OpenVDB

From dpawiki

(From the website) OpenVDB is an open source C++ library comprising a novel hierarchical data structure and a suite of tools for the efficient storage and manipulation of sparse volumetric data discretized on three-dimensional grids. It is developed and maintained by DreamWorks Animation for use in volumetric applications typically encountered in feature film production.

Contents

Installation

If in the DPA Studio...

DPA already has OpenVDB setup as part of the DPA pipeline. If you wish to link against the library the following are the paths you will need:

```
libopenvdb: /group/dpa/local/openvdb/lib
header files: /group/dpa/local/openvdb/include
```

You can get to the Doxygen pages for the library in our studio with the following command (assuming you have dpabashrcFile sourced):

```
dpaopen /group/dpa/local/openvdb/share/doc/openvdb/html/index.html
```

Installing OpenVDB Manually

You can get the latest stable build of OpenVDB and OpenVDB for Houdini at the OpenVDB website:

```
http://www.openvdb.org/download/
```

You can get the latest build of OpenVDB and OpenVDB for Houdini at github:

```
https://github.com/dreamworksanimation/openvdb_dev
```

It is recommended that you download the library directly from the website and through git. The git version is the development branch which could be less stable.

Dependencies

gcc (4.1+ or later, recommend 4.6.4 on palmetto or default version on SOC/DPA machines)

Boost [[1] (http://www.boost.org/)] (1.42.0 or later)

OpenEXR and ilmbase [[2] (http://www.openexr.com)]

tbb [[3] (http://www.threadingbuildingblocks.org)]

Setup on Palmetto

Create an interactive session. We need to install the libraries in the following order: boost, ilmbase, OpenEXR, tbb, OpenVDB. Install the libraries to a local directory (recommendation is $HOME/usr/local).

It is also recommended that you add $HOME/usr/local/lib to your LD_LIBRARY_PATH.

Assuming you are in the root directory of each library when building.

Boost (this one takes a while to build):

```
1) ./bootstrap --prefix=$HOME/usr/local --without-libraries=python
2) ./b2 install

alternative version
1)./bootstrap.sh --prefix=$HOME/usr/local --without-libraries=python
2)./bjam install
```

ilmbase:

```
1) ./configure --prefix=$HOME/usr/local
2) make
3) make install
```

OpenEXR:

```
1) ./configure --prefix=$HOME/usr/local --with-ilmbase-prefix=$HOME/usr/local --disable-ilmbasetest
2) make
3) make install
```

tbb:

```
1) make
2) cp -r include $HOME/usr/local
2) cp build/<tbb_build_prefix>_release/*.so* $HOME/usr/local/lib

where tbb_build_prefix you can get from make info.
use make test to check if compiled correctly.
```

OpenVDB:

```
This takes a little more work since we need to modify the Makefile.

change INSTALL_DIR := $(HOME)/usr/local (line 82)
add HT := $(HOME)/usr/local (line 83)
add HDSO := $(HOME)/usr/local/lib (line 84)
change CONCURRENT_MALLOC_LIB := -ltbbmalloc_proxy -ltbbmalloc (line 110/111)
comment out directory for CPPUNIT_INCL_DIR (line 117)
comment out directory for GLFW_INCL_DIR (line 124)
comment out version for PYTHON_VERSION (line 131)
comment out file name for DOXYGEN (line 158)

Afterwards, run 'make install' to build the libraries and commands into your install directory
```

Limiting the number of Threads when Rendering on Palmetto

When rendering on Palmetto, you must ensure OpenVDB does not use more threads than you request or your jobs may be terminated without warning. The following is a code snippet to address this issue:

```
#include <tbb/task_scheduler_init.h>
//...preamble code here
main() {
int numthreads;
//... set numthreads to the number of cores.

tbb::task_scheduler_init schedulerInit(numthreads);

//...rest of code here
return;
}
```

Installing OpenVDB for Houdini 12.1

Kacey: Will post, ask Kacey for now...

Quick OpenVDB Programming Guide

Detailed documentation can be found here[[4] (http://www.openvdb.org/documentation/doxygen/)]. The important parts are the Transforms and Maps, FAQ, and Cookbook discussions. Be aware that the library is heavily templated, so if will take quite some time to compile if you decide to include openvdb.h. Note that while the online documentation claims that openvdb::initialize() needs to be called once per program, it only needs to be called when doing file I/O with OpenVDB (so only if you are reading or writing an OpenVDB grid).

