Raytracing Sparse Volumes with NVIDIA® GVDB in DesignWorks

Rama Hoetzlein, NVIDIA Graphics Technologies, Professional Graphics

Agenda

1. Goals
2. Interactive Demo
3. Design of NVIDIA® GVDB
4. Using GVDB in Practice
5. Results
6. Resources & Availability

INTRODUCING
NVIDIA® GVDB AT SIGGRAPH 2016

Part of the DesignWorks ecosystem

NVIDIA® GVDB
Wednesday, 2:30pm
at NVIDIA Booth theater

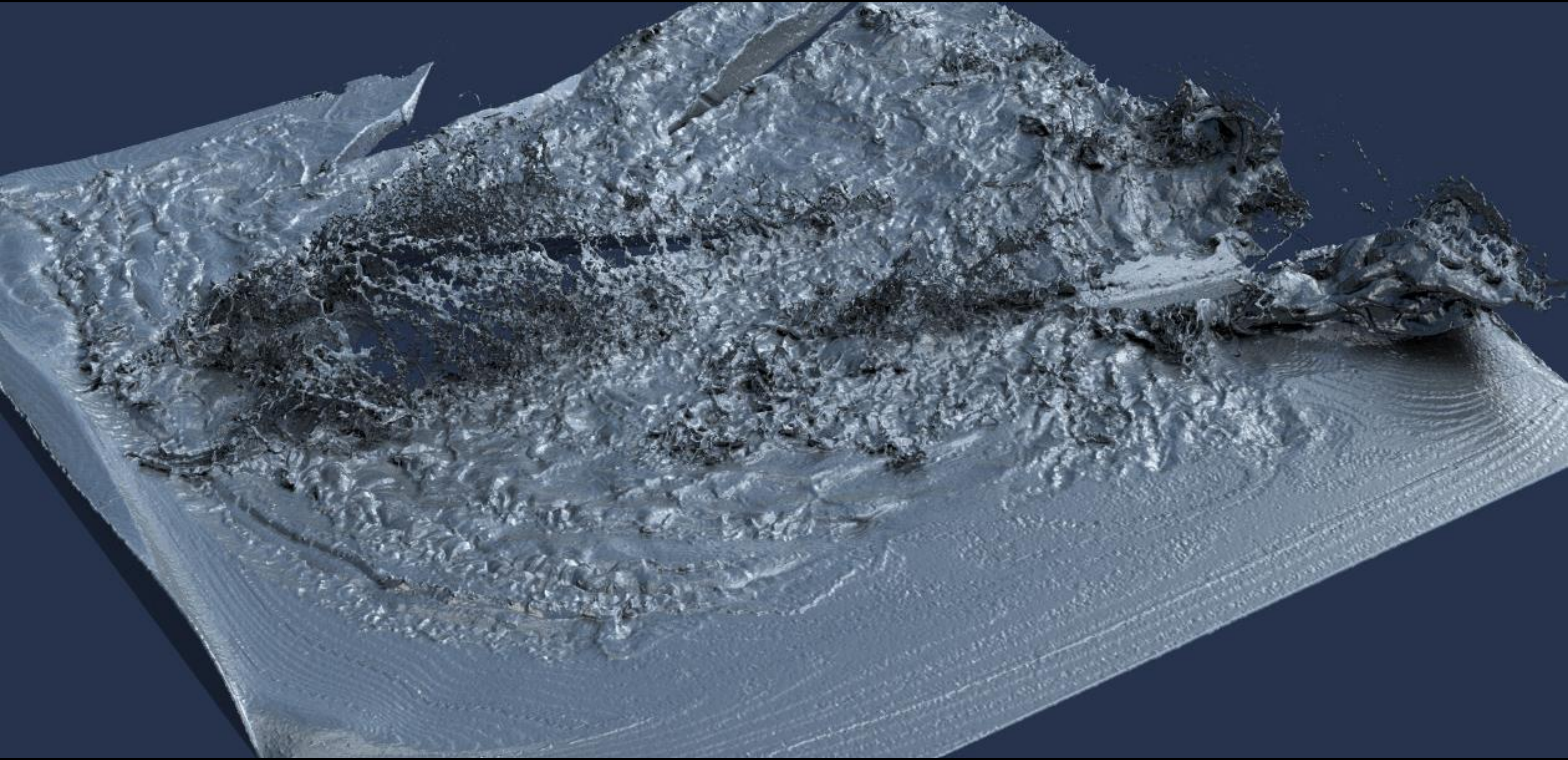
with Ken Museth, Lead Developer of OpenVDB

Goals

Motion Pictures

Increasing detail and complexity..

Property of DreamWorks Animation



Rendered in GVDB

Data Property of DreamWorks Animation

Goals:

“Data structures for dynamics must allow for both the grid values (e.g., simulation data) and topology (e.g., sparsity of values), to vary over time.” - Museth 2013

- Uncompressed scalar values
- Dynamic values *and* topology
- All in memory (out of core optional)
- Efficient compute on GPU
- High quality, efficient raytracing on GPU

Design of NVIDIA® GVDB

Representing Large Volumes

Dense Volumes

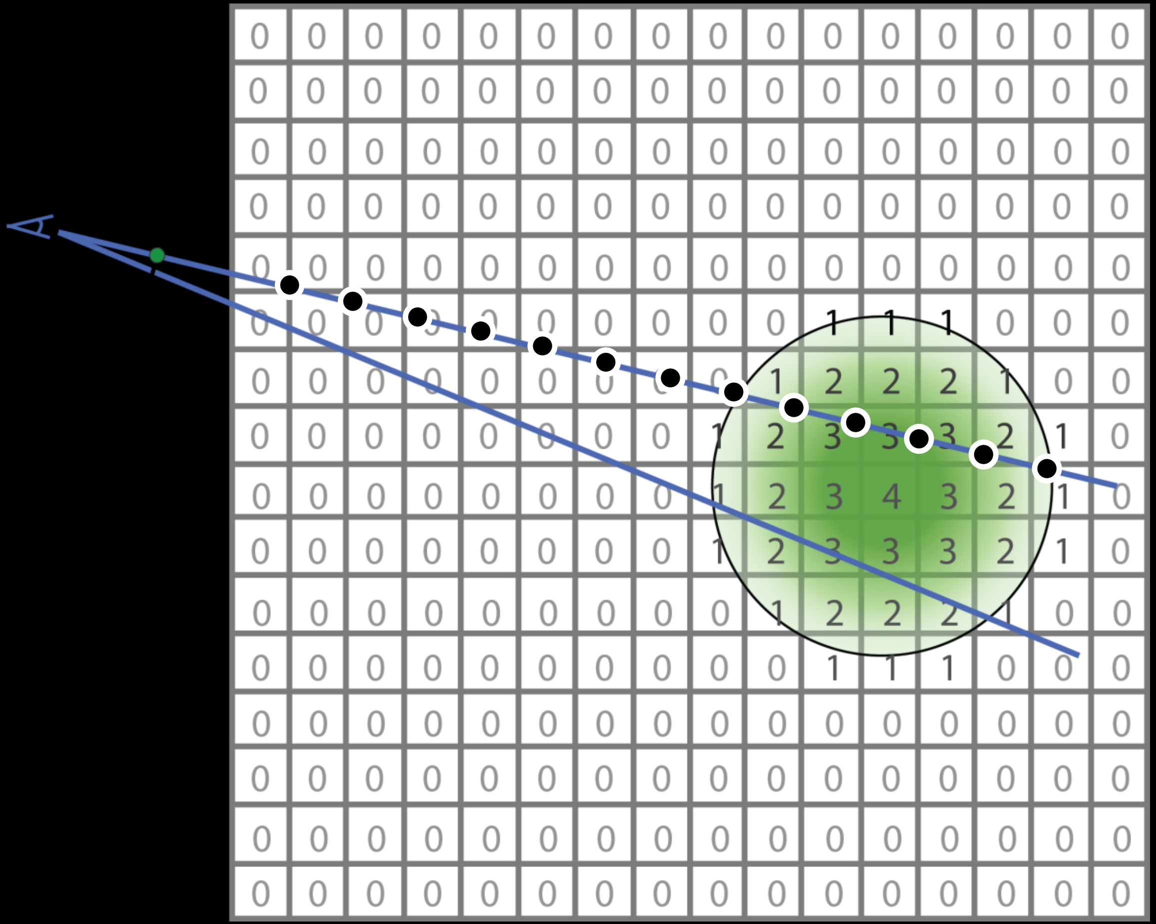
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0
0	0	0	0	0	0	0	0	0	1	2	2	2	1	0	0
0	0	0	0	0	0	0	0	1	2	3	3	3	2	1	0
0	0	0	0	0	0	0	0	1	2	3	4	3	2	1	0
0	0	0	0	0	0	0	0	1	2	3	3	3	2	1	0
0	0	0	0	0	0	0	0	0	1	2	2	2	1	0	0
0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

