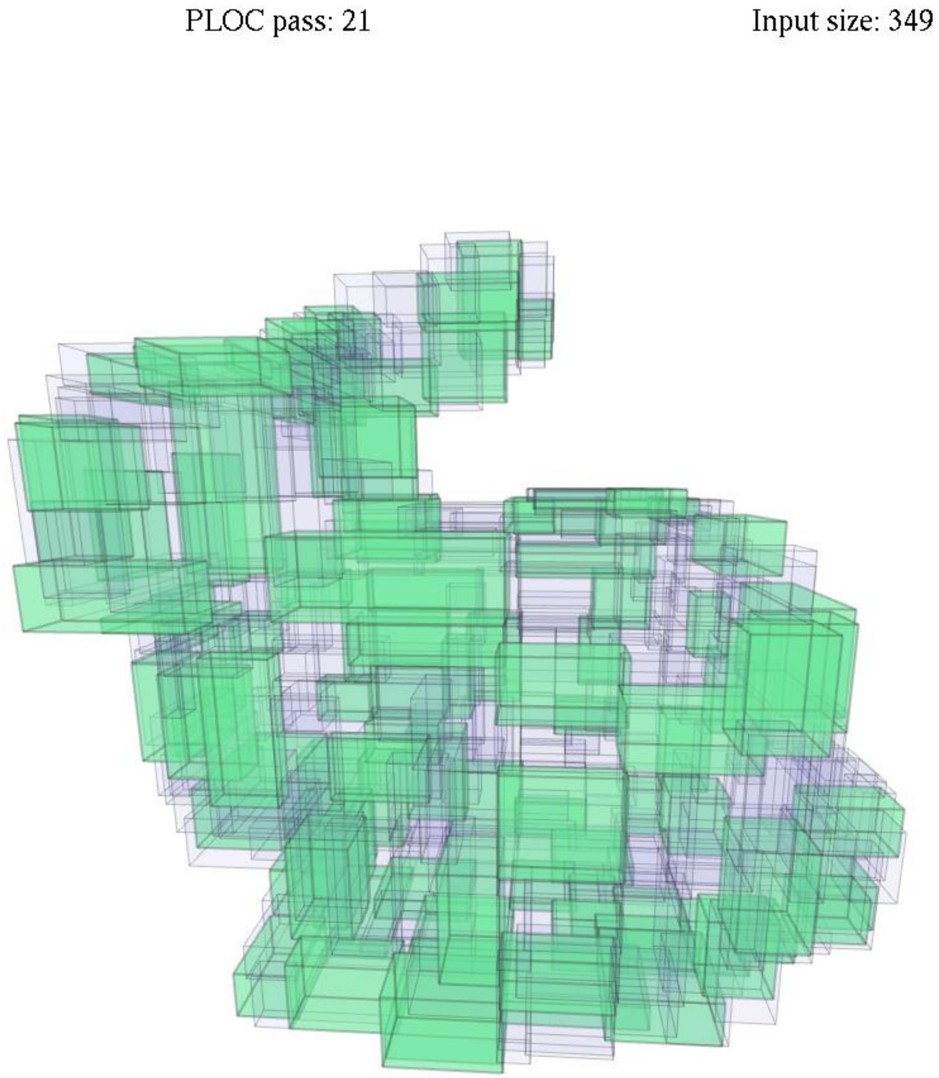
PLOC



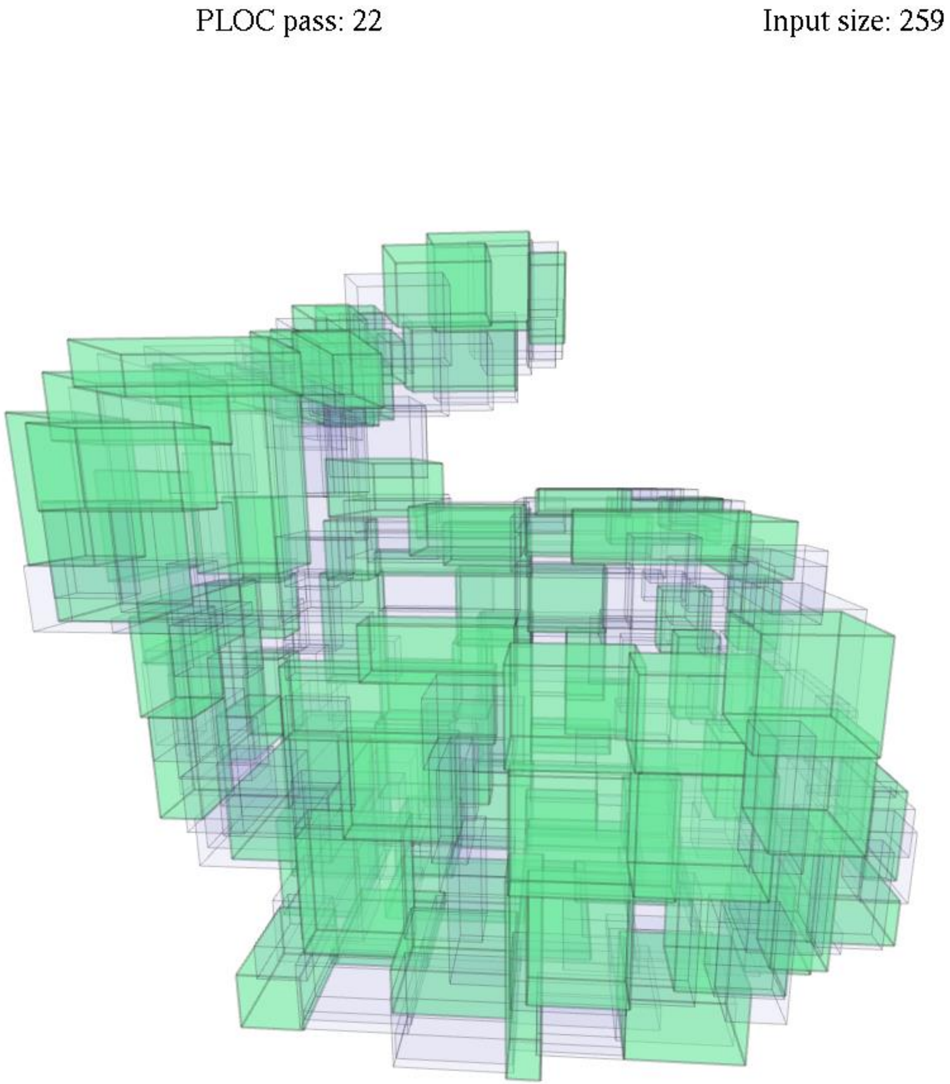
PLOC



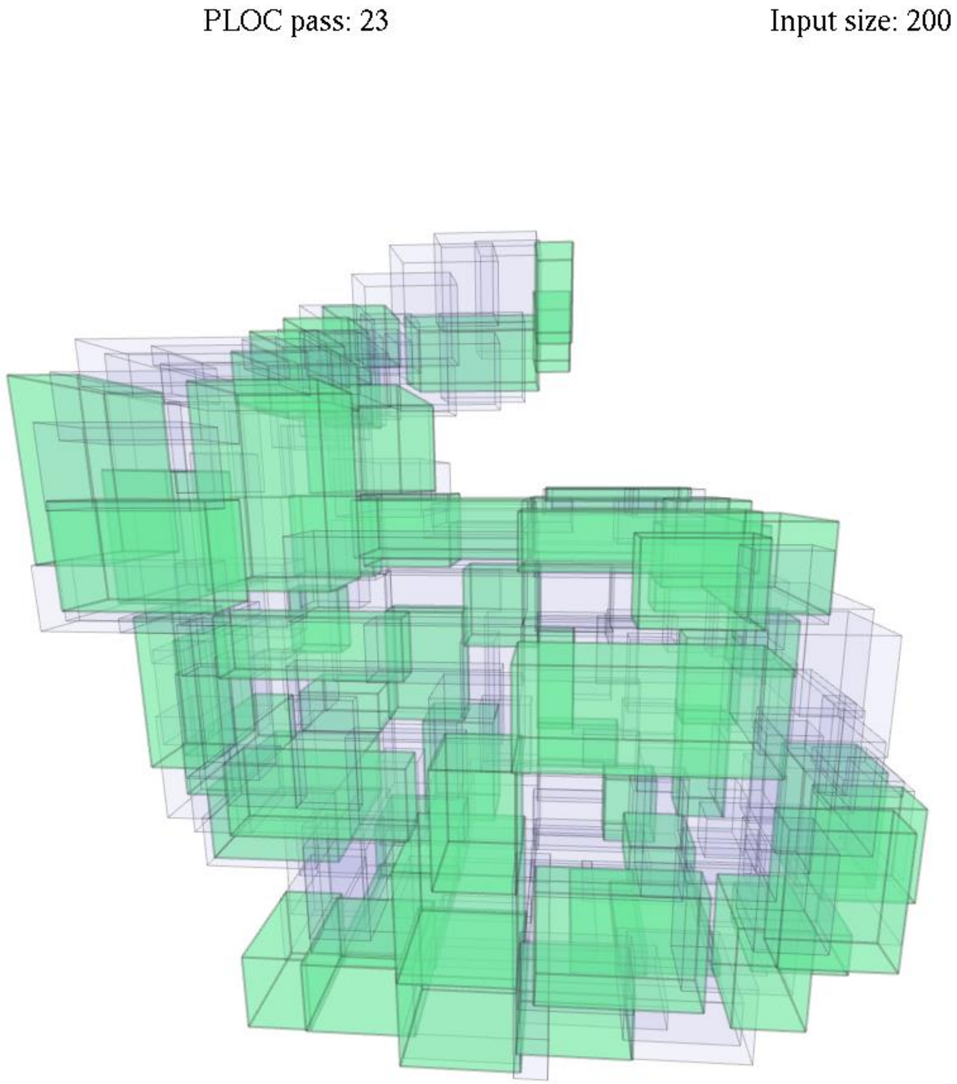
PLOC



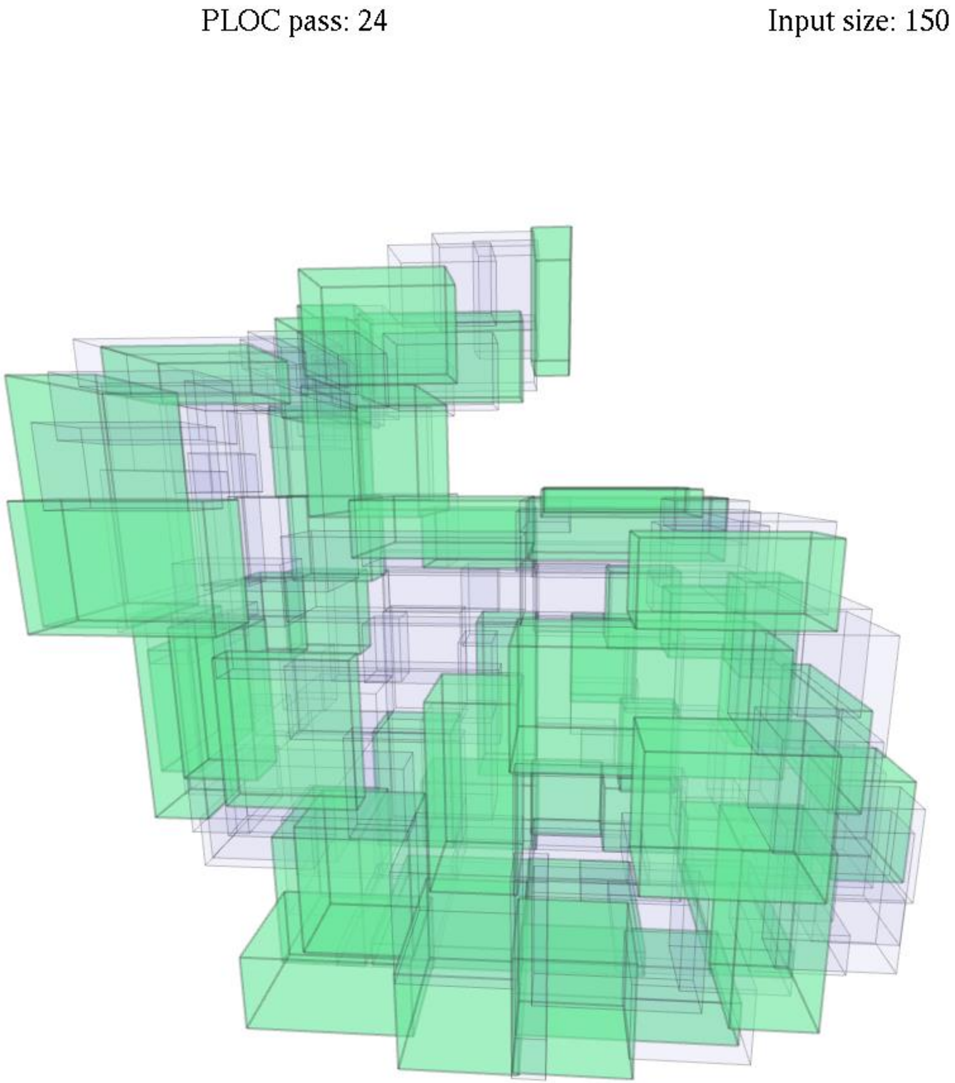
PLOC



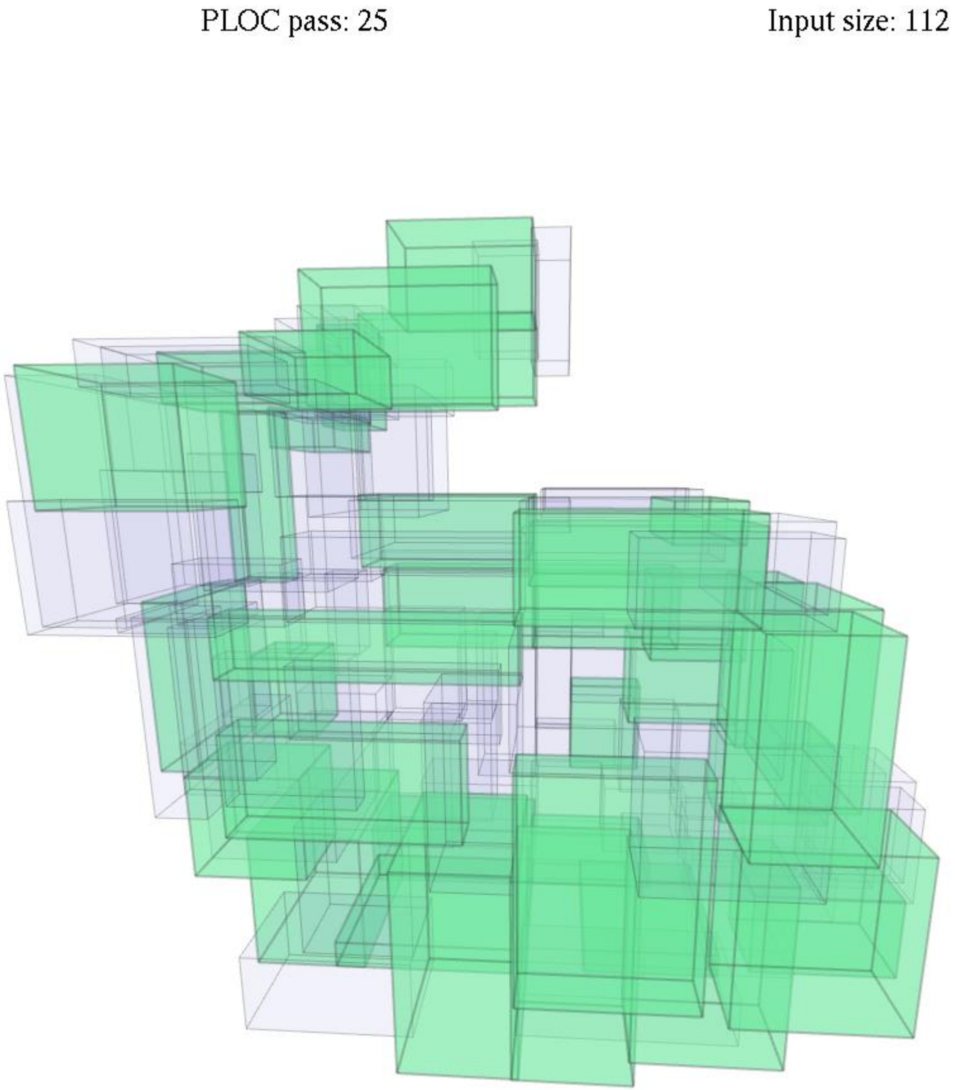
PLOC



