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.*
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

\[ 16 \times 16 = 256 \text{ data values} \]
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

Value Atlas
# Methods for Sparse Volumes

## Meshes & Point Clouds

<table>
<thead>
<tr>
<th>Binary Voxels</th>
<th>acyclic DAGs</th>
<th>voxel hashing (SDF)</th>
<th>graph similarities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kampe, 2013</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Niessner, 2013</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reichl, 2014</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Villanueva, 2016</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Meshes</th>
<th>sparse voxel octrees</th>
<th>sparse voxel octrees</th>
<th>fragment buffers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laine, 2010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chajdas, 2014</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reichl, 2015</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Isosurfaces</th>
<th>complex shaders</th>
<th>multi-res surfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hadwiger, 2005</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Knoll, 2009</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Volumetric Data

<table>
<thead>
<tr>
<th>Octrees</th>
<th>texture-based octree</th>
<th>gigavoxels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boada, 2001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Crassin, 2008</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tilemap Grids</th>
<th>per-sample, out-of-core index table, out-of-core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hadwiger, 2012</td>
<td></td>
</tr>
<tr>
<td>Fogal, 2013</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VDB Grids</th>
<th>hierarchy of N-ary grids</th>
</tr>
</thead>
<tbody>
<tr>
<td>Museth, 2013</td>
<td></td>
</tr>
</tbody>
</table>
OpenVDB

Hierarchy of Grids

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

Hierarchy of Grids

Many levels
Each level is a grid
Each level has its own resolution

\[
\begin{align*}
\text{e.g. top} &= 4x4 \\
\text{mid} &= 3x3 \\
\text{brick} &= 4x4
\end{align*}
\]

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.

\(< L_{N}, ..., L_{2}, L_{1}, L_{0} > \quad L_{0} = \text{Brick dim}\)

**Examples:**

<table>
<thead>
<tr>
<th>Log2 Dims</th>
<th>Tree Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;1, 1, ..., 1&gt;)</td>
<td>Octree</td>
</tr>
<tr>
<td>(&lt;10, 2&gt;)</td>
<td>Tile map</td>
</tr>
<tr>
<td>(*, 2&gt;)</td>
<td>Hash map</td>
</tr>
<tr>
<td>(&lt;5,4,3&gt;)</td>
<td>OpenVDB</td>
</tr>
<tr>
<td>(&lt;3,3,3,4&gt;)</td>
<td>GVDB</td>
</tr>
</tbody>
</table>
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

Lev 3

Root Node

Hashmap

Lev 2

5

Interior Nodes

Bitmask
Child List

Lev 1

4

Interior Nodes

Bitmask
Child List

Lev 0

3

Leaf Nodes

Value List

Value Pointer

Values
**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
Ideal Stencil kernels:

\[
\begin{align*}
v &= \text{tex3D} ( p.x - 1, p.y, p.z ); \\
v &= \text{tex3D} ( p.x + 1, p.y, p.z ); \\
v &= \text{tex3D} ( p.x, p.y - 1, p.z ); \\
v &= \text{tex3D} ( p.x, p.y + 1, p.z ); \\
\end{align*}
\]

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.
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.
Solution: Apron voxels

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

New Problem: How to populate the neighbors.
Update only the apron voxels.

Logically separate the atlas into apron slices along each axis.

Assign a thread to update each apron voxel.
GVDB Sparse Compute

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.
extern "C" __global__ void kernelSmooth ( int res, float amt )
{
    GVDB_SHARED_COPY

    float v = 6.0*svox[ndx.x][ndx.y][ndx.z];
    v += svox[ndx.x-1][ndx.y][ndx.z];
    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 );
}

Smoothing Example:

← Macro ensures neighbors are available in shared memory

← Smoothing operation (values from neighbors)

← Output value

Write kernels as if they were dense.
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
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
```c
#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 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 →
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.
Raytracing time improves with larger bricks.
Interactive Materials & Re-lighting
Resources & 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
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