16 x 16 =

256 data values

Representing Large Volumes

Dense Volumes

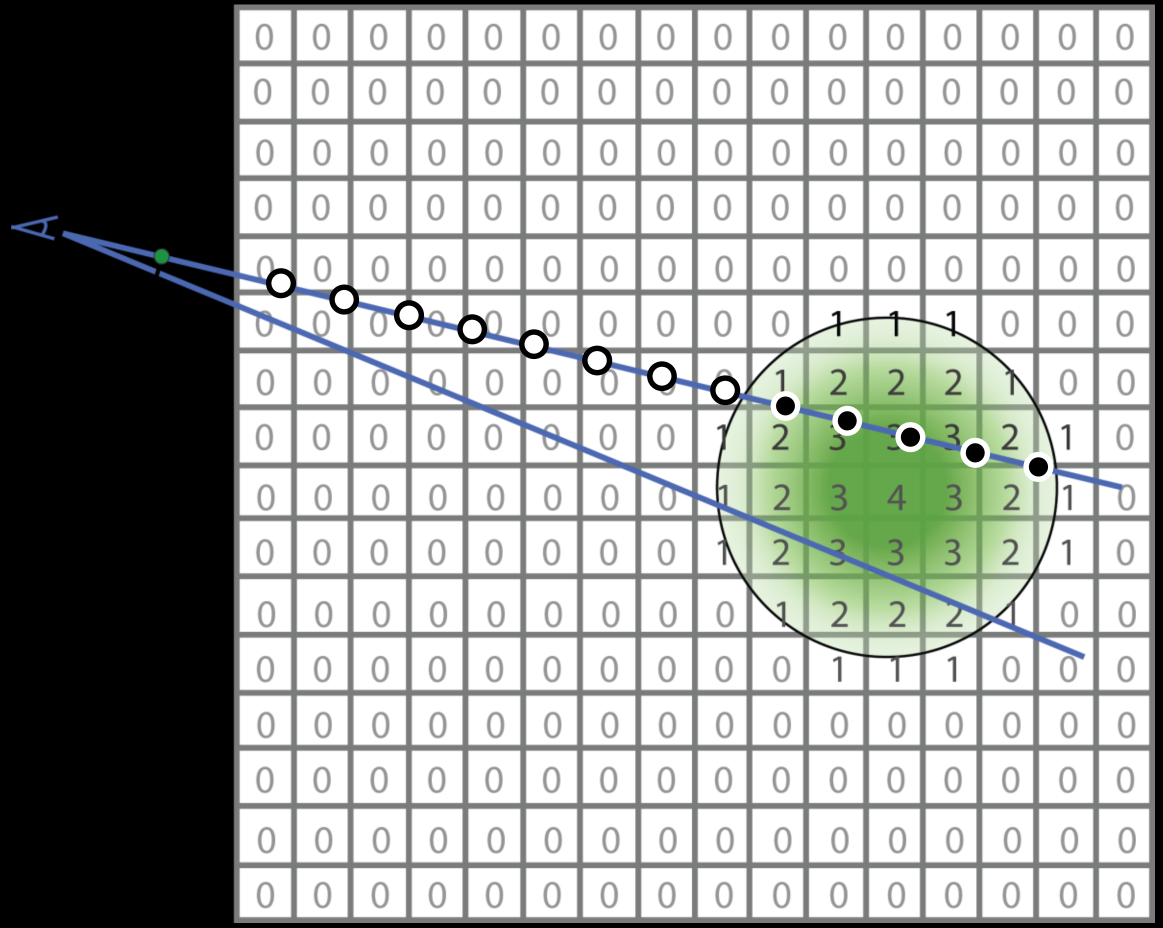


16 x 16 =

256 data values

Representing Large Volumes

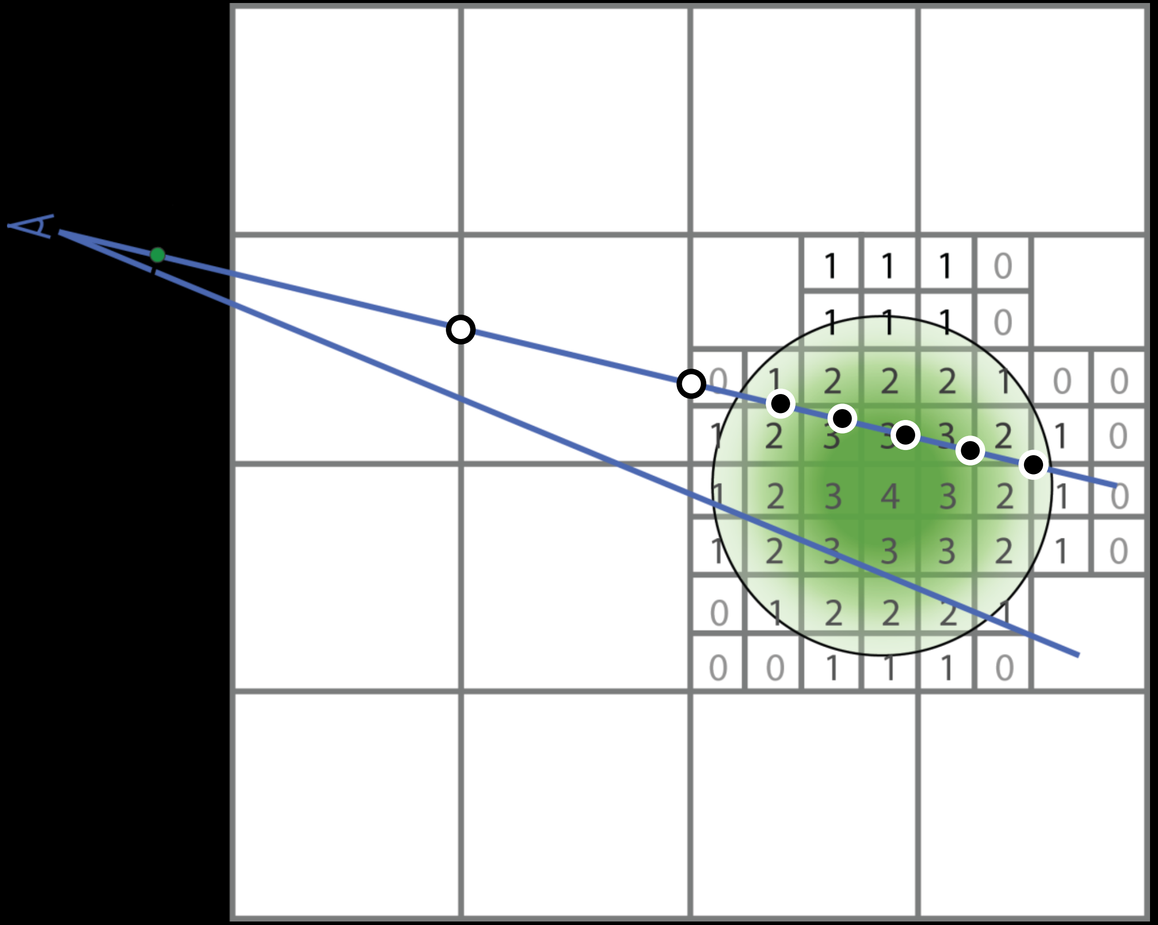
Dense Volumes



- 8 empty steps
- 5 active steps

Representing Large Volumes

Sparse Volumes



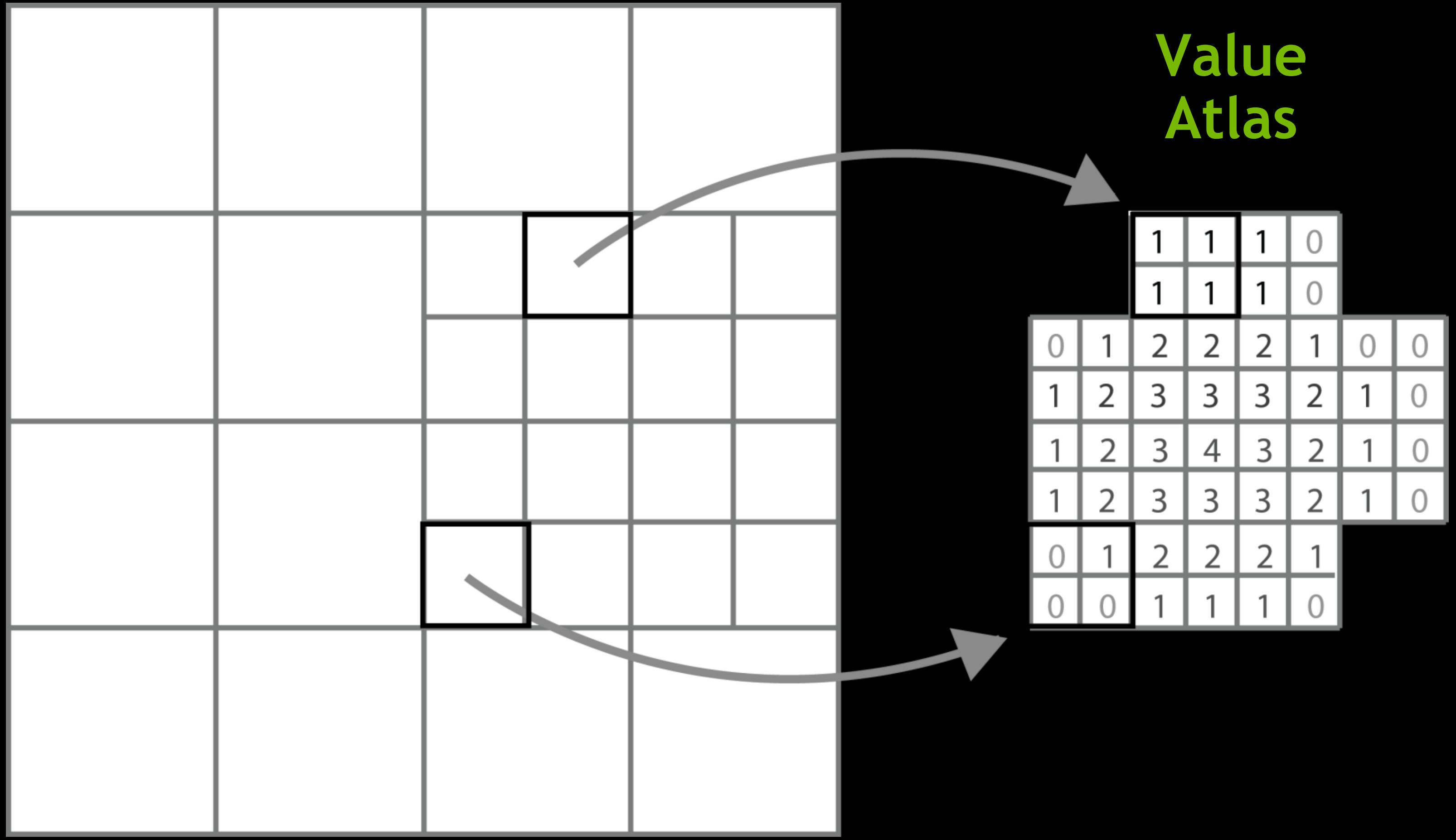
52 data values

(instead of 256!)

- ● 2 DDA skip steps
- ○ 5 sample steps

Representing Large Volumes

Topology



Methods for Sparse Volumes

Meshes & Point Clouds

Binary Voxels

Kampe, 2013	acyclic DAGs
Niessner, 2013	voxel hashing (SDF)
Reichl, 2014	voxel hashing
Villanueva, 2016	graph similarities

Meshes

Laine, 2010	sparse voxel octrees
Chajdas, 2014	sparse voxel octrees
Reichl, 2015	fragment buffers

Isosurfaces

Hadwiger, 2005	complex shaders
Knoll, 2009	multi-res surfaces

Volumetric Data

Octrees

Boada, 2001	texture-based octree
Crassin, 2008	gigavoxels

Tilemap Grids

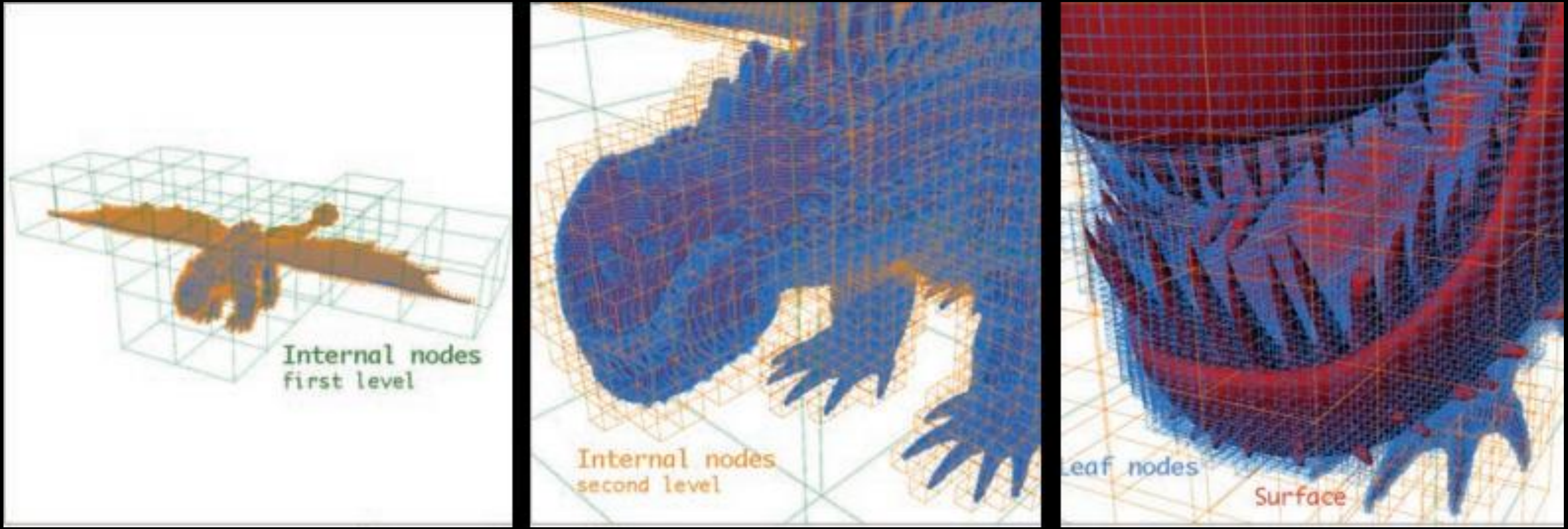
Hadwiger, 2012	per-sample, out-of-core
Fogal, 2013	index table, out-of-core