Example g++ command from terminal (assuming you followed the above library locations):

```
g++ -g -O2 -I$HOME/usr/local/include -o program program.cpp -L$HOME/usr/local/lib -lHalf -ltbb -lopenvdb
```

Note: you will have to use $(HOME) if you have a Makefile to compile.

Compilation Tricks

There are a few tricks that can be done to speed up compile times. The easiest method is to only include the headers needed to perform the operations desired, but that can become cumbersome when programming. Its easier to dump all OpenVDB headers needed into a single file and typedef all the types being used. This will force the compiler to expand out only the necessary bits and compile only once (hopefully). Please see Sam C. if you have any other suggestions to speed up compile time.

```
// Types.h
// typedef a whole bunch of templated objects from openvdb
// we want to include the minimal number of header files
// for compilation speed

#ifndef __TYPEDEF_CODE_H__
#define __TYPEDEF_CODE_H__

#include <openvdb/Grid.h>
#include <openvdb/math/Coord.h>
#include <openvdb/math/Transform.h>
#include <openvdb/math/Vec3.h>
#include <openvdb/math/Vec4.h>
#include <openvdb/tree/Tree.h>
#include <openvdb/util/Util.h>

// typedef a FloatTree and FloatGrid
```

```
typedef openvdb::tree::Tree4<float, 5, 4, 3>::Type FloatTree;
typedef openvdb::Grid<FloatTree> FloatGrid;

// typedef a Vec4 grid (for color and other useful things)
typedef openvdb::tree::Tree4<openvdb::math::Vec4s, 5, 4, 3>::Type Vec4
typedef openvdb::Grid<Vec4fTree> Vec4fGrid;

// typedef math objects
typedef openvdb::math::Coord Coord;
typedef openvdb::math::Vec3s Vec3s;
typedef openvdb::math::Vec3s Vec3f;
typedef openvdb::math::Vec3d Vec3d;
typedef openvdb::math::Vec3<uint32_t> Vec3I;
typedef openvdb::math::Vec4<uint32_t> Vec4I;
#define INVALID_IDX openvdb::util::INVALID_IDX

// grid stuff
typedef openvdb::math::Transform Transform;
typedef openvdb::math::UniformScaleMap UniformScaleMap;

#endif
```

Basic Grids and Accessors

A lot of OpenVDB is done using shared pointers, so the code might look a little strange to those who haven't used this sort of thing yet. Accessing voxels in the grid is normally done through something called an Accessor. The Accessor is a complicated object, as it maintains a cache to speed up voxel access. It also requires the coordinates to be in index space (relative to the grid, type is a openvdb::Coord). Calling a grid's getAccessor() function will return a new Accessor.

Please note that using a single Accessor across multiple threads messes with the cache, its more efficient to have each thread use it's own Accessor. Uncached thread safe random access can be done through the grid's tree (grid->tree().getValue(ijk)).

```
#include "Types.h"

int main(int argc, char** argv){
  // create a simple grid with default map and transform.
  FloatGrid::Ptr grid = FloatGrid::create();

  // create a simple grid with a background value of 0.1
  FloatGrid::Ptr grid2 = FloatGrid::create(0.1);

  // copy a grid
  FloatGrid::Ptr copy = grid->deepCopy();
```

```
    // create a coord and accessor
    FloatGrid::Accessor accessor = grid->getAccessor();
    Coord ijk(1, 1, 1);

    // set the voxel value at index [1,1,1] and print it out
    accessor.setValue(ijk, 1.0);
    std::cout << "Grid: " << ijk << " = " << accessor.getValue(ijk) << s
    std::cout << "Grid2: " << ijk << " = " << grid2->getAccessor().getVa
    std::cout << "copy: " << ijk << " = " << copy->getAccessor().getValu
    std::cout << "uncached tree access: " << ijk << " = " << grid->tree(
}
```

One of the awesome things about OpenVDB is that it tracks all voxels that are active, but what exactly is an active voxel? OpenVDB considers an active voxel to be one that has some value stored at it that is not the background value. But not just that, it also tracks a certain number of voxels surrounding that voxel which have the background value (this is controlled through the Narrow Band). A user can manually turn a voxel on or off through an Accessor (using accessor.setValueOn(Coord) or accessor.setValueOff(Coord)).