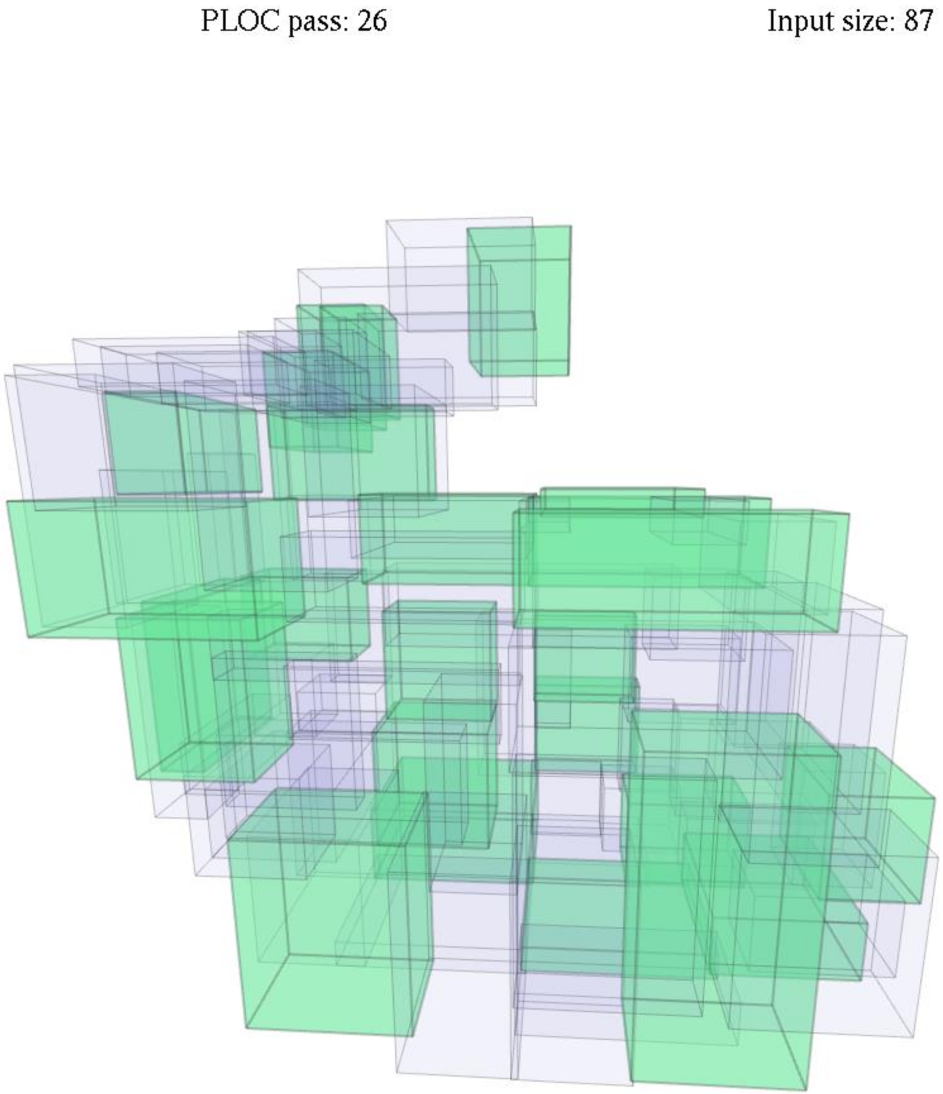
PLOC



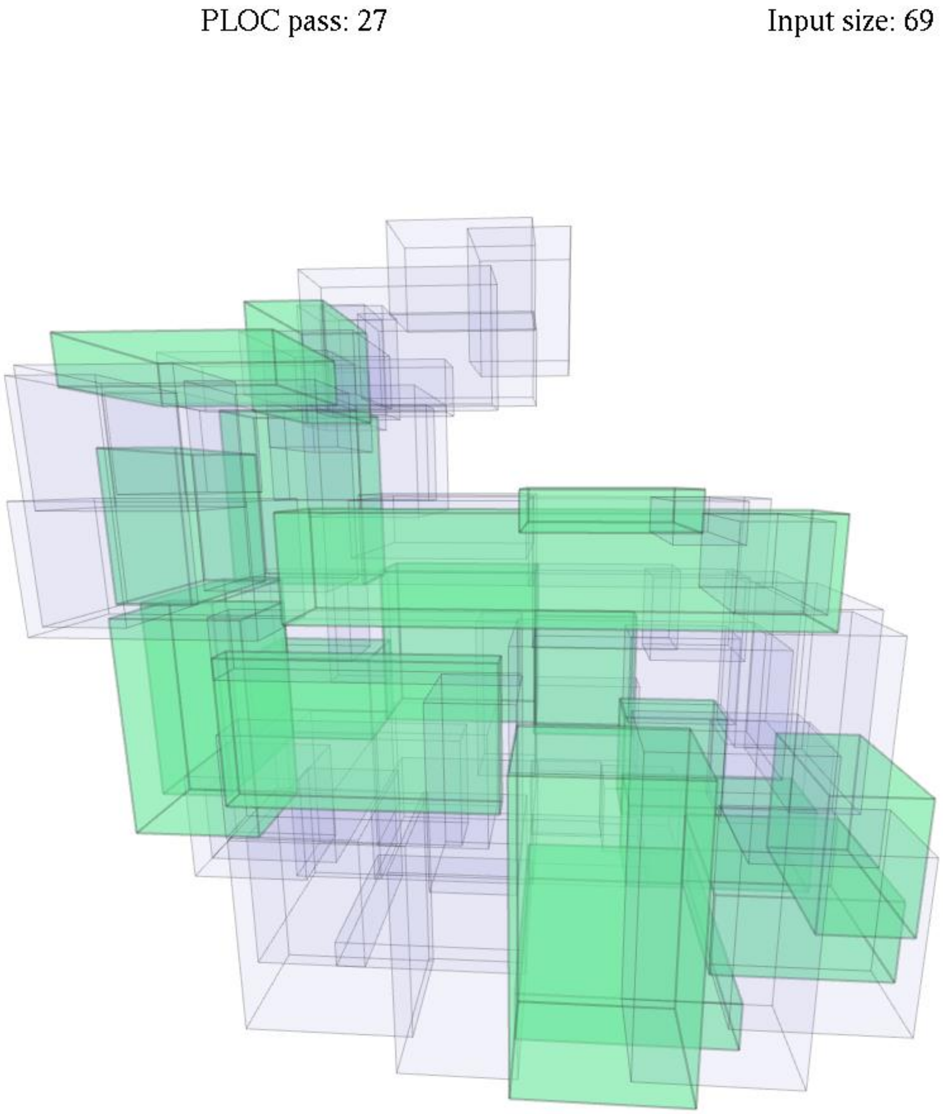
PLOC



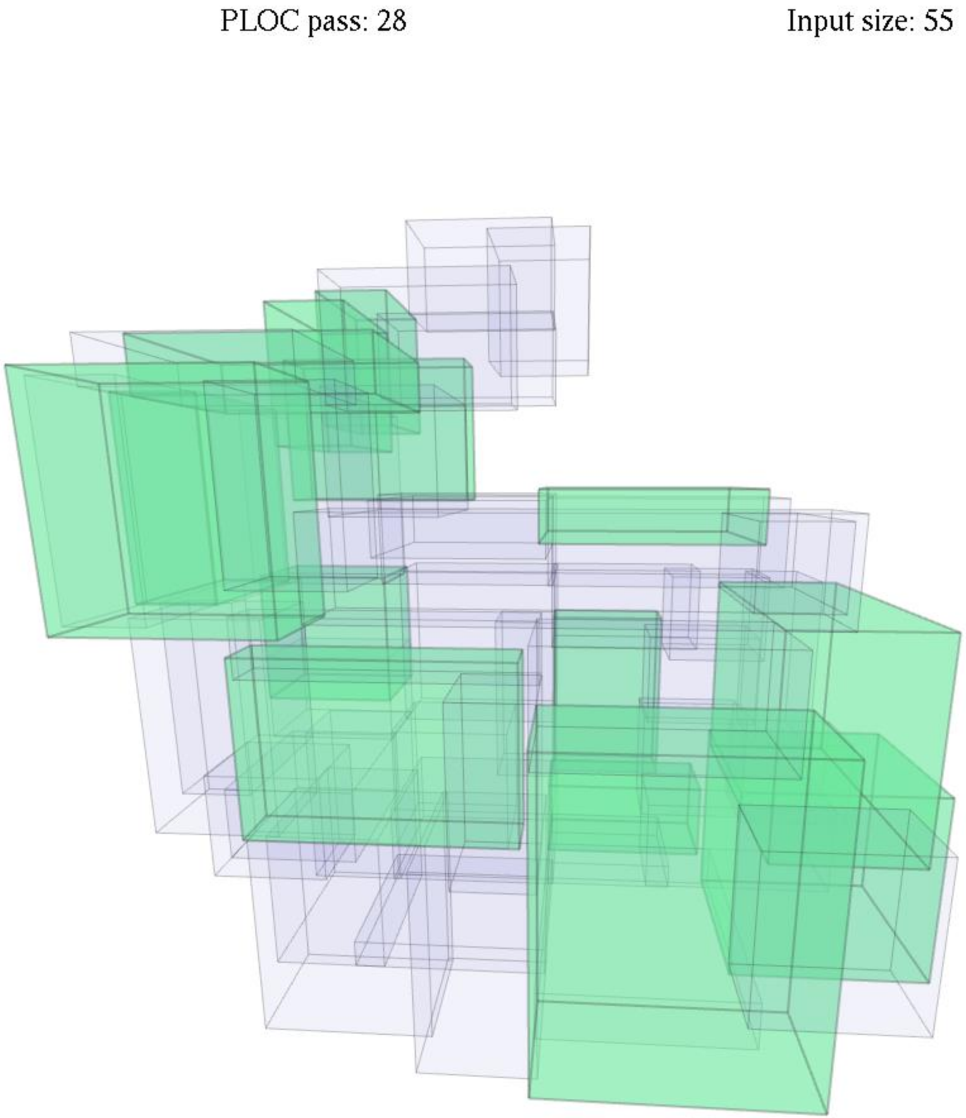
PLOC



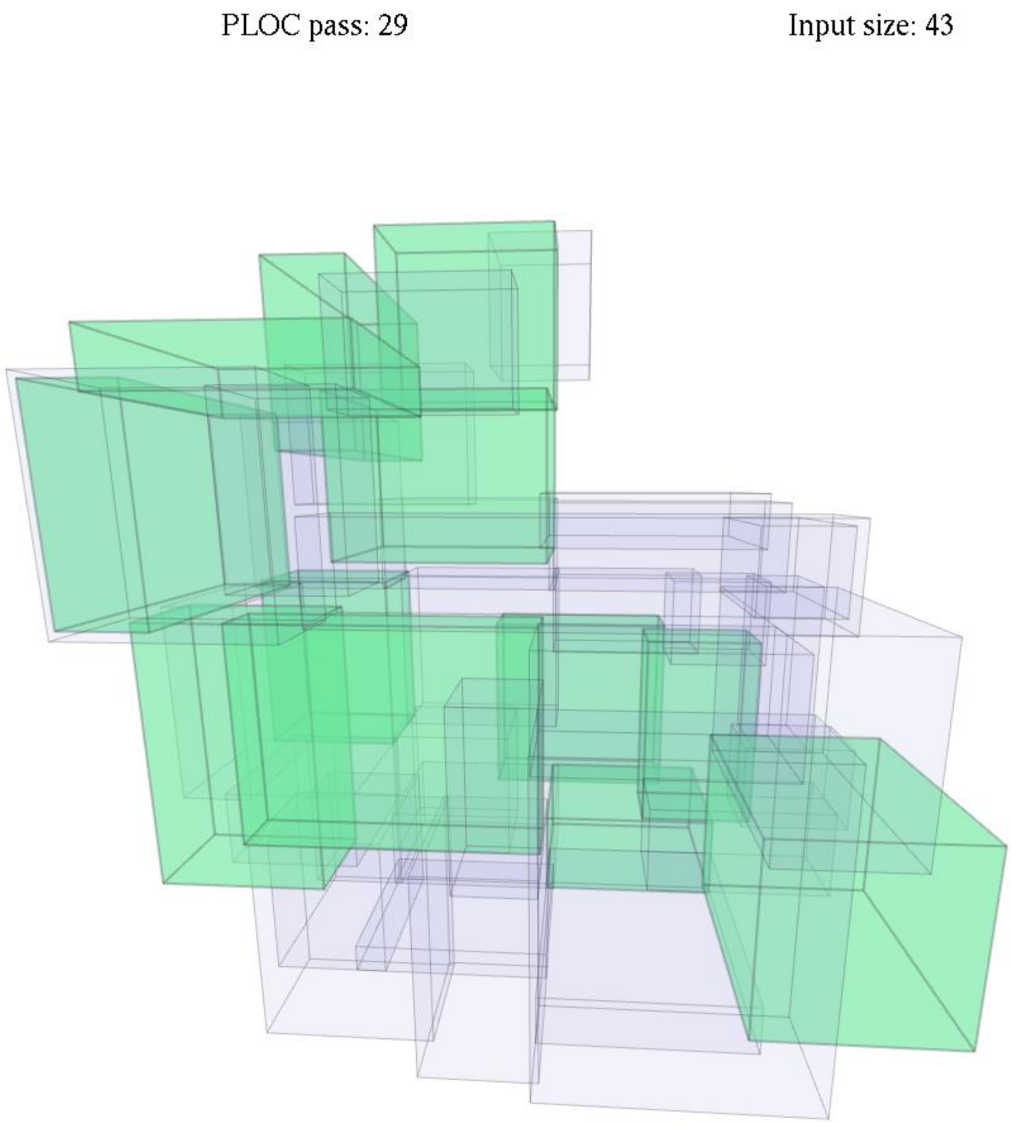
PLOC



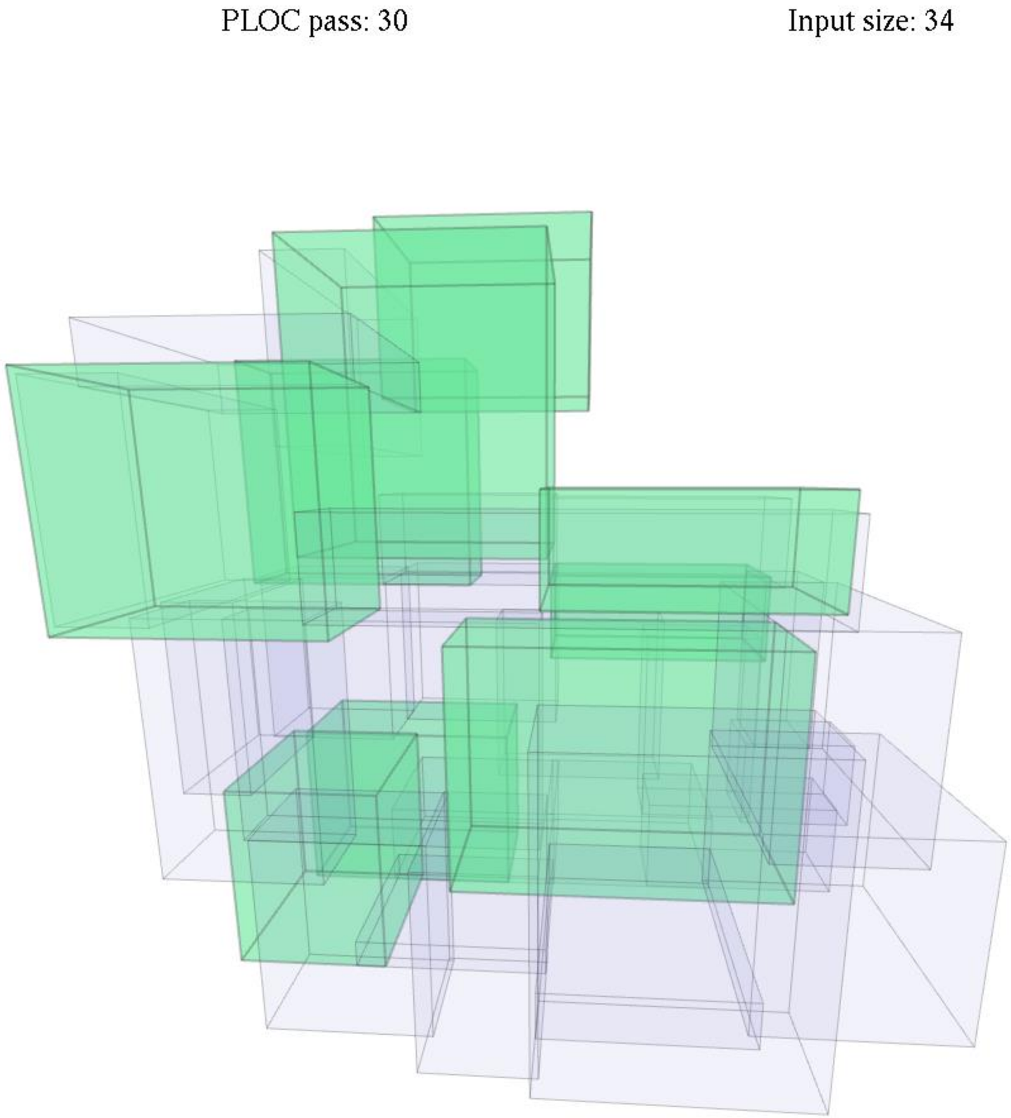
PLOC



PLOC