VDB Grids

Museth, 2013	hierarchy of N-ary grids

OpenVDB

Hierarchy of Grids



Ken Museth, VDB: High-resolution sparse volumes with dynamic topology, Transactions on Graphics, 2013

OpenVDB

Voxel Database Structure

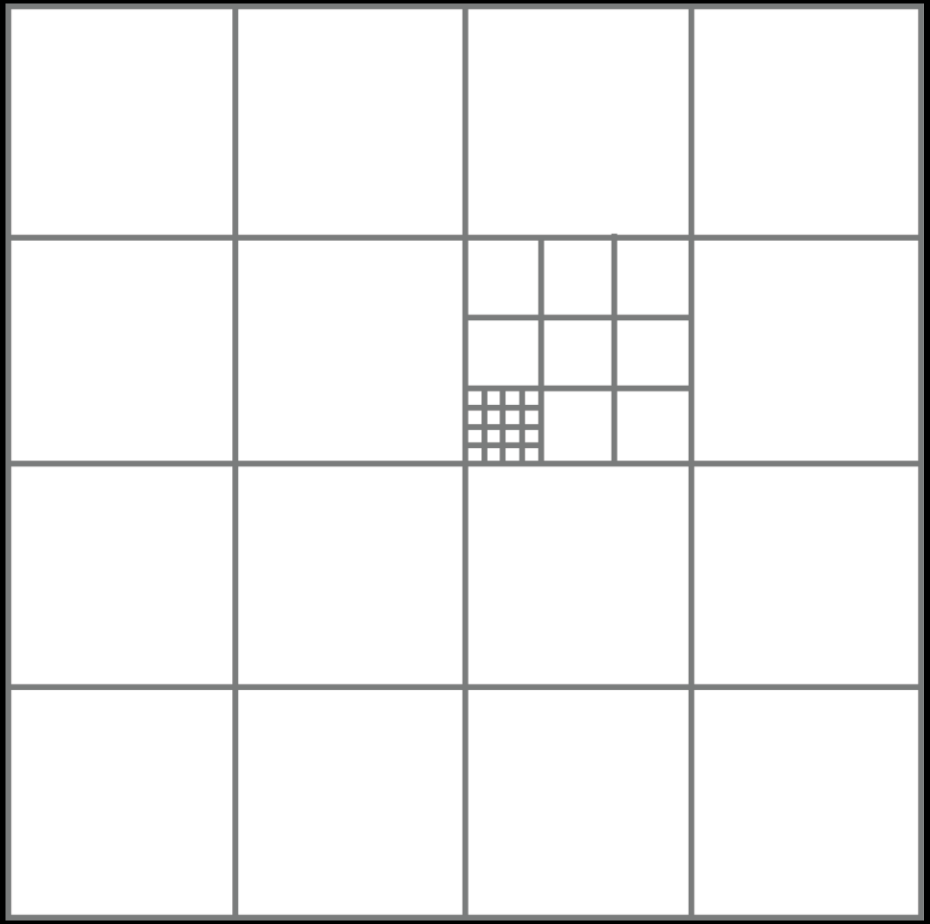
Hierarchy of Grids

Many levels

Each level is a grid

Each level has its *own* resolution

e.g. top = 4x4
mid = 3x3
brick = 4x4



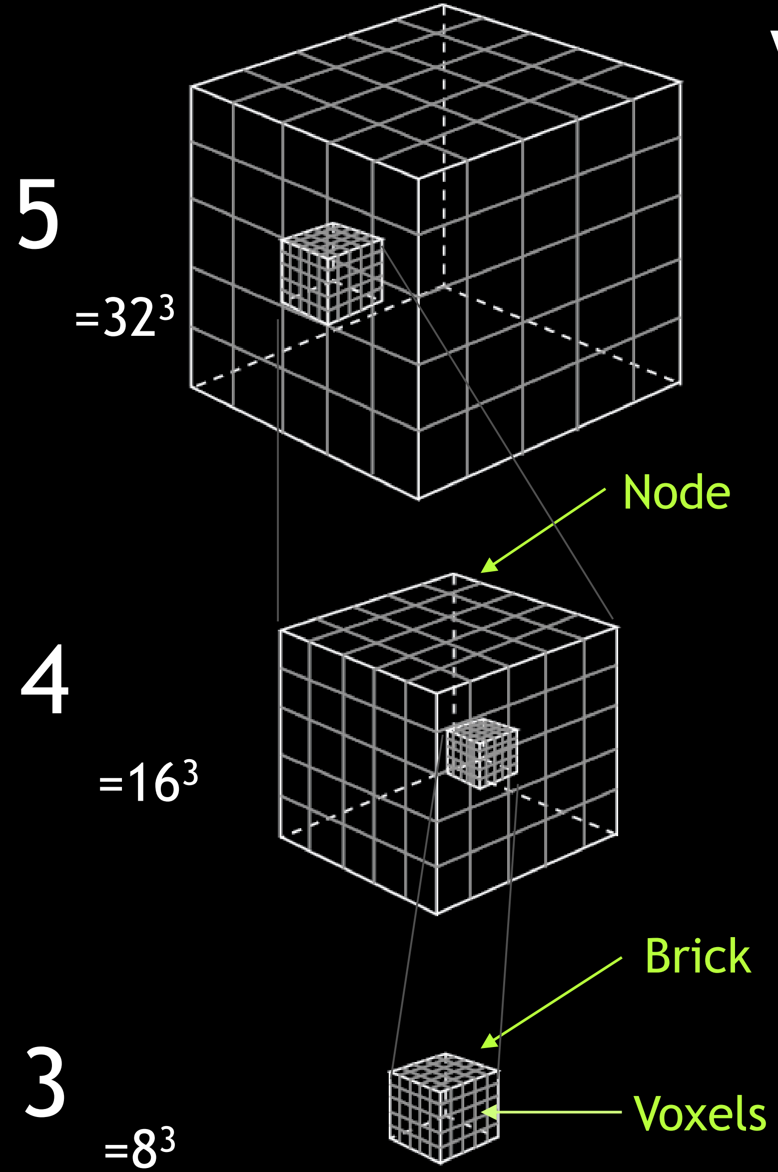
Key features:

Can store
very large volumes
with only a ***few* levels.**

Efficient to traverse,
since every level is a grid.

Voxel Database Structure

Hierarchy of Grids



VDB Configuration.

Each level is defined by its Log2 dimension.

$< L_N, .., L_2, L_1, L_0 >$ L_0 = Brick dim

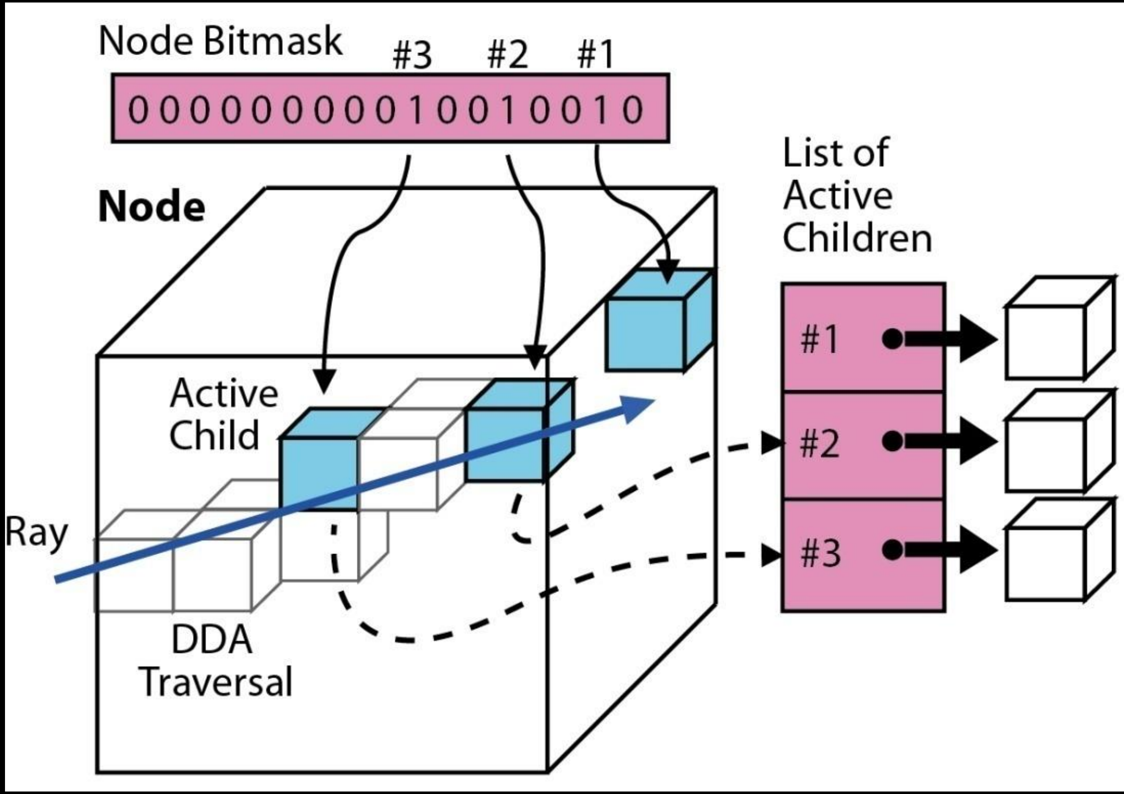
Examples:

Log2 Dims	**Tree Type**
<1, 1, .., 1>	Octree
<10, 2>	Tile map
<*, 2>	Hash map
<5,4,3>	OpenVDB
<3,3,3,4>	GVDB

Voxel Database Structure

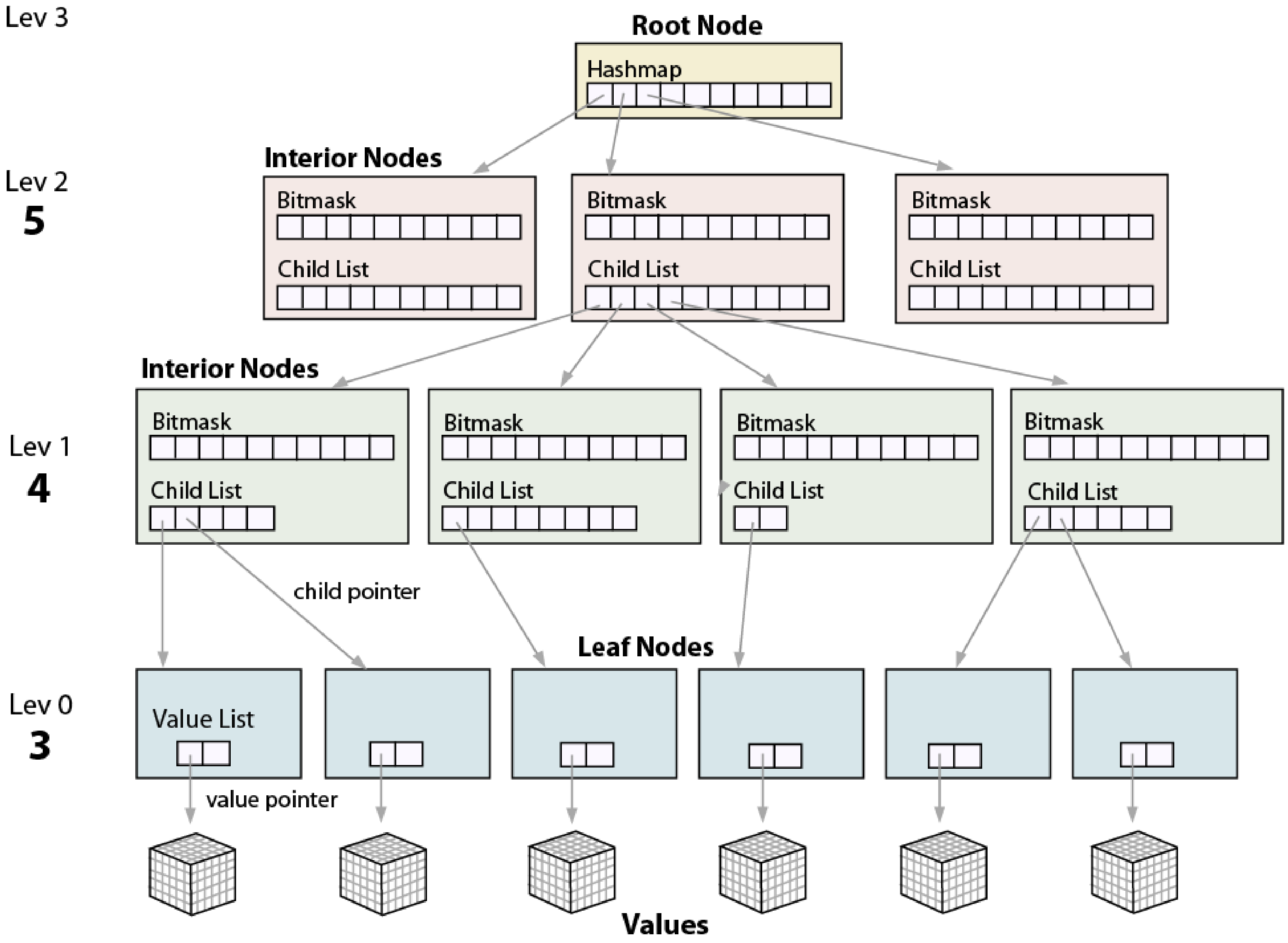
Hierarchy of Grids

How are sparse grids stored?