So why is this awesome? Well we can access only the active voxels since we know the set of voxels that are active. This is done using a Value Iterator.

```
// iterate over all active grid voxels
for(FloatGrid::ValueOnIter iter = grid->beginValueOn(); iter; ++iter){
  std::cout << "Grid: " << iter.getCoord() << " = " << *iter << std::e
}
```

You can also access those values as constants using FloatGrid::ValueOnCIter iter = grid->cbeginValueOn();.

Transforms and Maps

The subject of how Transforms and Maps work in OpenVDB can be a little complicated, and both are related. Typically a Transform is used to convert a World Space position (which is a 3d vector) to Index Space (a Coordinate or a 3d Vector) in OpenVDB. Of course the process can also be reversed (converting a Coordinate or 3d vector in Index Space to a 3d vector in World Space). However a Transform just provides a simple interface for the user to convert between spaces. The math is taken care of by a Map. Usually you only need the Transform, however there are a few tools which require the Map.

Quick note about Index Space. There are 2 ways to visualize the grid, a cell centered grid or a node centered grid. Cell centered grids store their value at the center of the cell and node centered grids store their value at the nodes. I find it easier to just use node centered grids, but the only important thing is that you are consistent across your code.

```cpp
#include "Types.h"

int main(int argc, char** argv){
  // get a grid's transform
  FloatGrid::Ptr grid = FloatGrid::create();
  Transform::Ptr transform = grid->transformPtr();

  // convert a world space position to index space position
  Vec3s xyz(1,1,1);
  Vec3s pos = transform->worldToIndex(xyz);
  std::cout << "pos in index " << pos << std::endl;

  // convert a world space position to index space coordinate
  Coord ijk = transform->worldToIndexNodeCentered(xyz);
  std::cout << "coord in index " << ijk << std::endl;

  // convert a index space coordinate to a world space position
  std::cout << "pos in index back to world " << transform->indexToWor

  // create a linear transform that sets the voxel size of the grid to
  transform = Transform::createLinearTransform(0.1);

  // create a grid with voxel size 0.1
  // note, there are multiple ways to do this. this might not be the r
  grid->setTransform(transform);

  std::cout << "voxel size: " << grid->voxelSize() << std::endl;
  pos = transform->worldToIndex(xyz);
  std::cout << "pos in index " << pos << std::endl;
  ijk = transform->worldToIndexNodeCentered(xyz);
  std::cout << "coord in index " << ijk << std::endl;
  std::cout << "pos in index back to world " << transform->indexToWor

  // get a transform's map
  // again, there are a few ways to get this.  UniformScaleMaps are th
  UniformScaleMap::Ptr map = transform->map<openvdb::math::UniformScal

  return 0;
}
```

Level Sets and Fog Volumes

In mathematics a Level Set is a set of variables where when evaluated by a function takes on a constant value. For the purposes of this class, a Level Set is a signed distance field where the surface of a volume exists where the distance is 0. Essentially this is a field that tells you how far away you are from the surface of a volume

when given a point in space. OpenVDB considers a positive value as outside the surface, and a negative value inside the surface.

There are a few different ways to build a level set. The recommended way is by using levelSetRebuild (using the default values for halfwidth, and xform), however there are a few things to keep in mind. The isovalue is the value that defines the surface, this should be a very small number when building a level set from a density volume. The halfWidth dictates the number of voxels inside and outside the surface that are filled in (since OpenVDB is a sparse grid we don't want to use more voxels then needed). By default the halfWidth is 3. This could cause problems if you are trying to perform an operation on a point in space that is not within the halfWidth. One of the ways to rebuilding a level set allows you to specify the number of voxels used outside the surface and inside. Look at the documentation for more details.