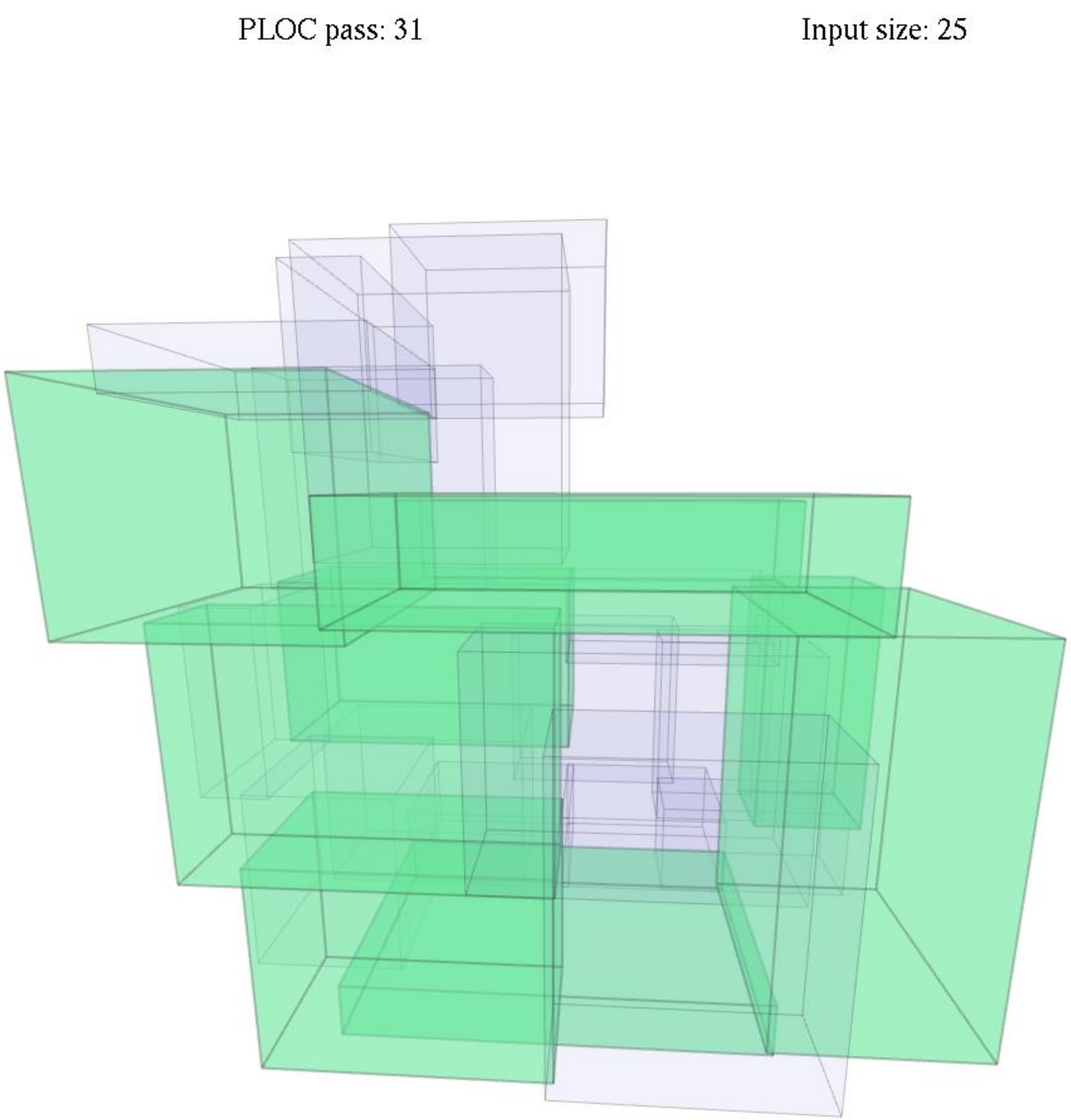
PLOC



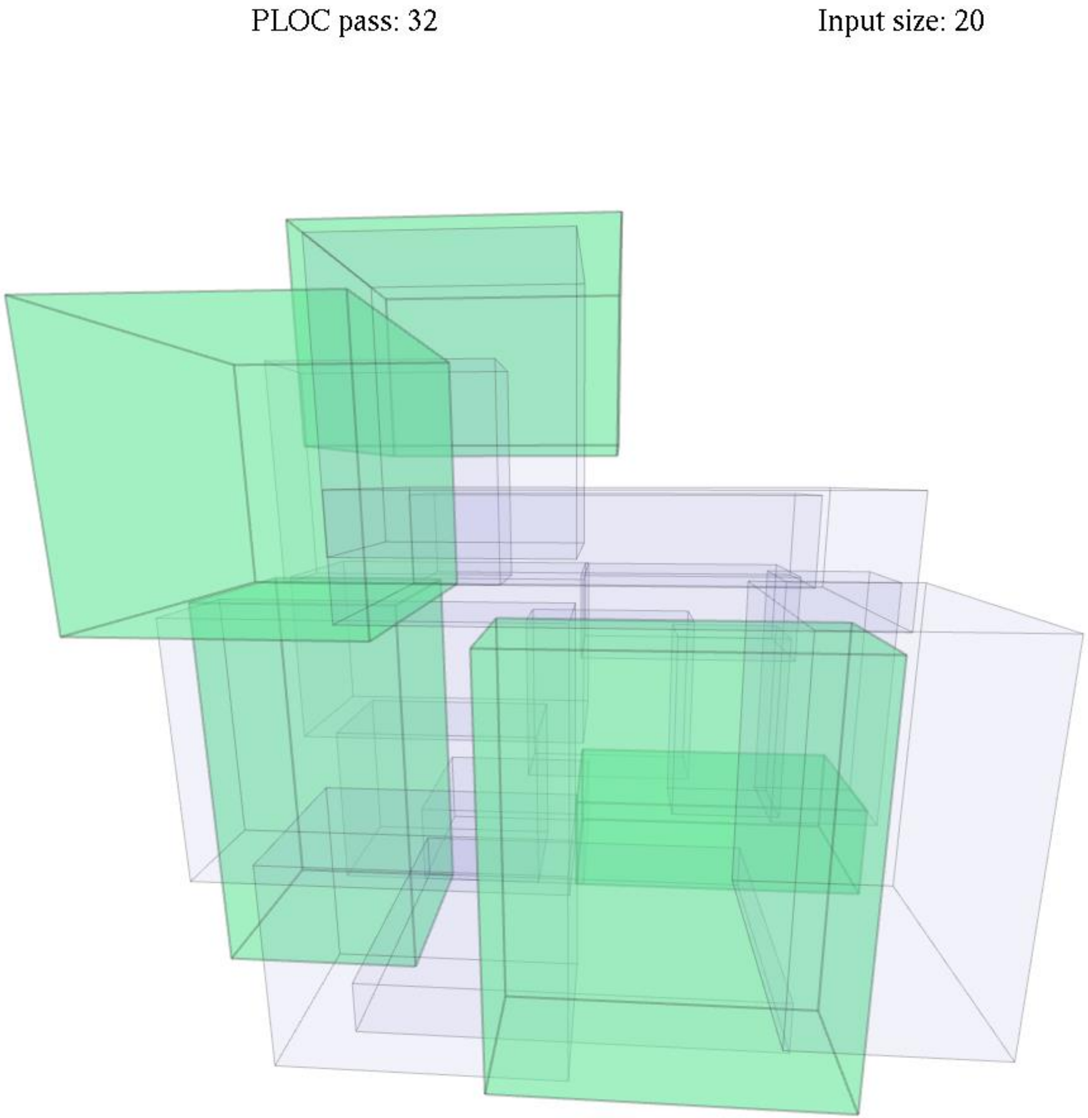
PLOC pass: 30

Input size: 34

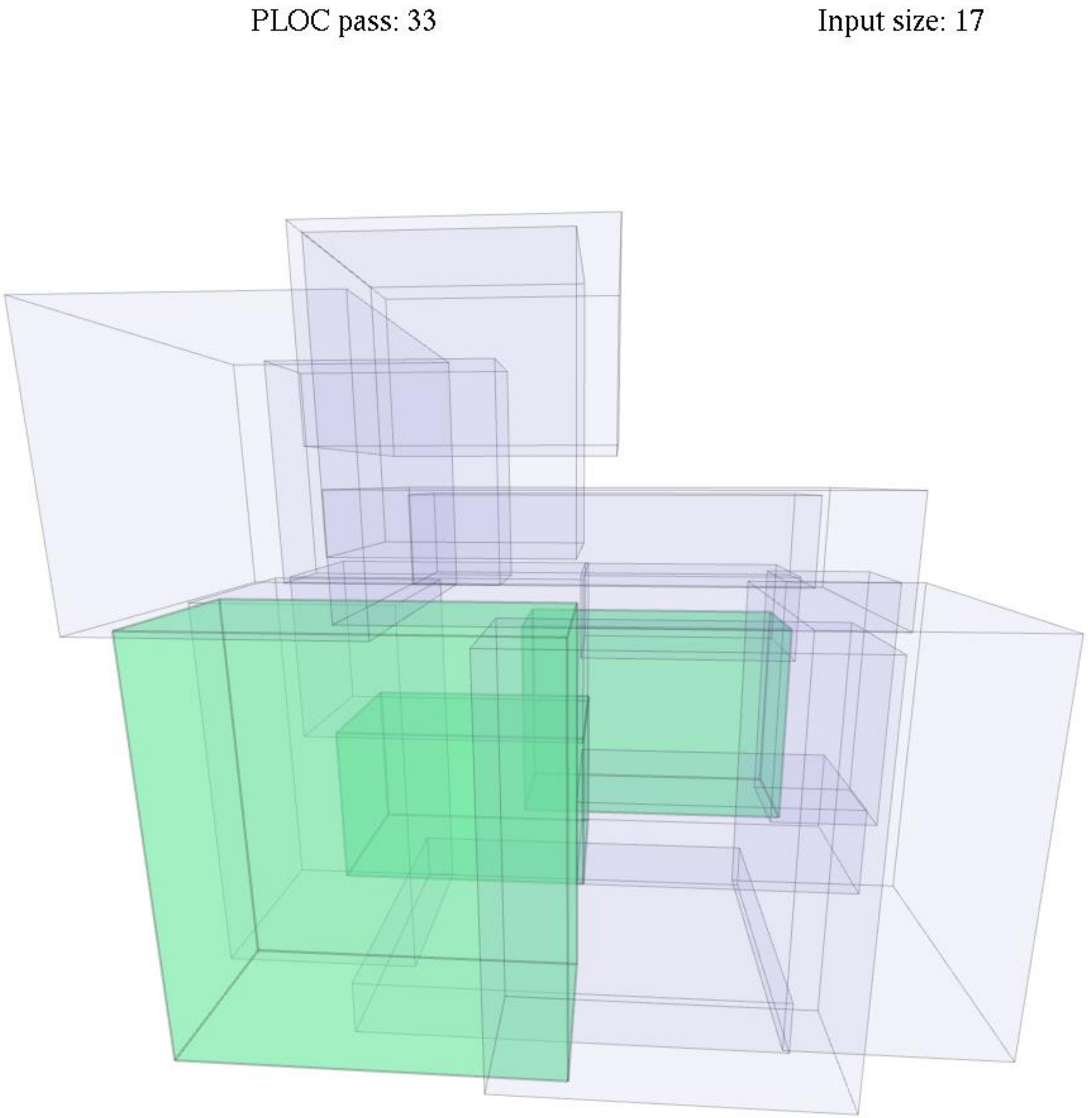
PLOC



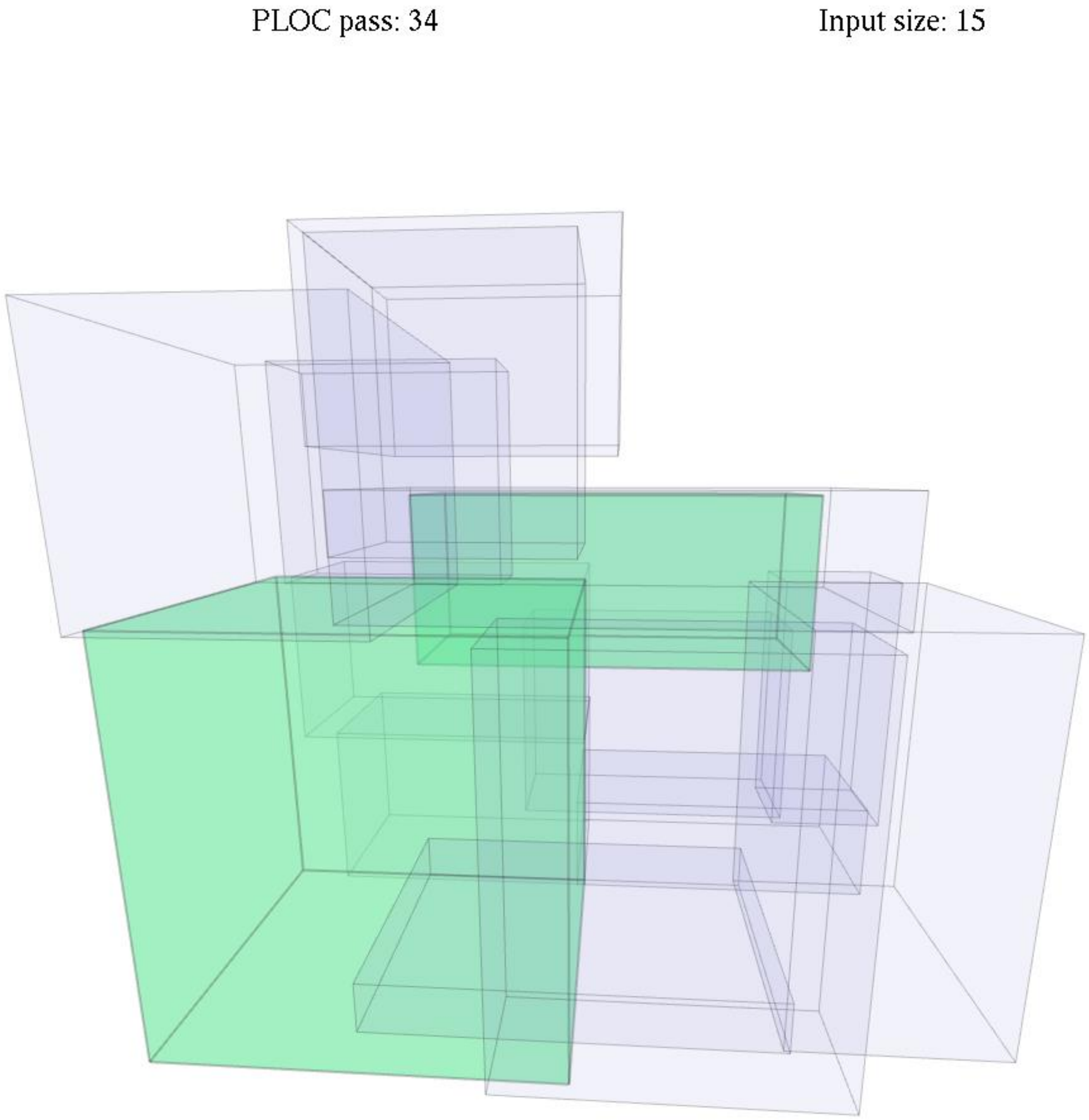
PLOC



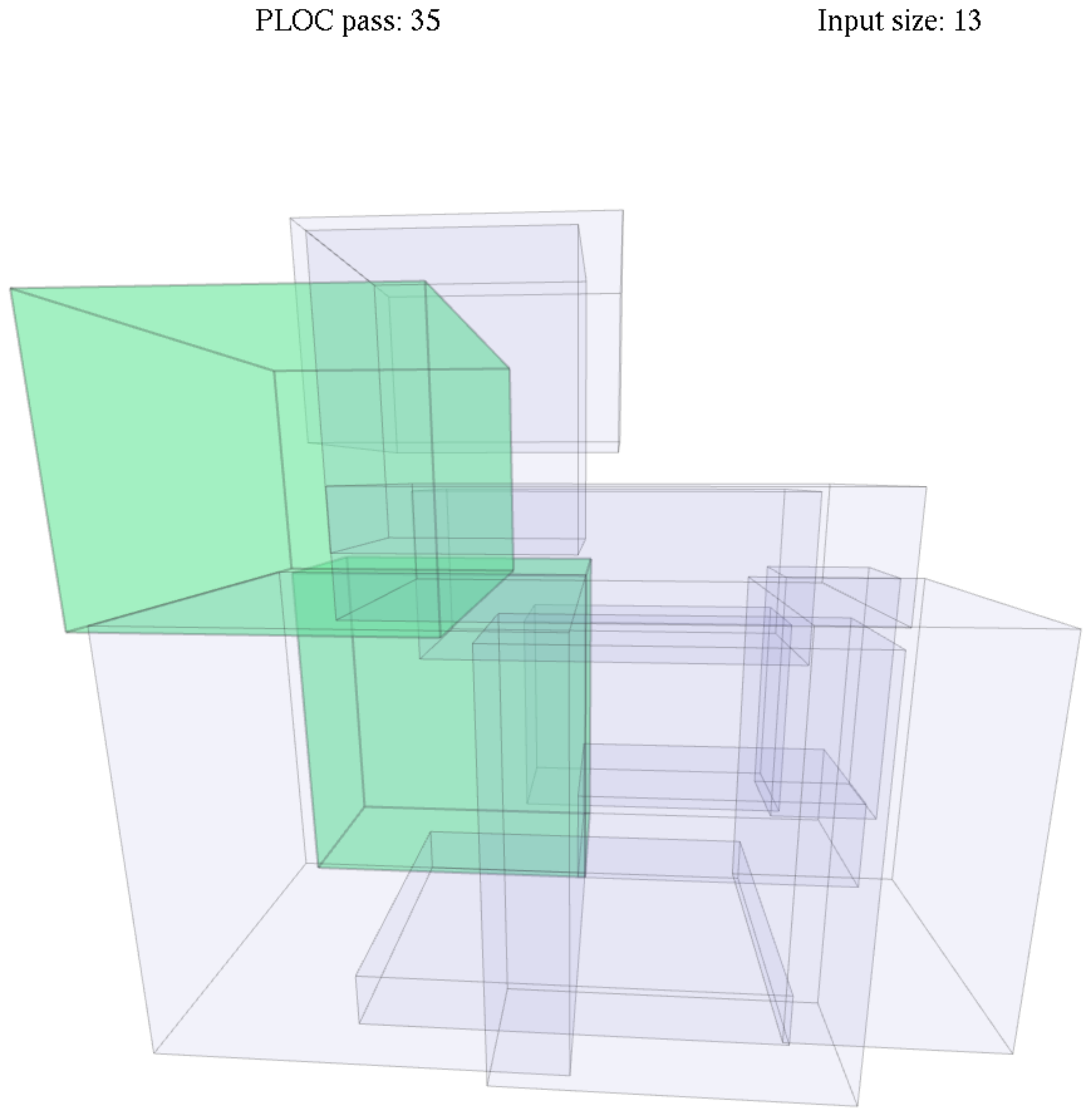
PLOC