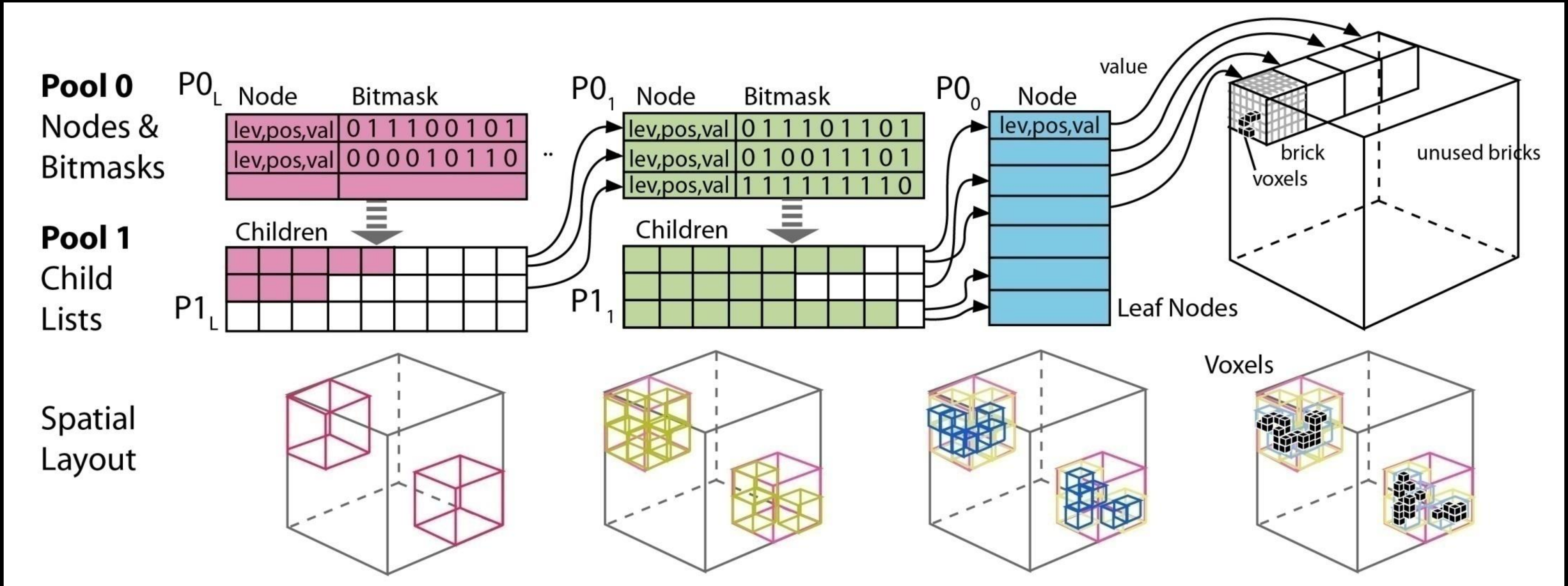


VDB:
Hierarchy of voxel grids,
where *active* children are enabled using
a bitmask for pointer indirection.
Inactive nodes and bricks
are not stored.

OpenVDB - Memory Layout



NVIDIA® GVDB Sparse Volumes



Sequence of node pools
Pool 0: List of node data and active bitmasks
Pool 1: List of active children

Benefits:
- Run-time config, Dynamic, Fast

Compared to OpenVDB:
- No host or device pointers
- Identical data on CPU and GPU
- Eliminate root, interior, leaf classes
- Eliminate templating
- Eliminate per-voxel iterators

NVIDIA® GVDB Sparse Volumes

Key Features:

- Identical spatial layout and numerical values as VDB grid
- Run-time tree configuration
- Memory pooling for efficient topology changes
- Identical data on CPU & GPU
- Fast raytracing and compute on GPU

Using NVIDIA® GVDB in practice

Compute Operations

Ideal GPU Kernels

Ideal Stencil kernels:

```
v =    tex3D ( p.x-1, p.y    ,  p.z );
v +=  tex3D ( p.x+1, p.y    ,  p.z );
v +=  tex3D ( p.x    , p.y-1  ,  p.z );
v +=  tex3D ( p.x    , p.y+1 ,  p.z );

surf3Dwrite ( volTex, v,  p.x, p.y, p.z );
```

No conditionals

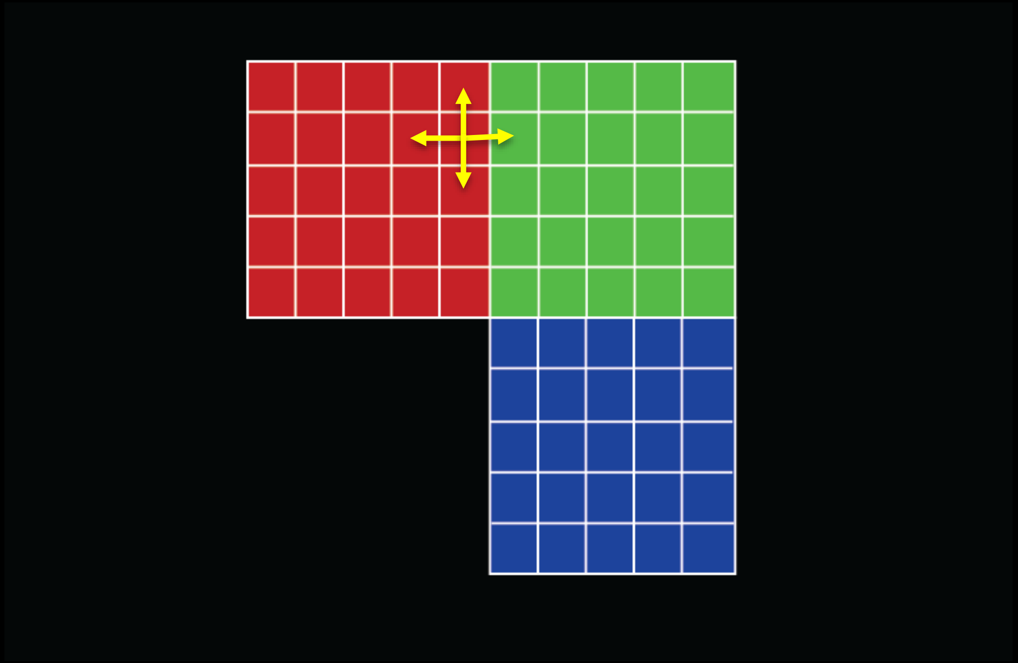
Neighbors directly accessed

Balanced workload on all voxels

In-place operation

Compute Operations

What to compute..



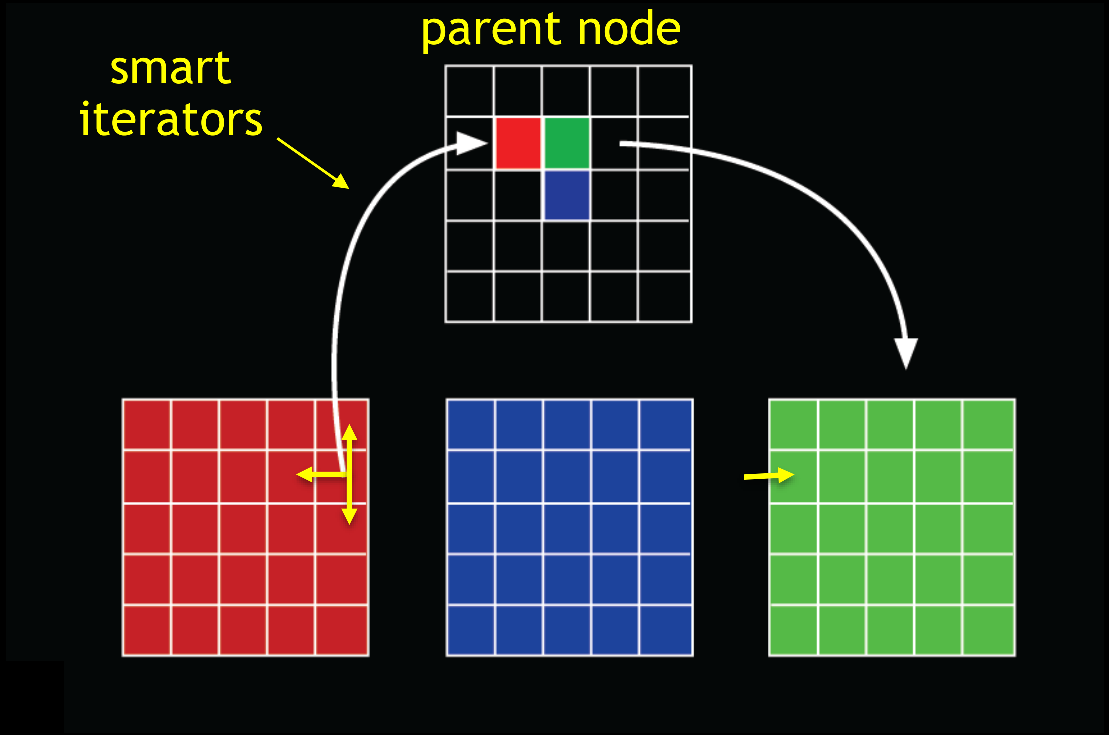
Each voxel must access neighboring voxels in *3D space*.

These may be in different bricks.

NVIDIA.