A fog volume is essentially OpenVDB terms for a density field (or a volume). You can convert a level set to a fog volume by using the sdfToFogVolume function. This function will start with density = 0 at the surface, and linearly ramp the interior density to 1. It is important to note that when you perform a deformation on a levelset, you change where the surface is. This could possibly damage the surface. To undo this, you need to rebuild the levelset. This is done by converting the level set into a fog volume, then converting the fog volume back into a level set. The fog volume conversion will give you the proper density field which will allow you to rebuild the level set.

```
#include <openvdb/tools/LevelSetSphere.h>
#include <openvdb/tools/LevelSetRebuild.h>
#include <openvdb/tools/LevelSetUtil.h>
#include "Types.h"
```

```
int main(int argc, char** argv){

  // there is a really neat function called createLevelSetSphere whic
  FloatGrid::Ptr sphere = openvdb::tools::createLevelSetSphere<FloatGr
  Transform::Ptr transform = sphere->transformPtr();
  FloatGrid::Accessor accessor = sphere->getAccessor();

  // level set outside the surface
  Vec3f xyz(51,0,0);
  Coord ijk = transform->worldToIndexNodeCentered(xyz);
  std::cout << "outside surface: " << accessor.getValue(ijk) << std::e

  // level set on the surface
  xyz = Vec3f(50,0,0);
  ijk = transform->worldToIndexNodeCentered(xyz);
  std::cout << "on surface: " << accessor.getValue(ijk) << std::endl;

  // level set in the surface
  xyz = Vec3f(49,0,0);
  ijk = transform->worldToIndexNodeCentered(xyz);
  std::cout << "in surface: " << accessor.getValue(ijk) << std::endl;
```

```
// convert level set to fog volume
openvdb::tools::sdfToFogVolume<FloatGrid>(*sphere);
transform = sphere->transformPtr();
accessor = sphere->getAccessor();

// fog volume outside the surface
xyz = Vec3f(51,0,0);
ijk = transform->worldToIndexNodeCentered(xyz);
std::cout << "outside surface: " << accessor.getValue(ijk) << std::e

// fog volume on the surface
xyz = Vec3f(50,0,0);
ijk = transform->worldToIndexNodeCentered(xyz);
std::cout << "on surface: " << accessor.getValue(ijk) << std::endl;

// for volume in the surface
xyz = Vec3f(49,0,0);
ijk = transform->worldToIndexNodeCentered(xyz);
std::cout << "in surface: " << accessor.getValue(ijk) << std::endl;

// rebuild the level set
FloatGrid::Ptr grid = openvdb::tools::levelSetRebuild<FloatGrid>(*sp
transform = grid->transformPtr();
accessor = grid->getAccessor();

  // level set outside the surface
  xyz = Vec3f(51,0,0);
  ijk = transform->worldToIndexNodeCentered(xyz);
  std::cout << "outside surface: " << accessor.getValue(ijk) << std::e

  // level set on the surface
  xyz = Vec3f(50,0,0);
  ijk = transform->worldToIndexNodeCentered(xyz);
  std::cout << "on surface: " << accessor.getValue(ijk) << std::endl;

  // level set in the surface
  xyz = Vec3f(49,0,0);
  ijk = transform->worldToIndexNodeCentered(xyz);
  std::cout << "in surface: " << accessor.getValue(ijk) << std::endl;

  return 0;
}
```

Dense Grids

Dense grids are in OpenVDB for convenience. By default the memory layout of a dense grid is z major, y, then x (vs x major, y, then z). Be aware that dense grids take up significantly more memory than sparse grids. For instance, a 500^3 floating point dense grid takes up about 500MB.

```
#include <openvdb/tools/Dense.h>
#include <openvdb/tools/LevelSetSphere.h>
#include "Types.h"

int main(int argc, char** argv)
{
  // setup initial grid
  FloatGrid::Ptr sphere = openvdb::tools::createLevelSetSphere<FloatGrid>(50.0, Vec3f(0,0,0), 0.5);
  Coord ijk(0, 0, 0);
  std::cout << "value at origin: " << sphere->getAccessor().getValue(ijk) << std::endl;

  // create a float type 500^3 dense grid centered at the origin (in index space)
  Coord dim(500, 500, 500);
  openvdb::tools::Dense<float> dense(dim);

  // copy float grid data in range to dense grid
  openvdb::tools::copyToDense<openvdb::tools::Dense<float>, FloatGrid>(*sphere, dense);
  std::cout << "value at origin: " << dense.getValue(ijk) << std::endl;

  // get a pointer to the dense grid's data
  float *data = dense.data();
  std::cout << "value at origin: " << data[0] << std::endl;

  // create a new sparse grid from dense grid
  FloatGrid::Ptr grid2 = FloatGrid::create();
  openvdb::tools::copyFromDense<openvdb::tools::Dense<float>, FloatGrid>(dense, *grid2, 0.001);
  std::cout << "value at origin: " << grid2->getAccessor().getValue(ijk) << std::endl;

  return 0;
}
```