PLOC



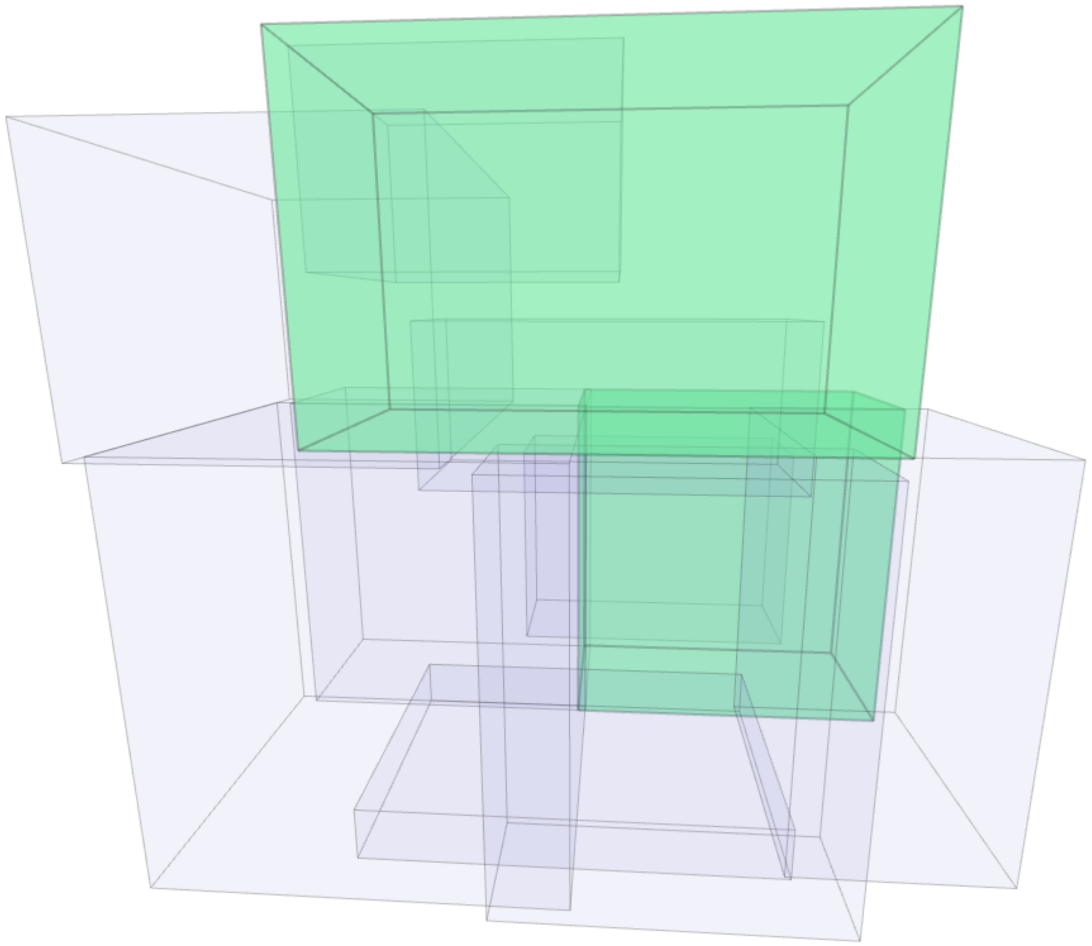
PLOC



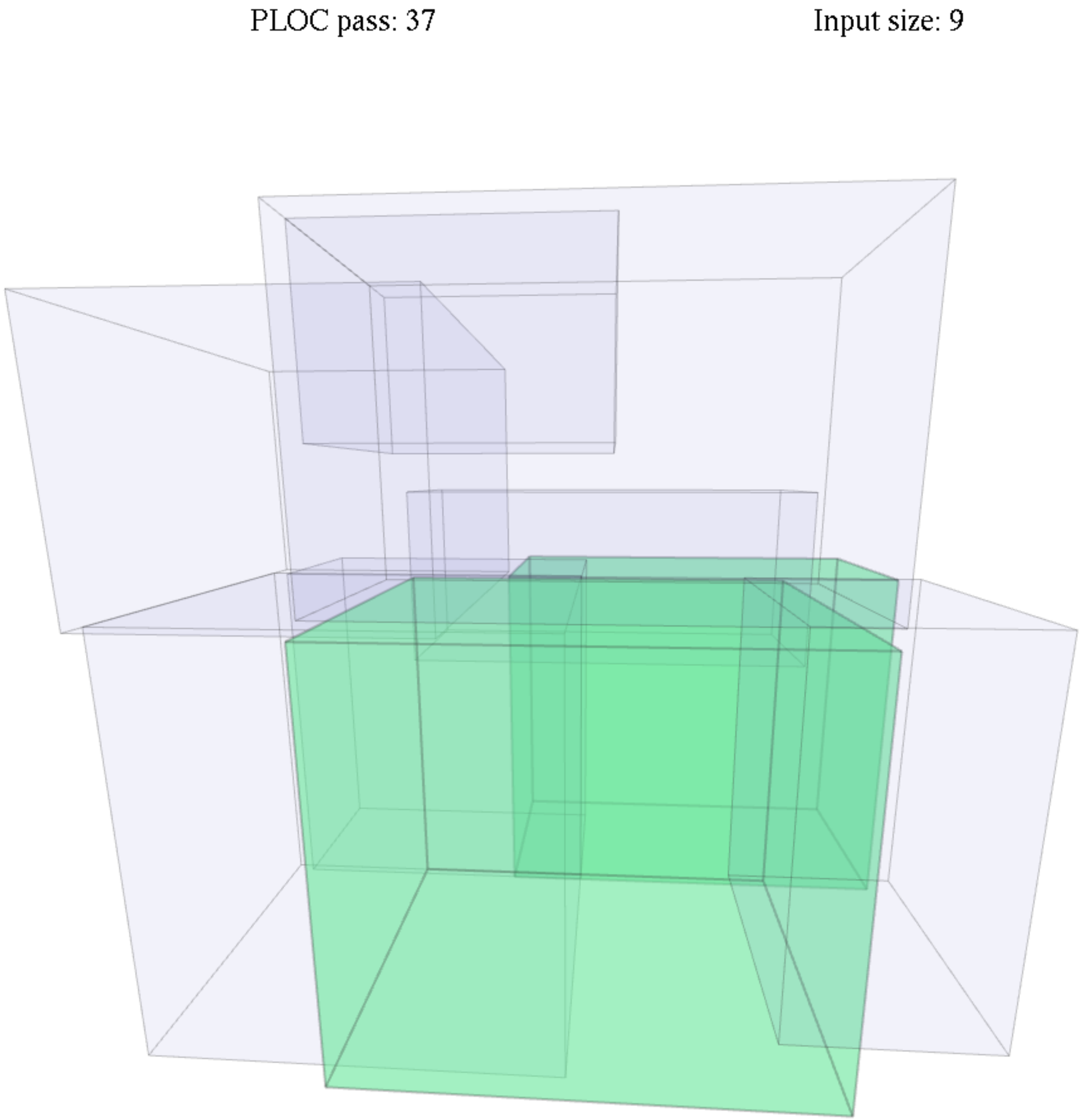
PLOC

PLOC pass: 36

Input size: 11



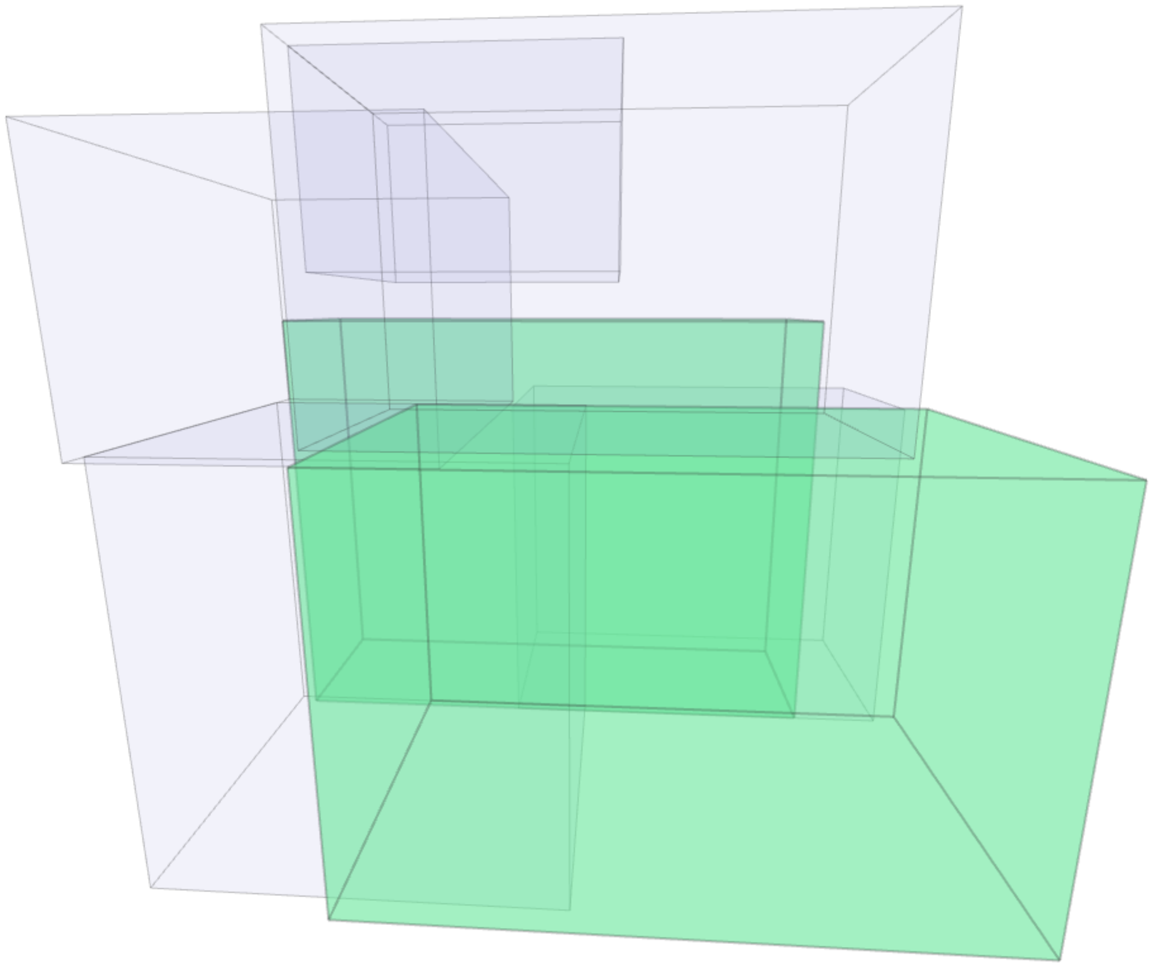
PLOC



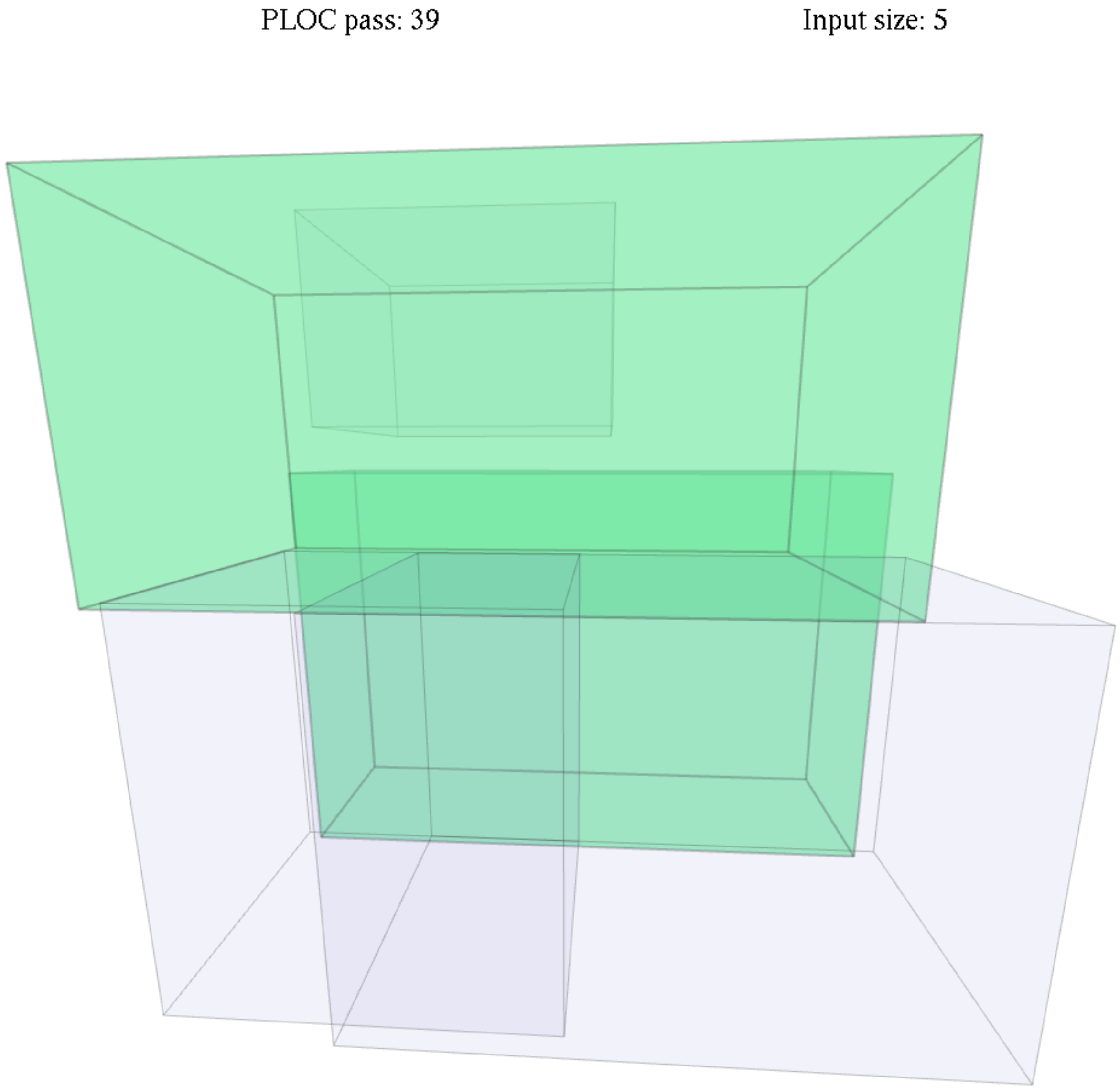
PLOC

PLOC pass: 38