Compute Operations

How OpenVDB works



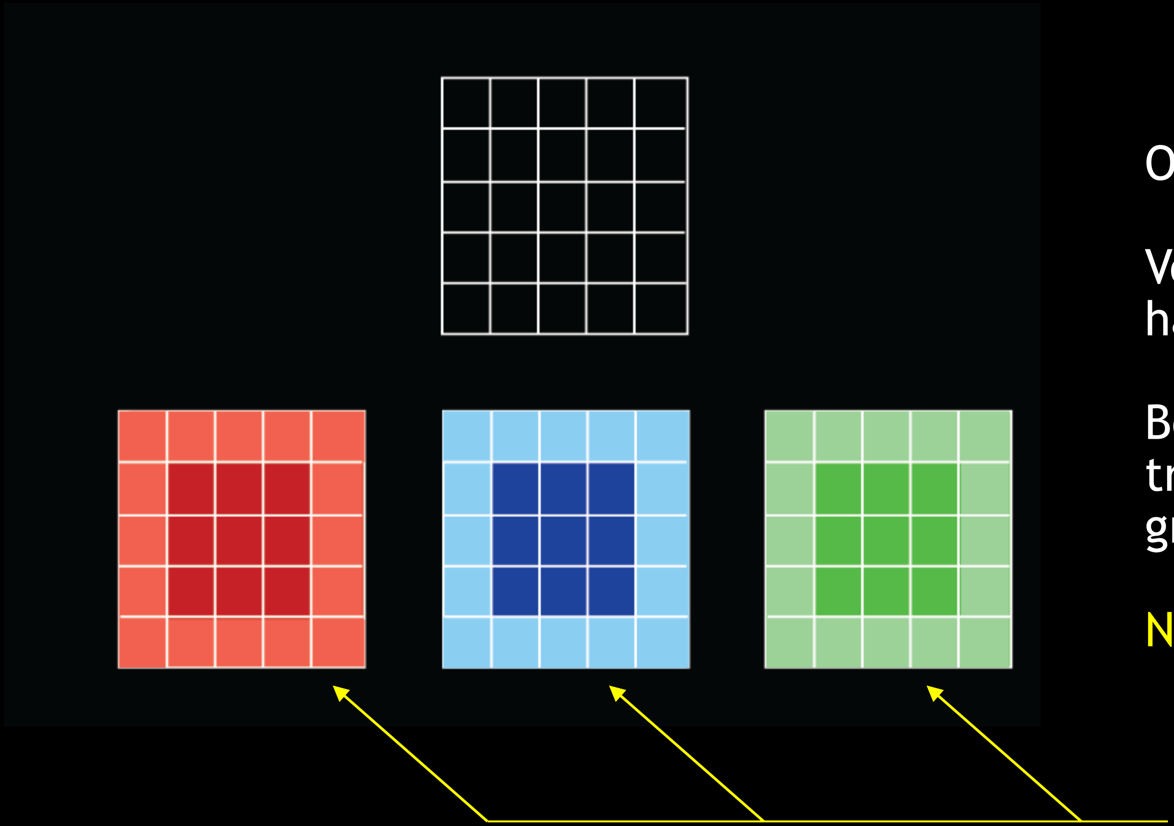
OpenVDB stores voxels in “value” blocks on CPU.

Neighbors are accessed with *smart iterators*, which cache repeatedly used paths in the tree. Suitable for multi-core archs.

Voxels travel up/down the tree, accessing neighbors as needed.

Compute Operations

Voxel workloads



Overall..

Voxels along boundaries have a higher workload.

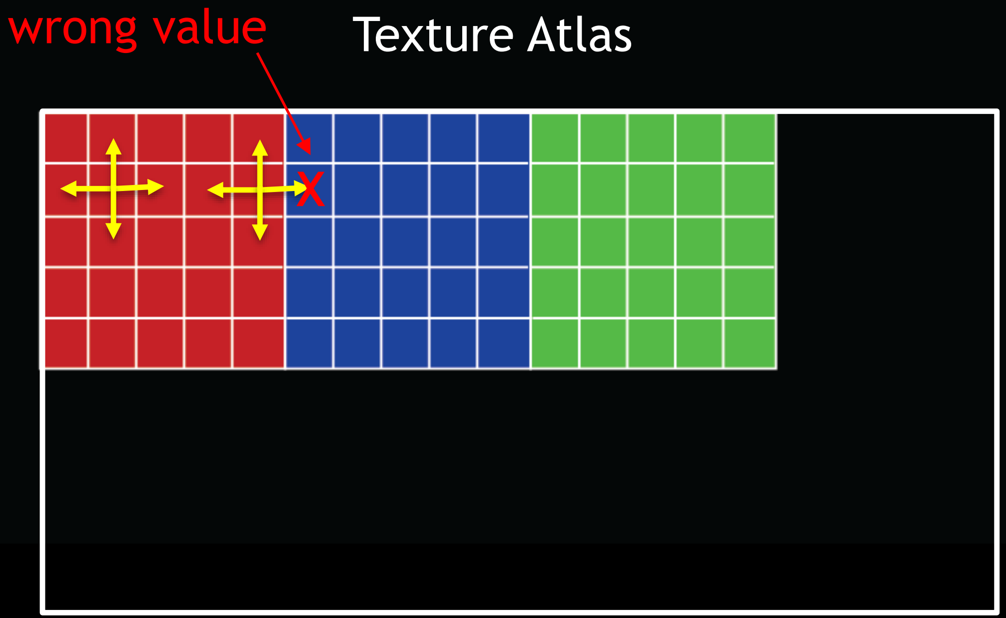
Boundary voxels must traverse the tree, while interior voxels can simply grab neighbors directly.

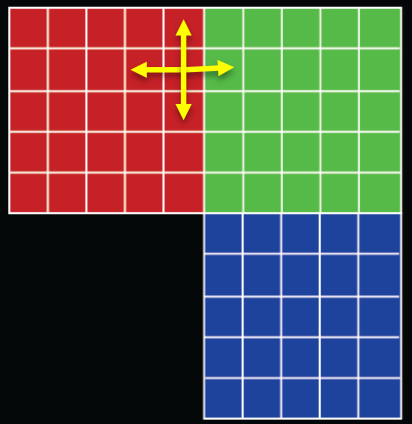
Not ideal for balanced GPU parallelism

Higher workload voxels

Compute Operations

Atlas-based Kernels





All voxels stored in a Texture Atlas

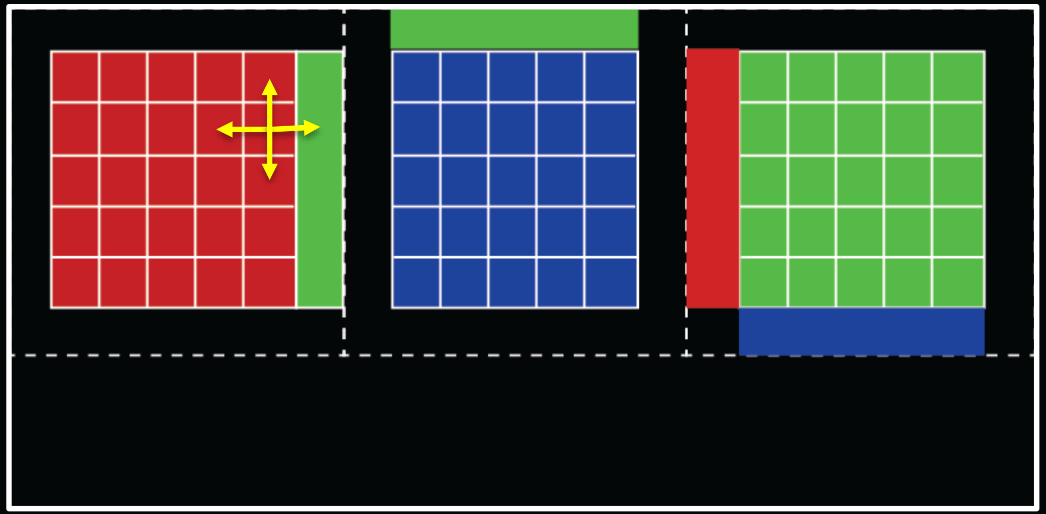
Goal: Run a *single kernel* on the atlas.

Problem: Neighbors are not accessible.

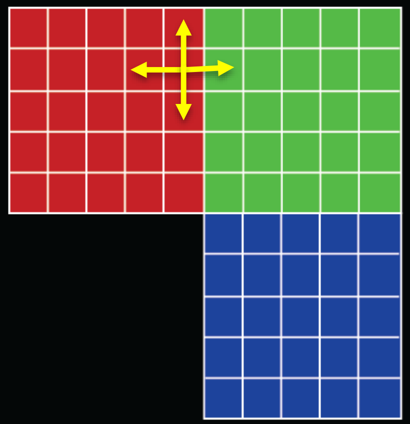
Compute Operations

Apron Cells

Texture Atlas



3D Spatial Layout



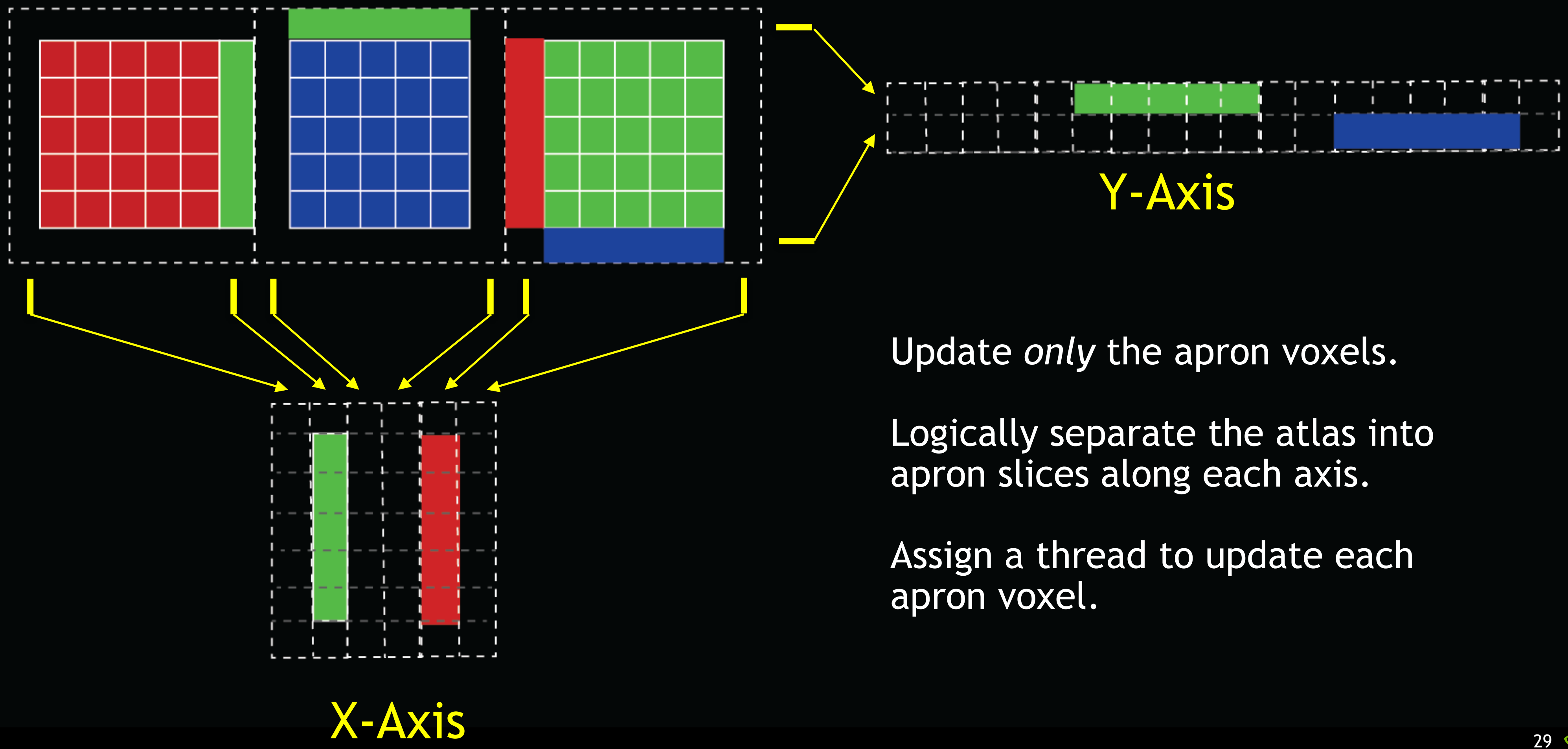
Solution: Apron voxels

Store a margin around each brick which contains correct neighbors at the boundaries.