Frustum Grids

OpenVDB handles frustum grids by means of a Transform, specifically a Frustum Transform. There are a few things that need to be defined first though.

```
#include <openvdb/math/BBox.h>
#include "Types.h"

int main(int argc, char** argv)
{
  FloatGrid::Ptr grid = FloatGrid::create();
  Coord ijk(5, 0, 0);
  std::cout << ijk << " transform: " << grid->transformPtr()->indexToWorld(ijk) << std::endl;

  // create the bounding box that will be mapped by the transform into the shape of a frustum.
  openvdb::math::BBox<openvdb::math::Vec3d> box(openvdb::math::Vec3d(0,0,0), openvdb::math::Vec3d(100,100,10

  // the far plane of the frustum is 2x the size of the near plane
  double taper = 2.0;

  // depth is 10x the x-width of the near plane
  double depth = 10.0;

  // x-width of the frustum in world space units
  double xwidth = 100.0;

  // construct the frustum transform
```

```
  Transform::Ptr frustum = Transform::createFrustumTransform(box, taper, depth, xwidth);

  // create a grid with a frustum transform
  grid->setTransform(frustum);
  std::cout << ijk << " transform: " << grid->transformPtr()->indexToWorld(ijk) << std::endl;

  return 0;
}
```

Sampling

There are multiple types of usable samplers. A PointSampler will round to the nearest voxel, while a BoxSampler or QuadraticSampler will do some type of interpolation. We will be using BoxSamplers for the most part since it performs trilinear interpolation. Furthermore, you can sample through an Accessor or directly from the tree data structure. Sampling from the tree directly is better if you are only doing a few samples, but if you are sampling frequently around an area then using the Accessor method is better.

Once constructed, you can sampler in either index space (isSample(Coord) or isSample(Vec3d)) or world space (wsSample(Vec3d)).

```
#include <openvdb/tools/Interpolation.h>
#include <openvdb/tools/LevelSetSphere.h>
#include "Types.h"

int main(int argc, char** argv)
{
  // create a sphere level set
  FloatGrid::Ptr grid = openvdb::tools::createLevelSetSphere<FloatGrid>(50.0, Vec3f(0,0,0), 0.5);

  // construct a float grid sampler using a grid
  openvdb::tools::GridSampler<FloatGrid, openvdb::tools::BoxSampler> sampler1(*grid);

  // construct a float grid sampler using an Accessor
  FloatGrid::Accessor acc = grid->getAccessor();
  openvdb::tools::GridSampler<FloatGrid::Accessor, openvdb::tools::BoxSampler> sampler2(acc, grid->transform

  // world space sample
  Vec3f pos(49.5, 0, 0);
  std::cout << pos << " sampler 1: " << sampler1.wsSample(pos) << std::endl;
  std::cout << pos << " sampler 2: " << sampler2.wsSample(pos) << std::endl;

  // index space sample
  Coord ijk(99,1,1);
  std::cout << ijk << " sampler 1: " << sampler1.isSample(ijk) << std::endl;
  std::cout << ijk << " sampler 2: " << sampler2.isSample(ijk) << std::endl;

  return 0;
}
```

Closest Point Transform

The CPT operator will compute the closest-point transform to a level set. There are two forms of the CPT, one which computes the transform in the domain space of the map (voxel space) and the other which computes in the range space of the map (world space).