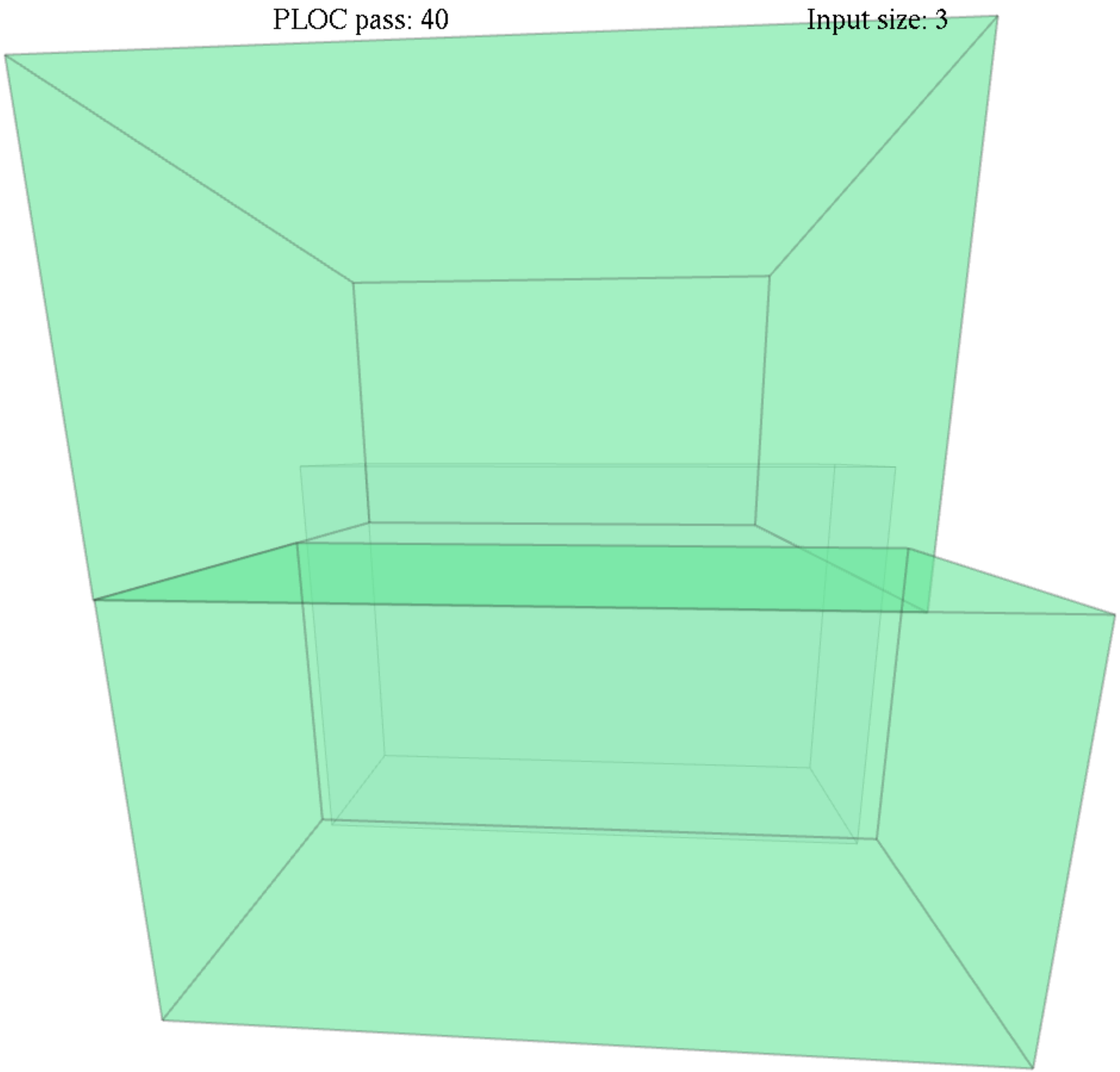
Input size: 7



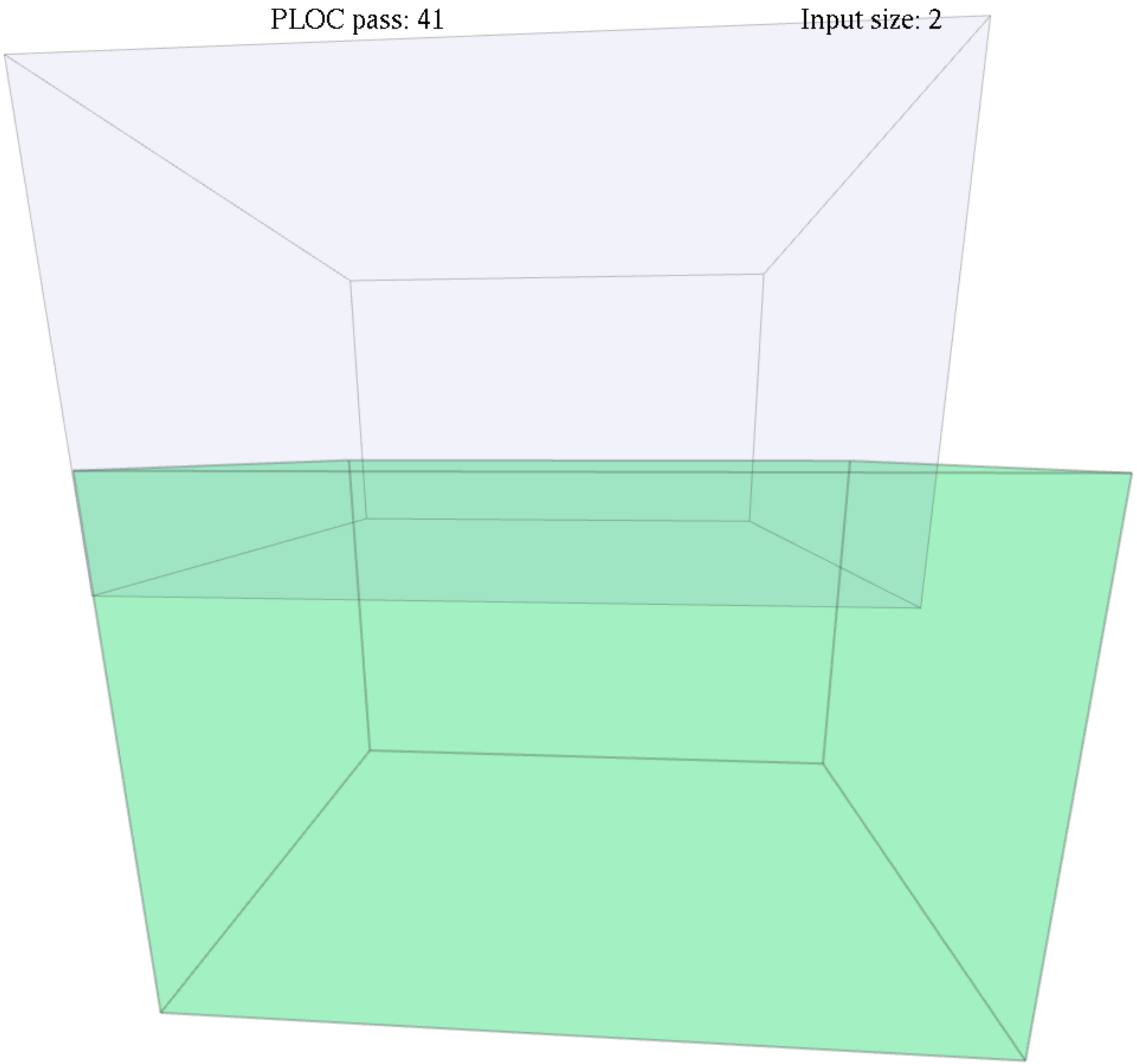
PLOC



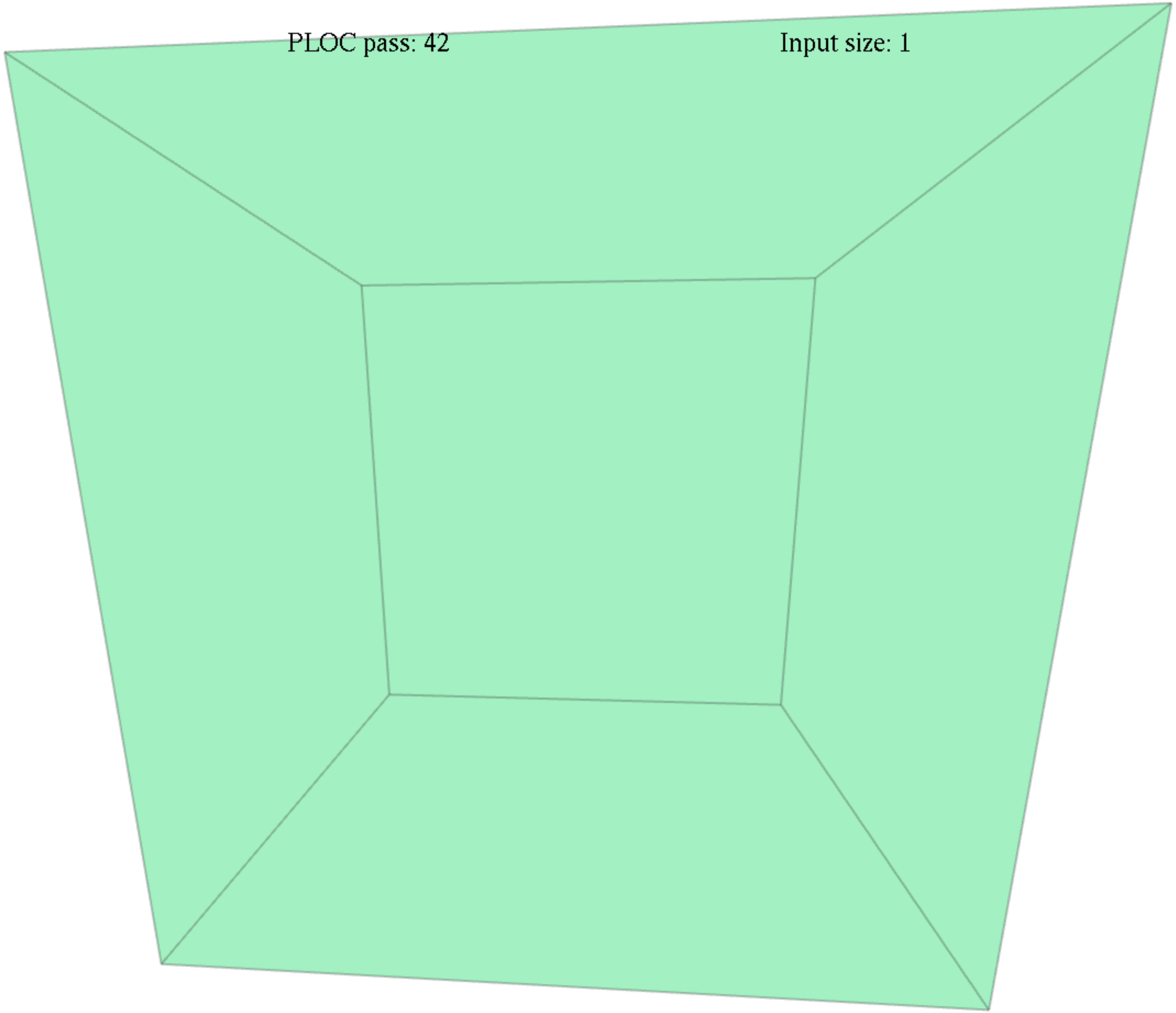
PLOC



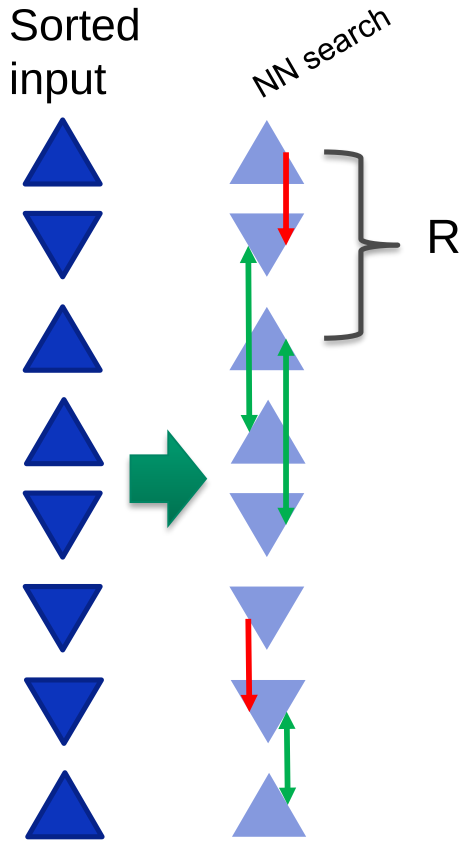
PLOC



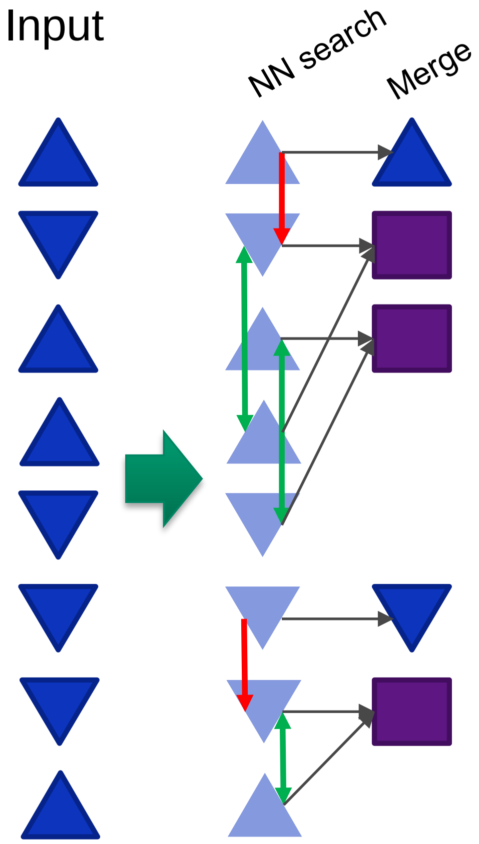
PLOC