New Problem: How to populate the neighbors.

Compute Operations

GVDB Axial Apron Updates



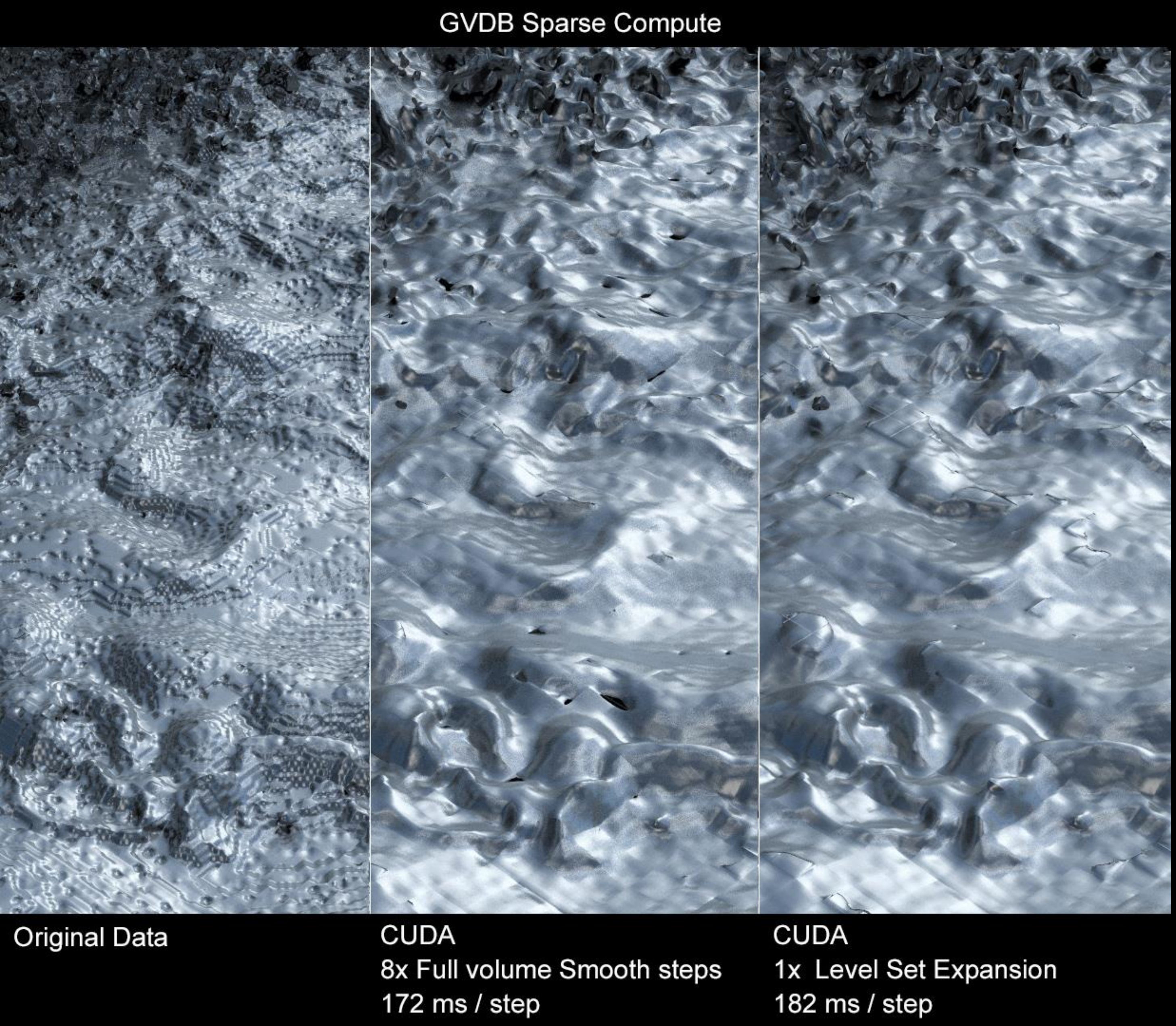
Update *only* the apron voxels.

Logically separate the atlas into apron slices along each axis.

Assign a thread to update each apron voxel.

NVIDIA® GVDB

Compute Operations



Fast GPU kernels over the all sparse voxels.

One user kernel launch.

Three internal apron updates, transparent to user.

Efficient compute on very large domains.

NVIDIA® GVDB

Compute API

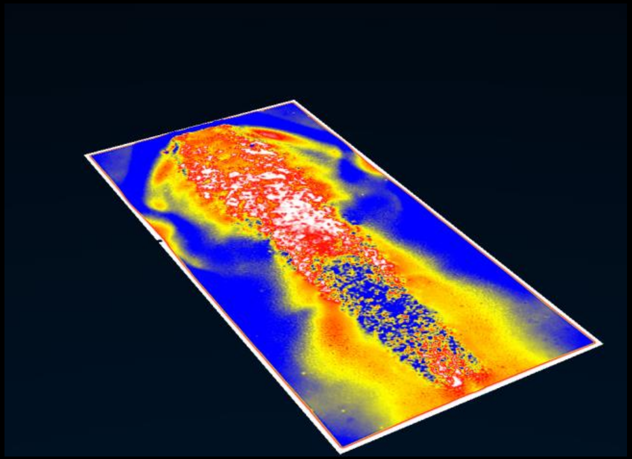
Smoothing Example:

```
extern "C" __global__ void kernelSmooth ( int res, float amt )
{
    GVDB_SHARED_COPY                          ← Macro ensures neighbors are available in shared memory

    float v = 6.0*svox[ndx.x][ndx.y][ndx.z];      ← Smoothing operation
    v += svox[ndx.x-1][ndx.y][ndx.z];                ( values from neighbors)
    v += svox[ndx.x+1][ndx.y][ndx.z];
    v += svox[ndx.x][ndx.y-1][ndx.z];
    v += svox[ndx.x][ndx.y+1][ndx.z];
    v += svox[ndx.x][ndx.y][ndx.z-1];
    v += svox[ndx.x][ndx.y][ndx.z+1];
    v /= 12.0;

    surf3Dwrite ( v, volTexOut, vox.x*sizeof(float), vox.y, vox.z );      ← Output value
}
```

Write kernels as if they were dense.

NVIDIA® GVDB

Compute API



Cross-Section Example:

```
extern "C" __global__ void kernelSectionGVDB ( uchar4* outBuf )
{
     int x = blockIdx.x * blockDim.x + threadIdx.x;
     int y = blockIdx.y * blockDim.y + threadIdx.y;
     if ( x >= scn.width || y >= scn.height ) return;

     // ray intersect with cross-section plane
     float t = rayPlaneIntersect ( scn.campos, rdir, scn.slice_norm, scn.slice_pnt );
     wpos = scn.campos + t*rdir;

     // get node at hit point
     float3 offs, vmin, vdel;
     VDBNode* node = getNodeAtPoint ( wpos, &offs, &vmin, &vdel );

     // get tricubic data value
     clr = transfer ( getTrilinear ( wpos, offs, vmin, vdel ) );

     outBuf [ y*scn.width + x ] = make_uchar4( clr.x*255, clr.y*255, clr.z*255, 255 );
}
```

← Get x,y for current pixel

← Compute world coordinate on a plane

← "Iterators" are still available per voxel.
getNodeAtPoint iterates on GVDB tree

← Get voxel value at hit brick

← Write screen pixel

Using NVIDIA® GVDB *for raytracing*

NVIDIA® GVDB RAYTRACING

Host API

```
gvdb.SetCudaDevice ( devid );

gvdb.Initialize ();

gvdb.LoadVBX ( scnpath );

gvdb.AddRenderBuf ( 0, w, h, 4 );

cuModuleGetFunction ( &cuRaycastKernel,
cuCustom, “my_raycast_kernel" )


gvdb.RenderKernel ( cuRaycastKernel );

unsigned char* buf = malloc ( w*h*4 );
gvdb.ReadRenderBuf ( 0, buf );

save_png ( “out.png", buf, w, h, 4 );
```

← Load a sparse volume from .VBX file

← Create a screen buffer

← Load a user-define raytracing kernel

← Render GVDB with your kernel

← Retrieve the pixels

← Save output

NVIDIA® GVDB RAYTRACING

Kernel API

Get the current pixel →

Ask GVDB to trace the ray, returning hit point and normal →

Custom shading →

Write color to pixel output →

```
#include "cuda_gvdb.cuh“
..
__global__ void raycast_kernel ( uchar4* outBuf )
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    if ( x >= scn.width || y >= scn.height ) return;

    rayMarch ( gvdb.top_lev, 0, scn.campos,
        rdir, hit, norm );        // Trace ray into GVDB

    if ( hit.x != NOHIT ) {
      float3 R= normalize ( reflect3 ( eyedir, norm ) );
      float clr = tex3D ( envmap, R.xy );
    } else {
      clr = make_float3 ( 0.0, 0.0, 0.1 );
    }
    outBuf [ y*scn.width + x ] = make_uchar4(
                clr.x*255, clr.y*255, clr.z*255, 255 );
}
```

NVIDIA® GVDB

API Features:

Write custom shading, custom raytracing kernels, or both

GVDB provides helpers to access nodes, voxels, and neighbors.

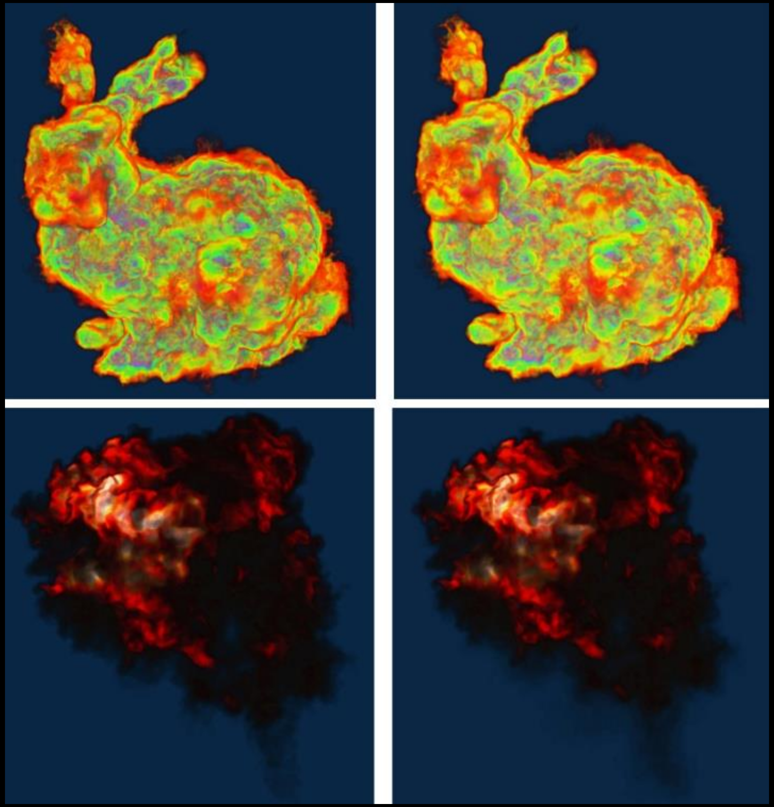
Kernels can be written like they are dense.