Note: The given coordinate must be within the bounds of the grid half-width (ie, on an active voxel).

```
#include <openvdb/math/FiniteDifference.h>
#include <openvdb/math/Operators.h>
#include <openvdb/tools/LevelSetSphere.h>
#include "Types.h"

int main(int argc, char** argv)
{
  // create a sphere level set and accessor
  FloatGrid::Ptr grid = openvdb::tools::createLevelSetSphere<FloatGrid>(50.0, Vec3f(0,0,0), 0.5);
  FloatGrid::Accessor acc = grid->getAccessor();
  Transform::Ptr transform = grid->transformPtr();
  UniformScaleMap::Ptr map = transform->map<UniformScaleMap>();

  // create a closest point transform operator using a uniform scale map
  // and a 2nd order center difference finite difference scheme
  // CPT_RANGE will compute in the range space of the map (world space)
  openvdb::math::CPT_RANGE<UniformScaleMap, openvdb::math::CD_2ND> cptROp;

  // create a closest point transform operator using a uniform scale map
  // and a 2nd order center difference finite difference scheme
  // CPT will compute in domain space of the map (voxel space)
  openvdb::math::CPT<UniformScaleMap, openvdb::math::CD_2ND> cptOp;

  // Transform a world space coordinate (outside the level set) into index space for CPT
  Vec3f pos(50.5, 0.1, 0.1);
  Coord ijk(transform->worldToIndexNodeCentered(pos));

  // Find the closest point on the surface of level set
  std::cout << pos << " cpt: " << cptOp.result(*map, acc, ijk) << std::endl;
  std::cout << pos << " cptR: " << cptROp.result(*map, acc, ijk) << std::endl;

  return 0;
}
```

Gradient

Gradient computes the rate of change in all directions (in this case x, y, and z) in the range space of the map (world space).

Note: The given coordinate must be within the bounds of the grid half-width (ie, on an active voxel).

```
#include <openvdb/math/FiniteDifference.h>
#include <openvdb/math/Operators.h>
#include <openvdb/tools/LevelSetSphere.h>
#include "Types.h"

int main(int argc, char** argv)
{
  // create a sphere level set and accessor
  FloatGrid::Ptr grid = openvdb::tools::createLevelSetSphere<FloatGrid>(50.0, Vec3f(0,0,0), 0.5);
  FloatGrid::ConstAccessor acc = grid->getConstAccessor();
  Transform::Ptr transform = grid->transformPtr();
  UniformScaleMap::Ptr map = transform->map<UniformScaleMap>();

  // create a gradient operator using a Uniform Scale Map
  // and a 2nd order center difference finite difference scheme
  // computation is done in the range space of the map (world space)
  openvdb::math::Gradient<UniformScaleMap, openvdb::math::CD_2ND> gradOp;

  // Transform a world space coordinate (outside the level set) into index space
  Vec3f pos(50.5, 0.5, 0.5);
```

```
  Coord ijk(transform->worldToIndexNodeCentered(pos));

  // Find the closest point on the surface of level set
  std::cout << pos << " grad: " << gradOp.result((*map), acc, ijk) << std::endl;

  return 0;
}
```

MeshToVolume and VolumeToMesh

Creating a level set from a mesh is not completely straight forward. You need to create a vector of verticies (Vec3s) in the mesh as well as a vector of faces (Vec4I). If you have triangles, then the forth index in the face must be set to INVALID_IDX (to tell OpenVDB that there are only 3 verticies making up the face). Also, you need to transform the verticies from world space to index space before creating the level set.