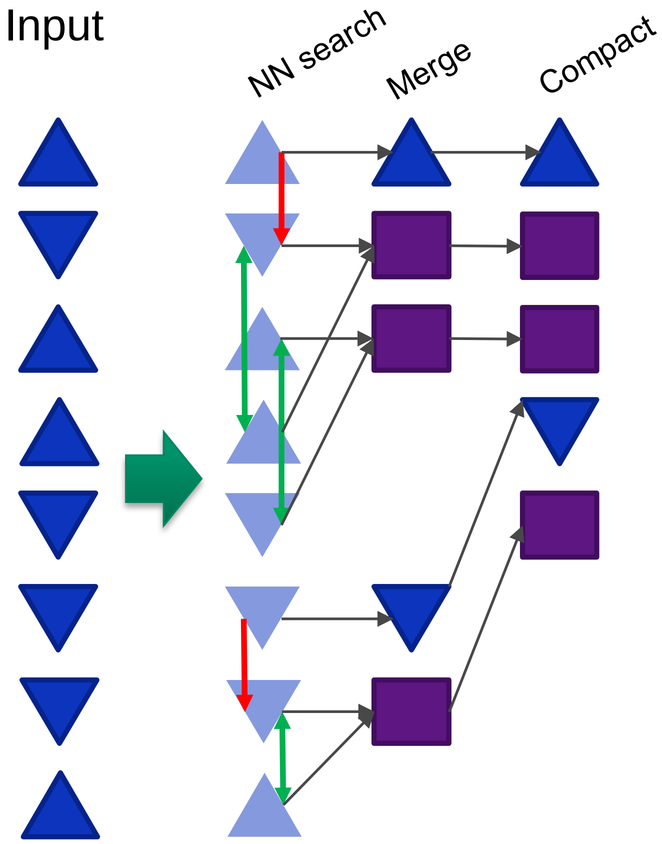
PLOC ALGORITHM (MEISTER AND BITTNER 2018)



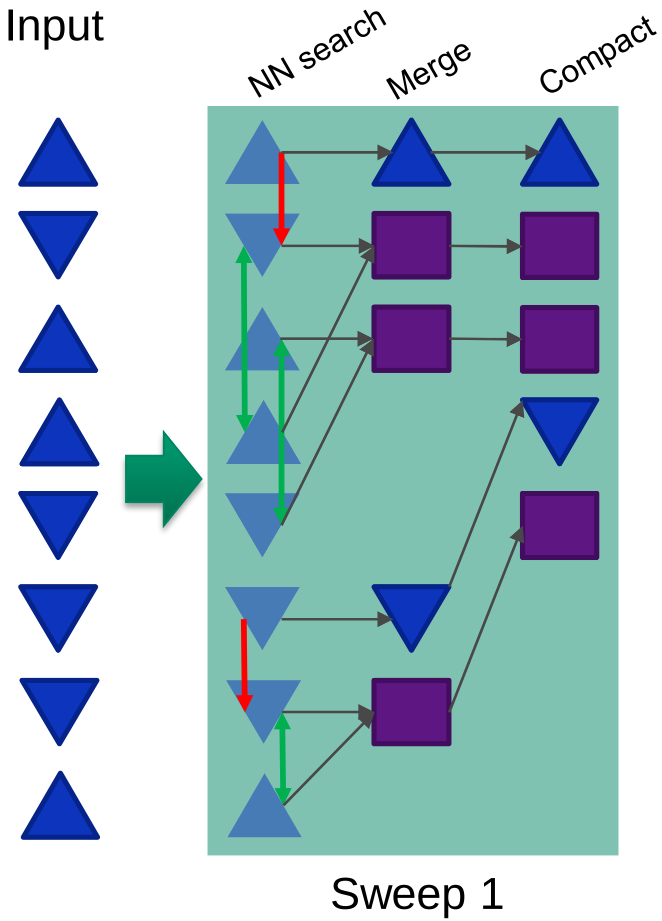
PLOC ALGORITHM (MEISTER AND BITTNER 2018)



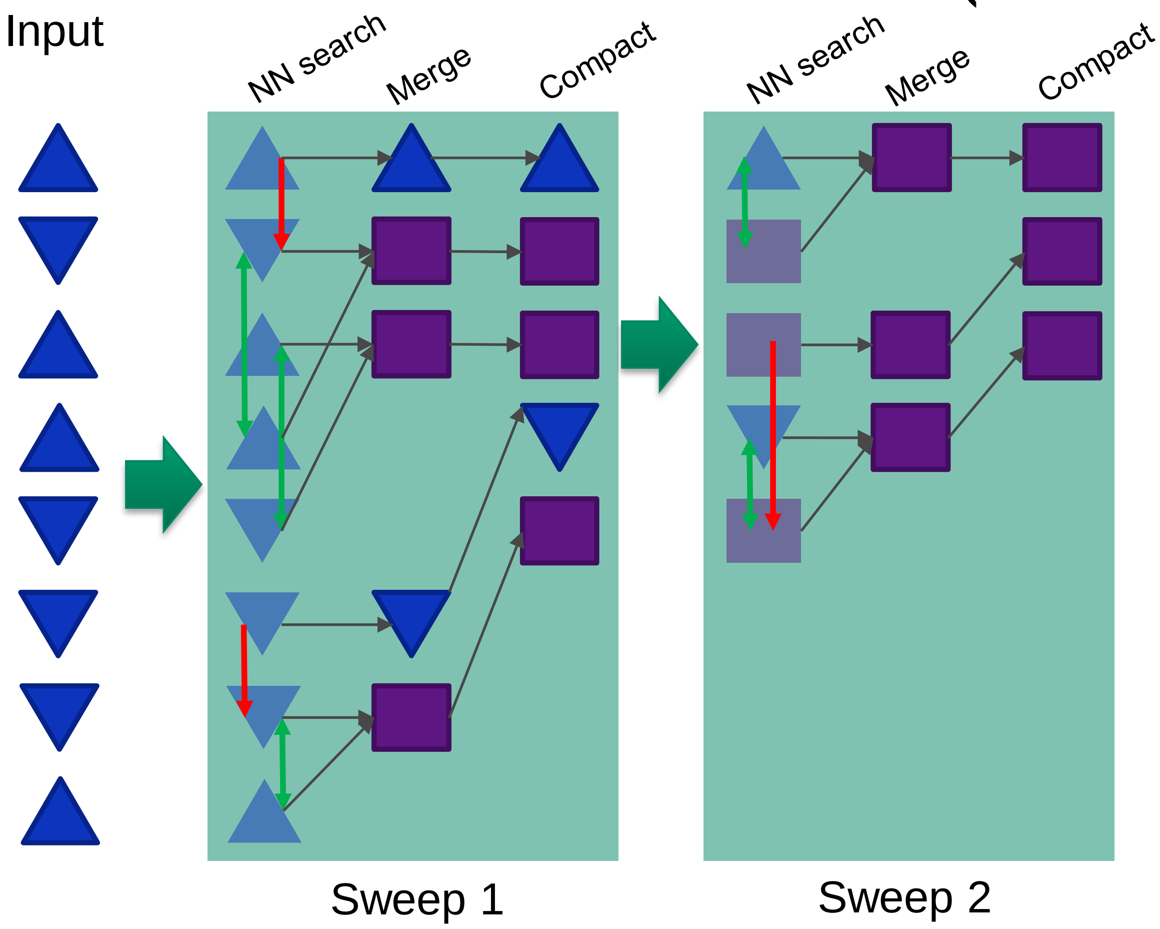
PLOC ALGORITHM (MEISTER AND BITTNER 2018)