Load/save from multiple formats, including .VDB

Run-time VDB configuration

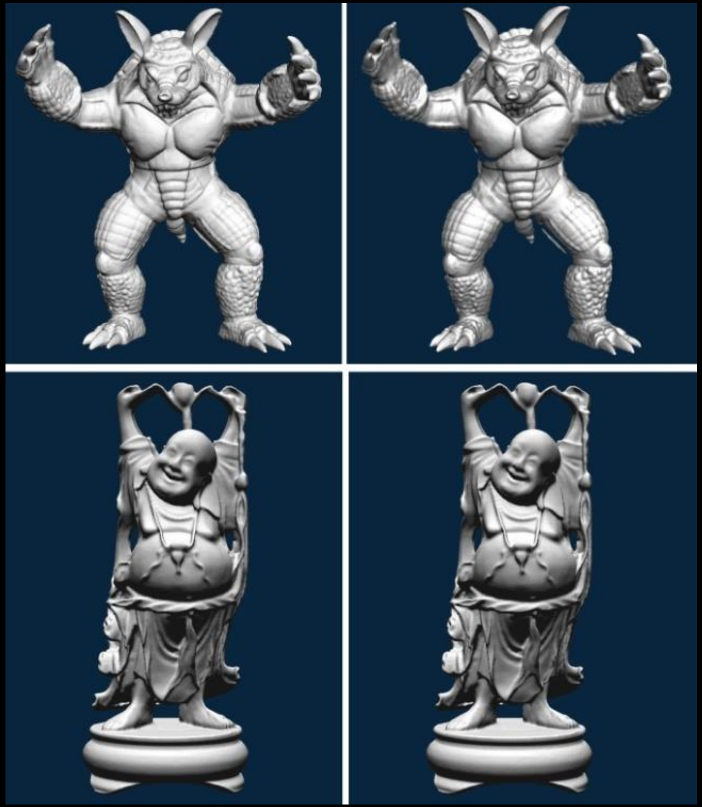
Results

NVIDIA® GVDB



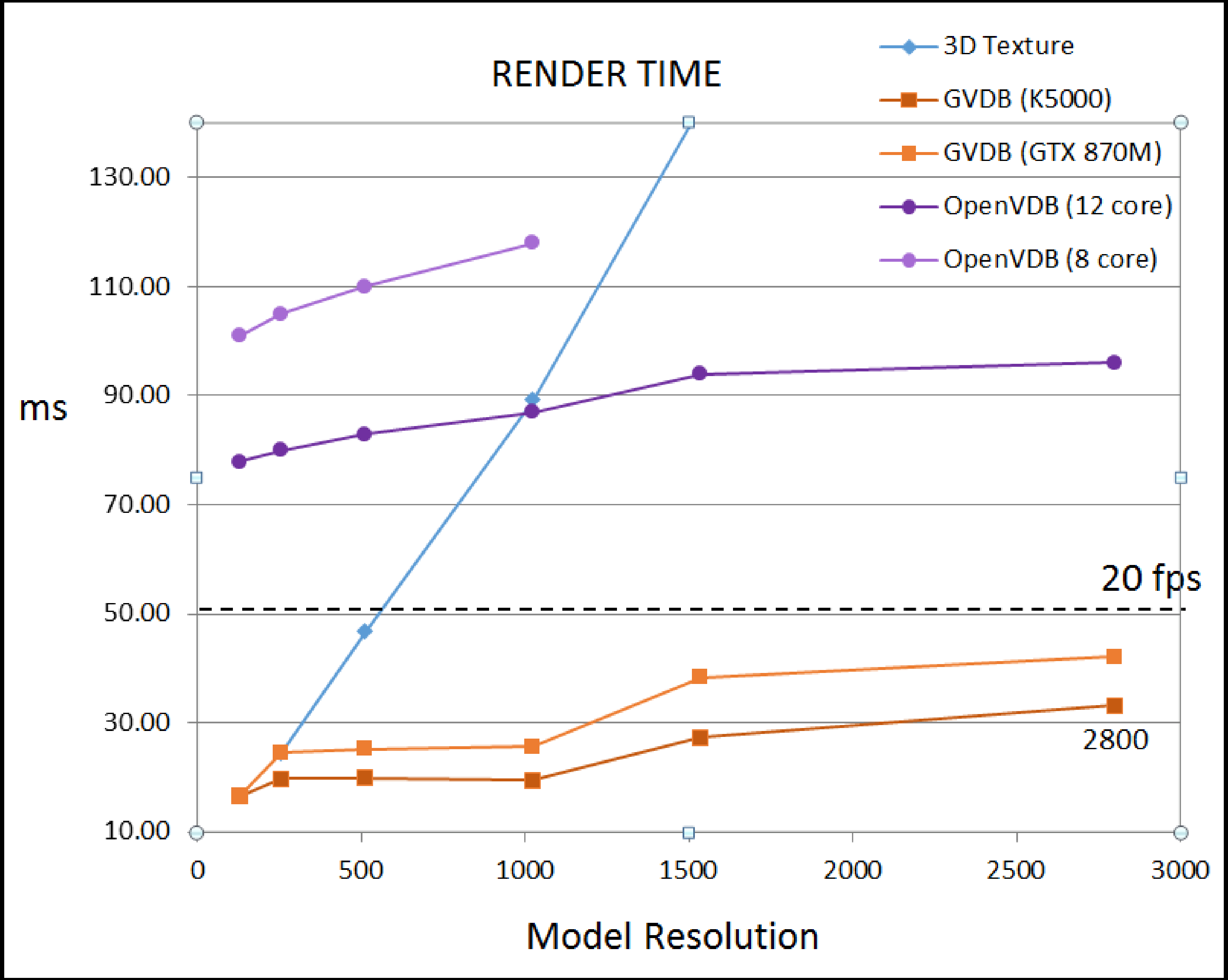
GVDB OpenVDB

Volumes



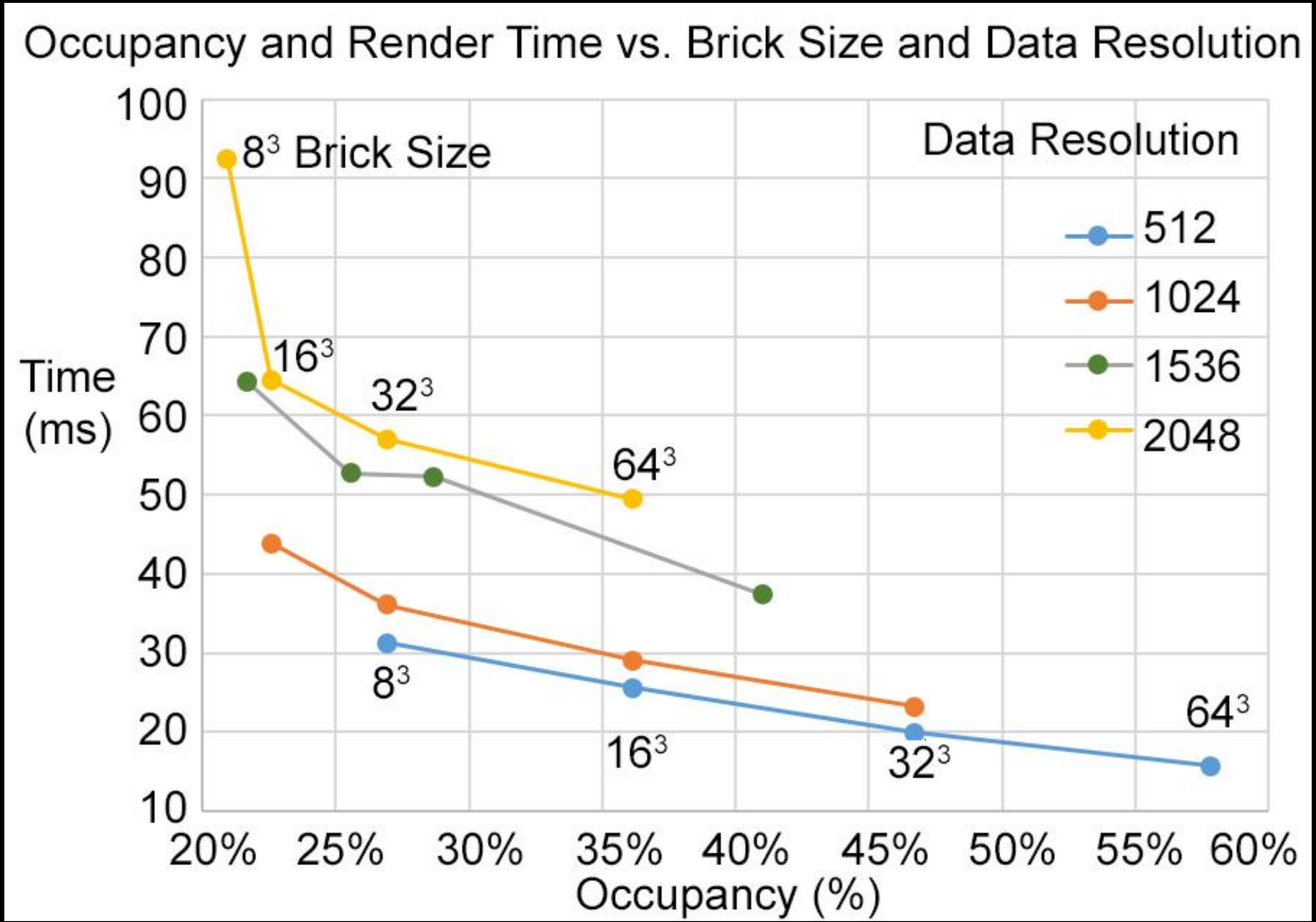
GVDB OpenVDB

Level Sets

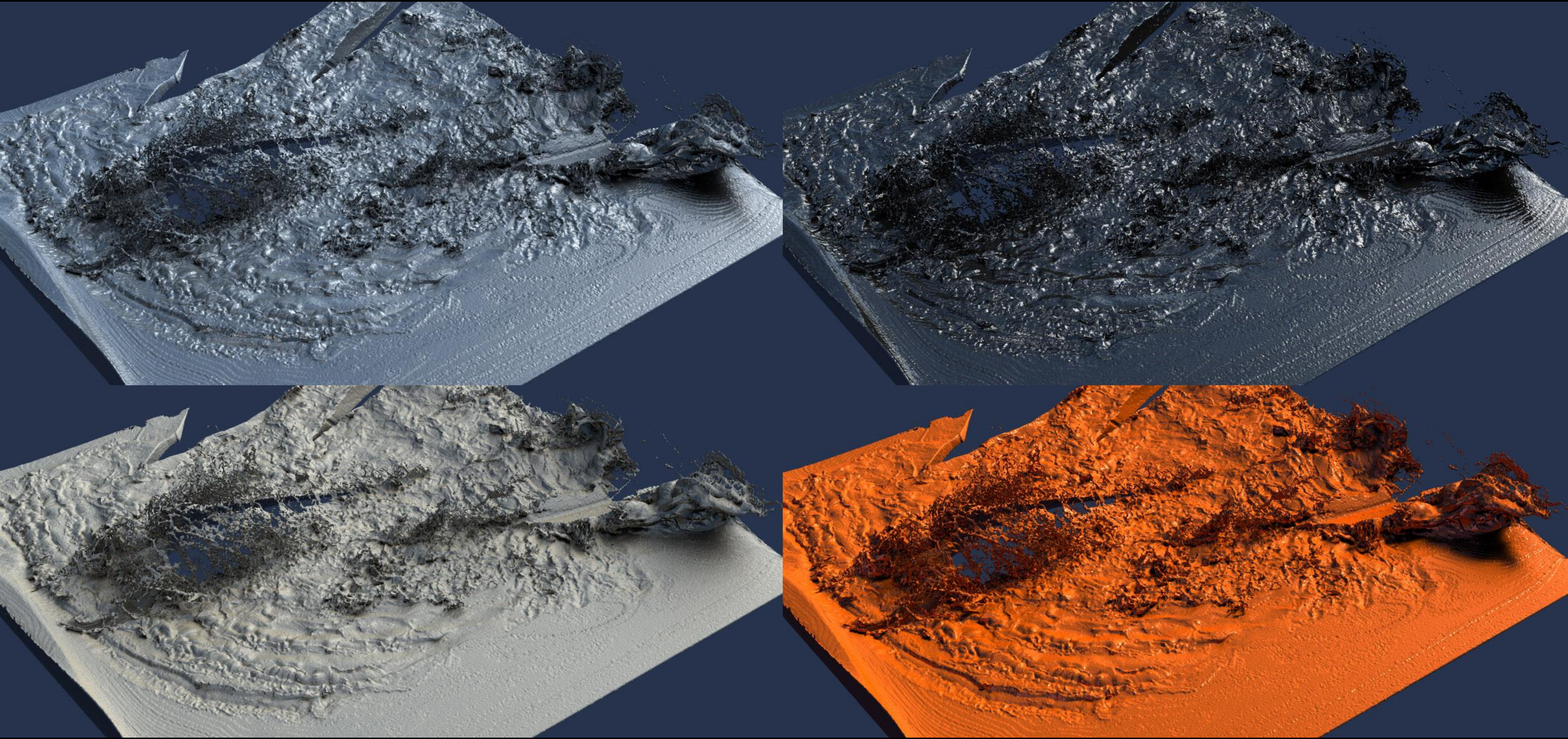


Scaling is similar to OpenVDB, but between 10x-30x faster than CPU

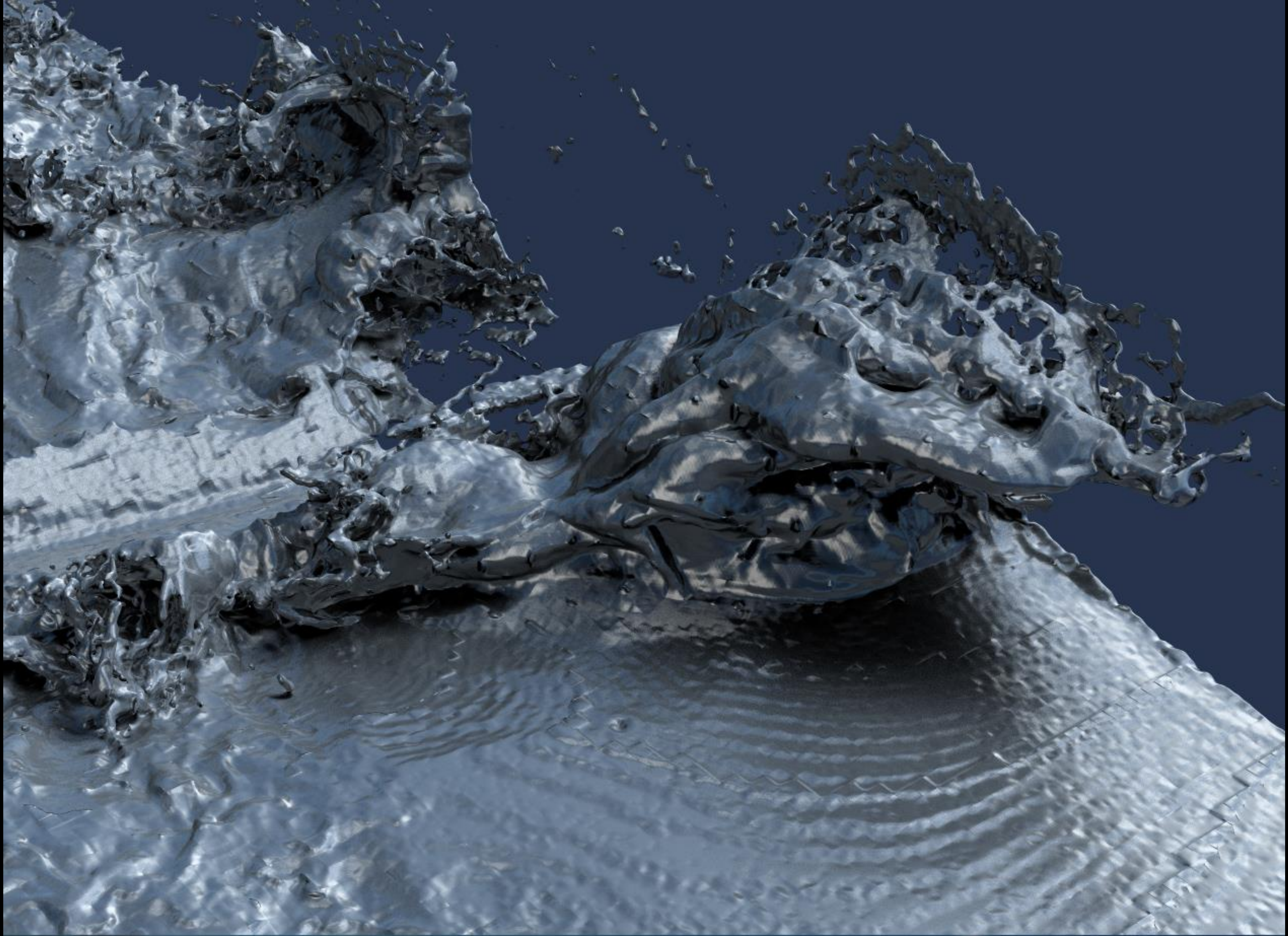
NVIDIA® GVDB



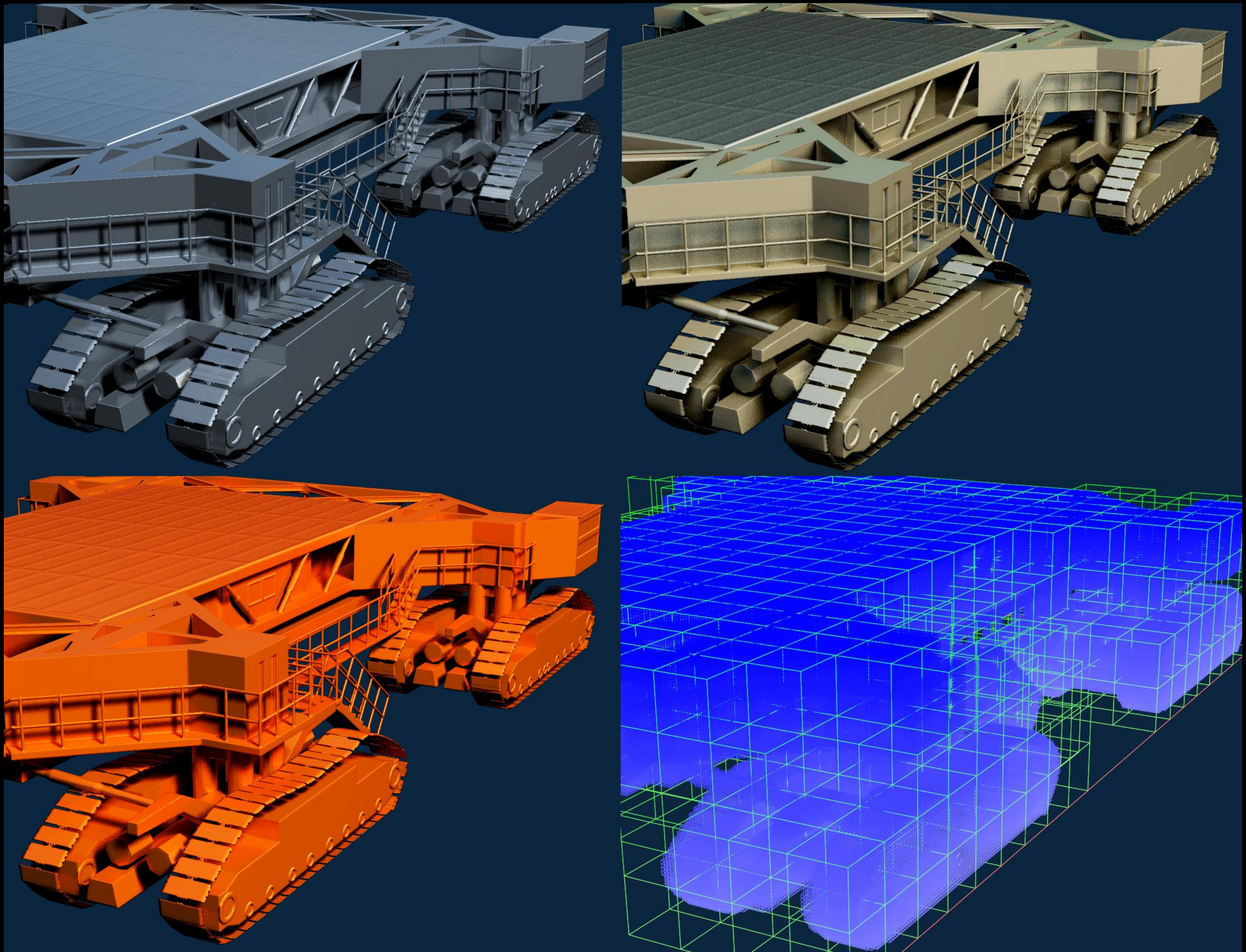
Raytracing time improves with larger bricks.



Interactive Materials & Re-lighting



GVDB Interactive Materials

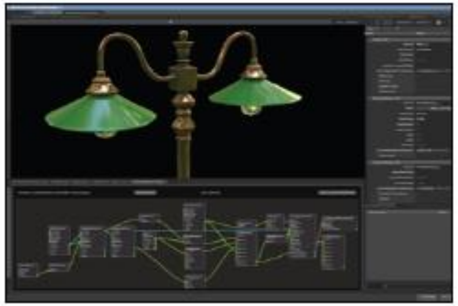


Resources & Availability

NVIDIA DESIGNWORKS™

Iray

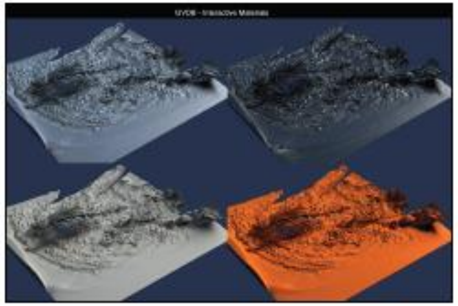
MDL



OptiX

Capture

Video CODEC



GVDB

360 Video

Nsight VSE

NVIDIA® GVDB Sparse Volumes

Availability

API Library with multiple samples

Based on CUDA

Integration with OpenVDB and NVIDIA® OptiX

Open Source with BSD 3-clause License

Available in late September 2016

“ *GVDB is a new rendering engine for VDB data, uniquely suited for NVIDIA GPUs and perfectly complements the CPU-based OpenVDB standard while improving on performance. I am excited to take part in the future adoption of GVDB in the open-source community for visual FX.* ”

— Dr. Ken Museth, Lead Developer of OpenVDB (DreamWorks Animation & SpaceX)

NVIDIA® GVDB Sparse Volumes

Resources

Web Page:

http://developer.nvidia.com/gvdb

Papers & Presentations:

- SIGGRAPH 2016. Raytracing Sparse Volumes with NVIDIA® GVDB in DesignWorks
- High Performance Graphics 2016. GVDB: Raytracing Sparse Voxel Database Structures on the GPU
- GPU Technology Conference 2016. Raytracing Scientific Data in NVIDIA OptiX with GVDB Sparse Volumes.

NVIDIA® GVDB Sparse Volumes

Rama Hoetzlein, rhoetzlein@nvidia.com

Thank you!

http://developer.nvidia.com/gvdb

Thanks to:

Ken Museth — Dreamworks Animation & SpaceX

Tristan Lorach, Tom Fogal
Holger Kunz, Christoph Kubisch
Steven Parker, Chris Hebert