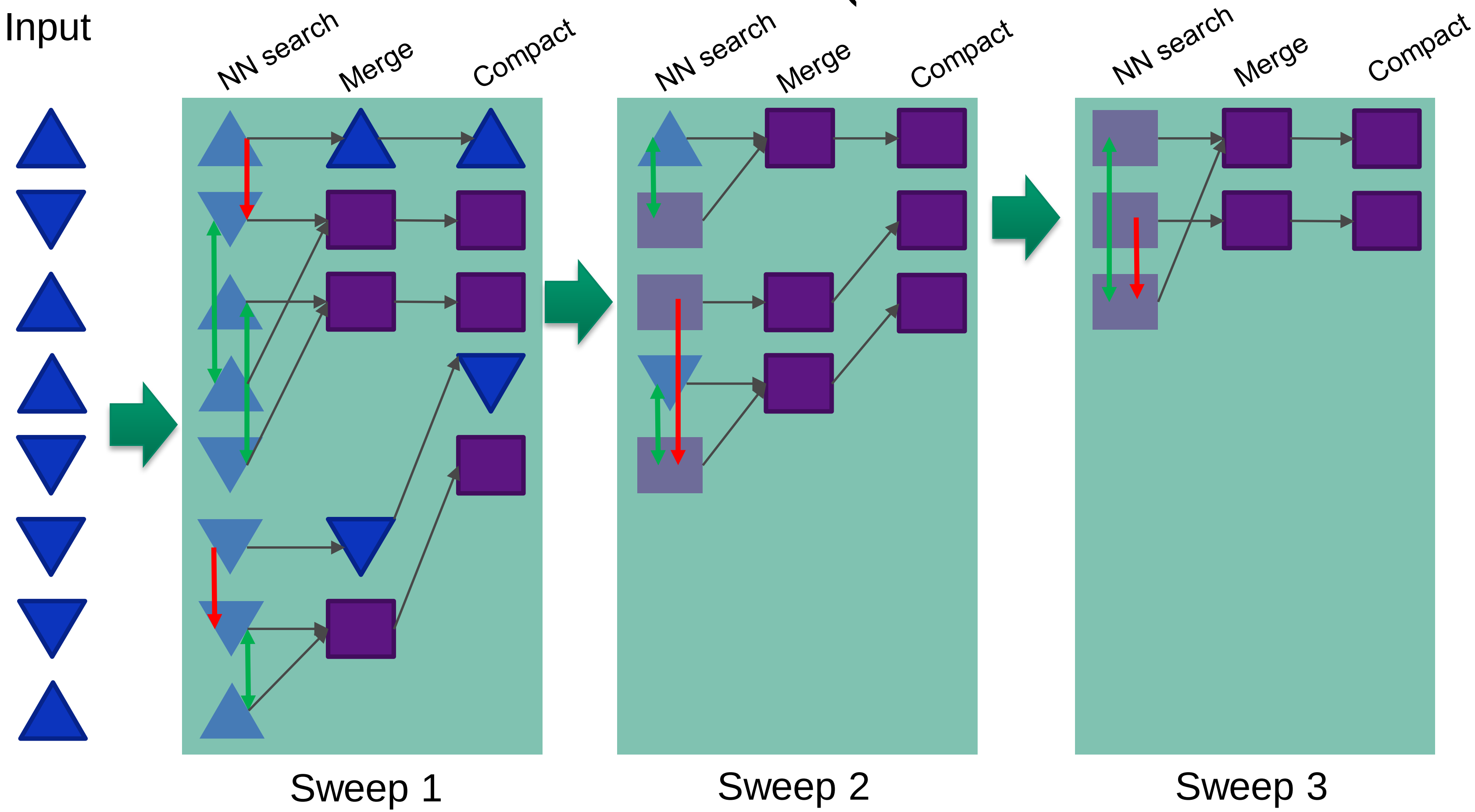
PLOC ALGORITHM (MEISTER AND BITTNER 2018)



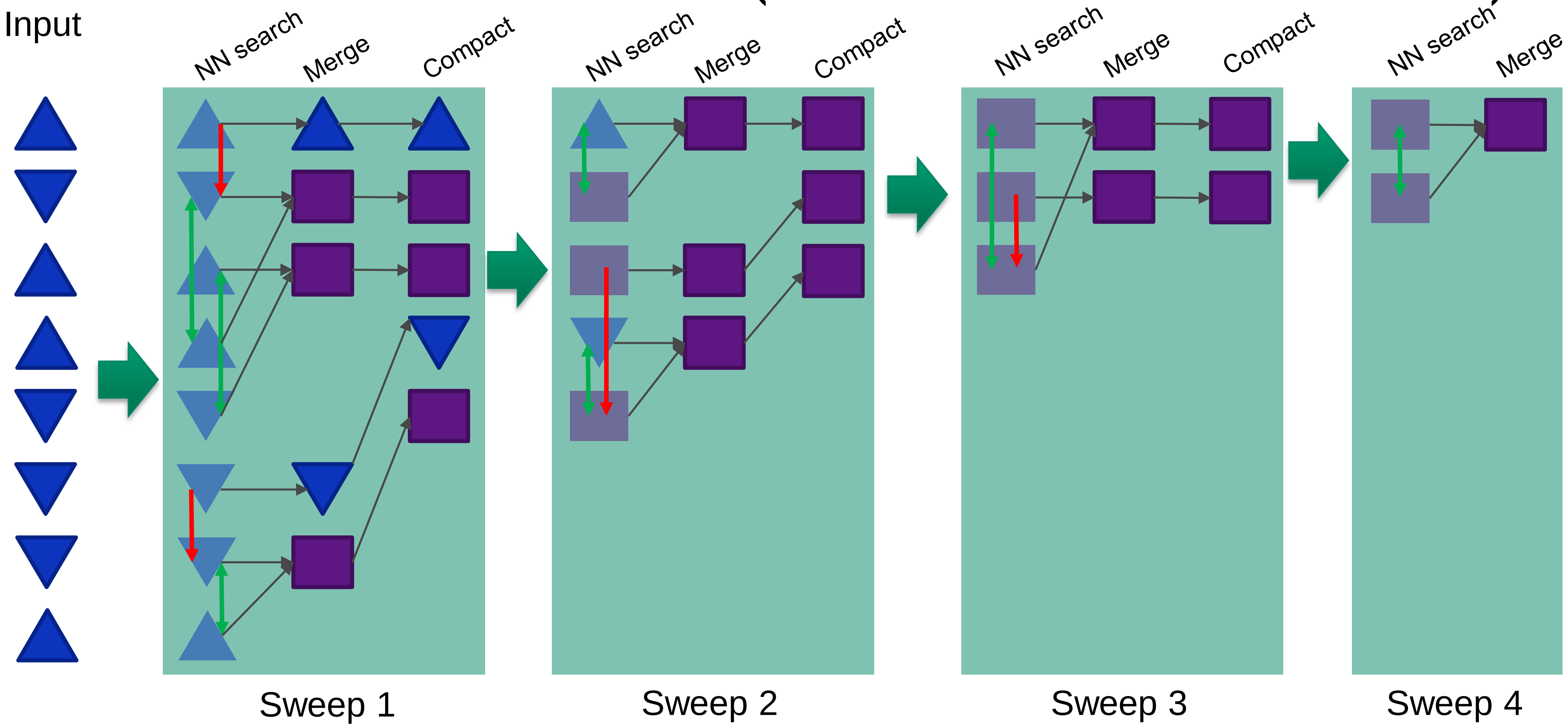
PLOC ALGORITHM (MEISTER AND BITTNER 2018)



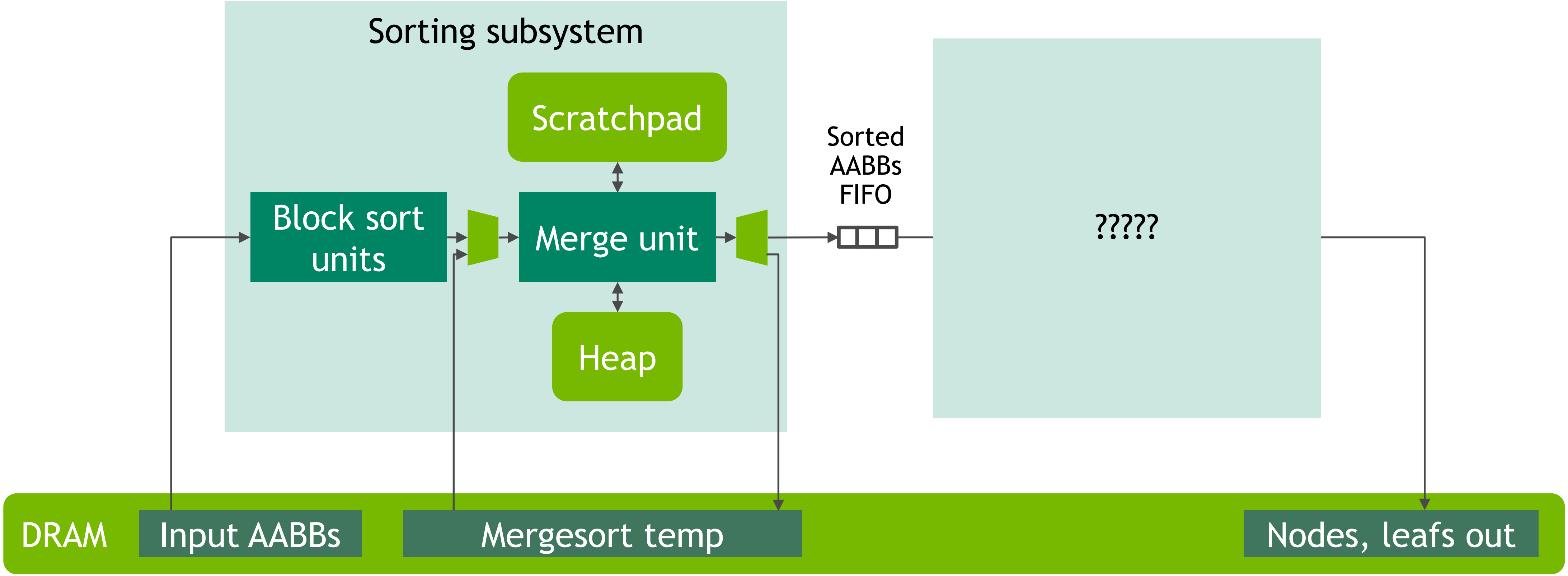
PLOC ALGORITHM (MEISTER AND BITTNER 2018)



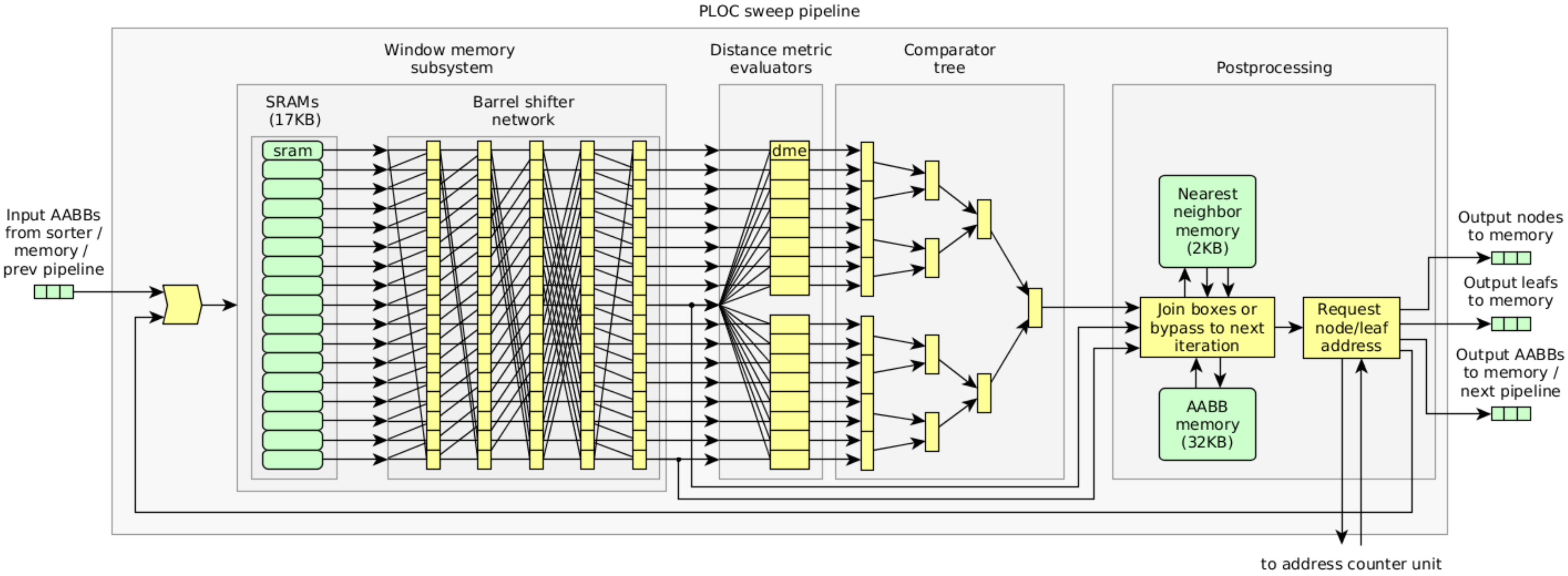
PLOC ALGORITHM (MEISTER AND BITTNER 2018)



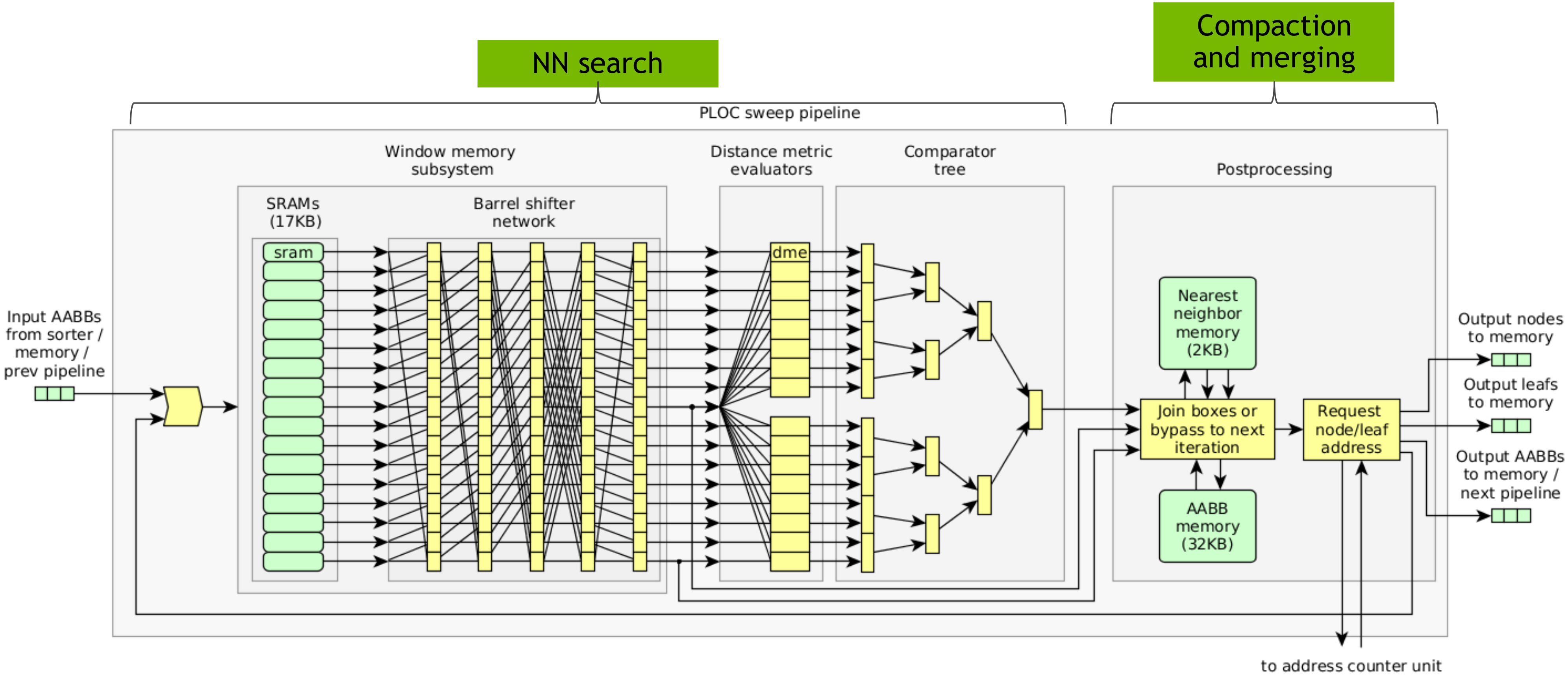
PLOCTREE ARCHITECTURE



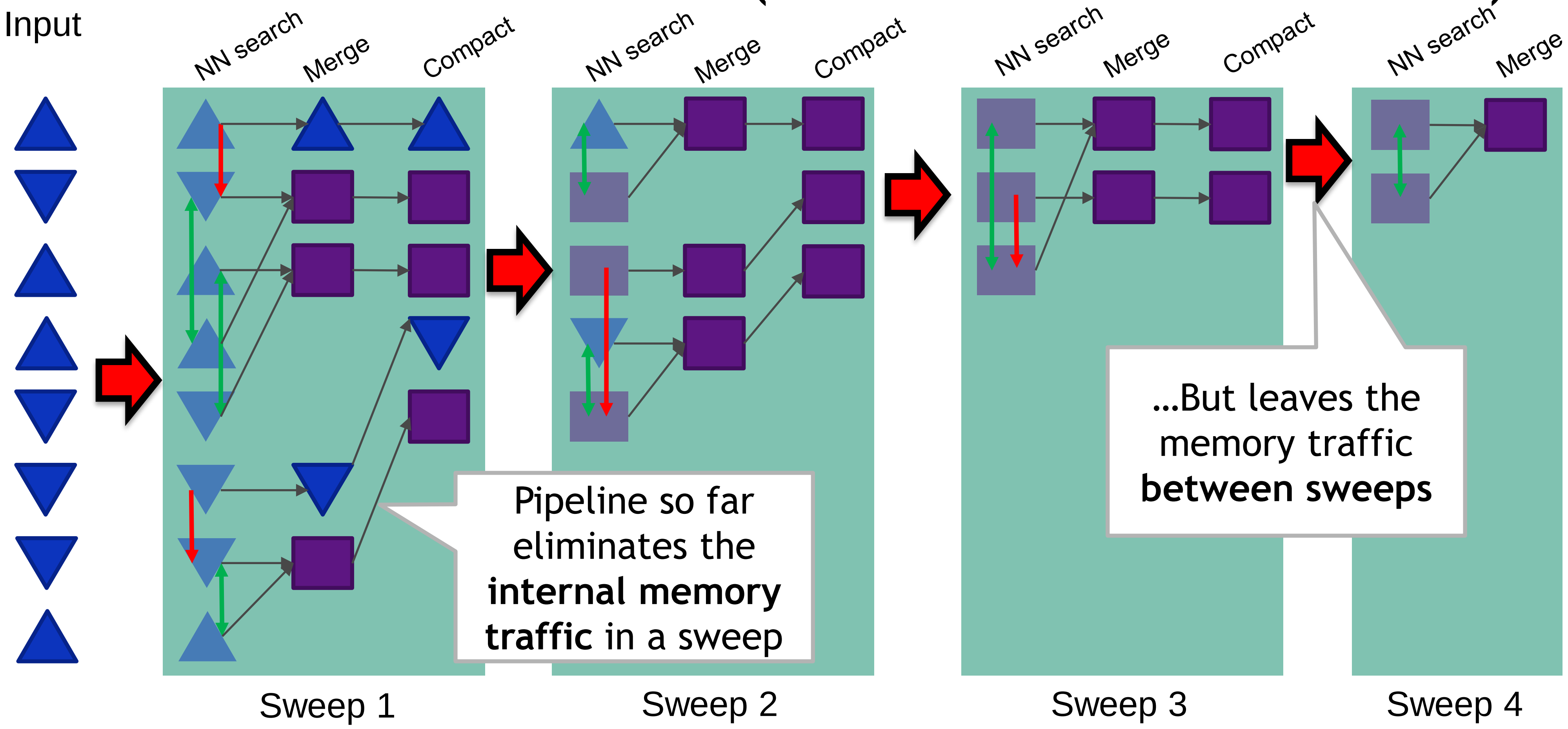
PLOC SWEEP PIPELINE


PLOC sweep pipeline
Window memory subsystem
SRAMs (17KB)
sram
Barrel shifter network
Distance metric evaluators
dme
Comparator tree
Postprocessing
Nearest neighbor memory (2KB)
Join boxes or bypass to next iteration
AABB memory (32KB)
Request node/leaf address
Input AABBs from sorter / memory / prev pipeline
Output nodes to memory
Output leafs to memory
Output AABBs to memory / next pipeline
to address counter unit

PLOC SWEEP PIPELINE



PLOC ALGORITHM (MEISTER AND BITTNER 2018)

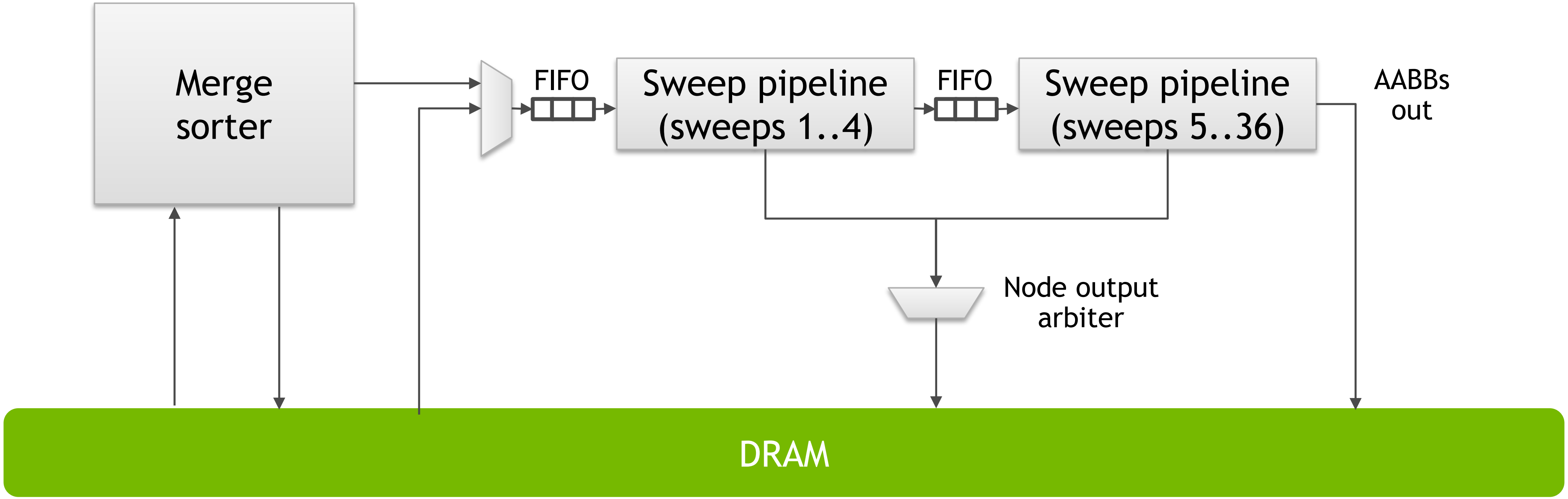


Pipeline so far eliminates the **internal memory traffic** in a sweep

…But leaves the memory traffic **between sweeps**

MULTIPLE PIPELINES

And multiple sweeps per pipeline



HW BVH BUILDER OVERVIEW

	Algorithm	Tree quality	Area (mm2) (scaled to 28nm)	Speedup (vs. GPU)	DRAM traffic savings (v. GPU)	Area efficiency	BW (GB/s)
(Doyle, 2013)	Binned SAH sweep	High	12.76	**9.4x** *	~2-3x **	14780% *	44
MergeTree	LBVH	Low	**1.77**	0.68x	3.3x	23435%	42.7
PLOCTree	PLOC	Medium	2.43	3.9x	**7.7x**	**97901%**	42.7
GTX 1080			610			100%	484

* Sopin (2011) on GTX 480

** HLBVH on GTX 480

Source: Source information is 8 pt, italic

OPEN PROBLEMS

OPEN PROBLEMS

Compressed BVHs: Incremental compression

Store ~5-6 bit coordinates *relative to parent bounding box* (Keely 2013, Vaidyanathan 2016)

Problem 1: Have to refit bottom-up, then compress top-down → more expensive refits

Tried to work around this in Viitanen (2017), but it only partly worked out

Problem 2: Nodes are small (8B) relative to cache lines (64B..128B)

→ Have to optimize node placement in cache lines for traversal perf (Liktor 2016)

Keely, *Reduced precision hardware for ray tracing*, HPG 2013

Vaidyanathan et al., *Watertight ray traversal with reduced precision*, HPG 2016

Liktor and Vaidyanathan, *Bandwidth-efficient BVH layout for incremental hardware traversal*, HPG 2016

Viitanen et al., *Fast hardware construction and refitting of quantized bounding volume hierarchies*, EGSR 2017

OPEN PROBLEMS

Compressed BVHs: The MBVH way

Compress wide BVHs; store coordinate origin and scale in each node

(Ylitie 2017, Vaidyanathan 2019)

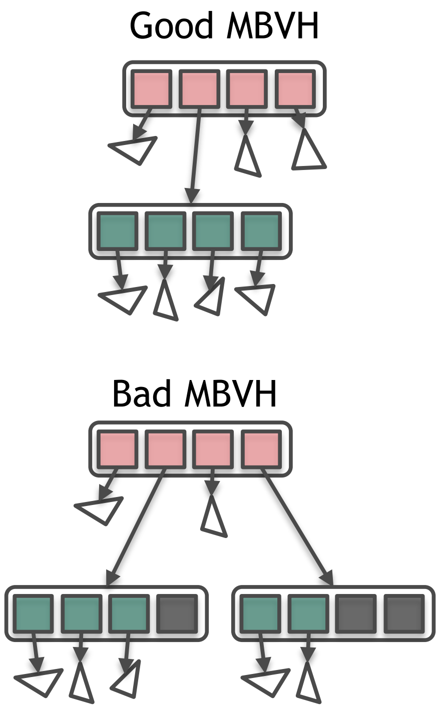
When shared between enough AABBs, compression ratio is still good

Nodes can be standalone and cache line sized

Problem: How to generate good MBVH layouts fast (even in SW)?

Similar to cache line opt. in incremental compression, but more constraints

…At least does not need to be done on refit



Ylitie et al., *Efficient incoherent ray traversal on GPUs through compressed wide BVHs*, HPG 2017

Vaidtanathan et al. Wide BVH Traversal with a Short Stack, HPG 2019

OPEN PROBLEMS

HW builder scaling

- Designs so far are serial pipelines
- Can parallelize by having multiple pipelines work on different BLASes, but
    - BLAS parallelism is limited and depends on workload
- Any way to collaborate on the same instance?

CONCLUSION

- SW BVH construction is fast enough for AAA games with RT effects and getting faster
  - But does need some dev effort to get there (e.g. asynch overlapping, geometry culling)
  - And has some corner cases where it’s easy to hit traversal slowdowns (sharp triangles, loose instance grouping, refit from degen)
- BVH hardware might give a speedup, but big hurdles left to clear, mainly:
  - Compressed BVH output
  - Scaling to multiple pipelines
- → We aren’t done yet
- RTX enabled games look like a gold mine for researchers; very different workloads from classic builder benchmarks

NVIDIA®