OpenVDB performs the reverse operation through an algorithm like marching cubes. As such, you will probably end up with more verticies and faces than what you started with. Also, the scaling might not be the same either (but that is a simple fix). VolumeToMesh will store the verticies in a PointList (a boost::scoped_array<Vec3s>) and the faces in a PolygonPoolList (a boost::scoped_array<PolygonPool>). Since there could be multiple PolygonPools, you need to check to size of the list. Also, the PolygonPool might contain both triangles and quads, so it has methods to access that data if it exists.

```
#include <openvdb/tools/MeshToVolume.h>
#include <openvdb/tools/VolumeToMesh.h>
#include <vector>
#include "Types.h"
```

```
int main(int argc, char** argv)
{
  // create a simple box mesh
  std::vector<Vec3s> verts;
  std::vector<Vec4I> faces;
  verts.push_back(Vec3s(0,0,0));
  verts.push_back(Vec3s(0,0,1));
  verts.push_back(Vec3s(0,1,0));
  verts.push_back(Vec3s(0,1,1));
  verts.push_back(Vec3s(1,0,0));
  verts.push_back(Vec3s(1,0,1));
  verts.push_back(Vec3s(1,1,0));
  verts.push_back(Vec3s(1,1,1));
  faces.push_back(Vec4I(0,6,4,INVALID_IDX));
  faces.push_back(Vec4I(0,2,6,INVALID_IDX));
  faces.push_back(Vec4I(0,3,2,INVALID_IDX));
  faces.push_back(Vec4I(0,1,3,INVALID_IDX));
  faces.push_back(Vec4I(2,7,6,INVALID_IDX));
  faces.push_back(Vec4I(2,3,7,INVALID_IDX));
  faces.push_back(Vec4I(4,6,7,INVALID_IDX));
  faces.push_back(Vec4I(4,7,5,INVALID_IDX));
  faces.push_back(Vec4I(0,4,5,INVALID_IDX));
  faces.push_back(Vec4I(0,5,1,INVALID_IDX));
  faces.push_back(Vec4I(1,5,7,INVALID_IDX));
  faces.push_back(Vec4I(1,7,3,INVALID_IDX));

  // create a transform to convert world space coords to index space
  Transform::Ptr transform = Transform::createLinearTransform();
  for(size_t ndx = 0; ndx < verts.size(); ndx++)
    verts[ndx] = transform->worldToIndex(verts[ndx]);

  // create the level set from the mesh
  // note: this uses the default interior and exterior band width of 3 voxels
  openvdb::tools::MeshToVolume<FloatGrid> fromMesh(transform);
```

```
fromMesh.convertToLevelSet(verts, faces);

// get the signed distance grid
FloatGrid::Ptr grid = fromMesh.distGridPtr();
std::cout << "Volume active voxels: " << grid->activeVoxelCount() << std::endl;

// new verts & faces & transform
std::vector<Vec3s> verts2;
std::vector<Vec4I> faces2;
Transform::Ptr transform2 = grid->transformPtr();

// create a mesher and convert the grid
// since this is a level set, the surface is at isovalue = 0
openvdb::tools::VolumeToMesh mesher;
mesher.operator()<FloatGrid>(*grid);

// now get the verts and faces from the mesher
openvdb::tools::PointList *vertlist = &mesher.pointList();
openvdb::tools::PolygonPoolList *facelist = &mesher.polygonPoolList();

// convert verts to world space
for(int i = 0; i < mesher.pointListSize(); i++)
  verts2.push_back(transform2->indexToWorld((*vertlist)[i]));

// get all the quads and triangles from the polygon pool
for( size_t i = 0; i < mesher.polygonPoolListSize(); i++ ){

  for( size_t ndx = 0; ndx < (*facelist)[i].numTriangles(); ndx++ ){
    Vec3I *p = &((*facelist)[i].triangle(ndx));
    faces2.push_back( Vec4I(p->x(), p->y(), p->z(), INVALID_IDX) );
  }
  for( size_t ndx = 0; ndx < (*facelist)[i].numQuads(); ndx++ ){
    openvdb::Vec4I *p = &((*facelist)[i].quad(ndx));
    faces2.push_back( Vec4I(p->x(), p->y(), p->z(), p->w()) );
  }

  // print all information out
  for(size_t i = 0; i < verts2.size(); i++)
    std::cout << "vert " << i << " " << verts2[i] << std::endl;

  for(size_t i = 0; i < faces2.size(); i++)
    std::cout << "face " << i << " " << faces2[i] << std::endl;

  return 0;
}
```

Writing and Reading Grids

This is a pretty straight forward process. The only thing needed to be done is to register a bunch of types with the system before we can do any file I/O. Reading grids from a file is pretty simple too, however we need to be careful about how we read the grids since there could be multiple stored in the file.

```
#include <openvdb/openvdb.h>
#include <openvdb/tools/LevelSetSphere.h>
#include "Types.h"

int main(int argc, char** argv)
{
  // register openvdb types with the system
  openvdb::initialize();

  /*** File Output ***/
  // create a sphere level set
```

```
FloatGrid::Ptr grid = openvdb::tools::createLevelSetSphere<FloatGrid>(50.0, Vec3f(0,0,0), 0.5);

// set some optional meta data with the grid for identification.
// There are 4 types, GRID_UNKNOWN, GRID_LEVEL_SET, GRID_FOG_VOLUME, and GRID_STAGGERED.
grid->setGridClass(openvdb::GRID_LEVEL_SET);

// set a name for your grid.
grid->setName("sphereLevelSet");

std::cout << "grid name: " << grid->getName() << std::endl;
std::cout << "grid class: " << grid->getGridClass() << std::endl;

// create a VDB file object.
openvdb::io::File file("spherels.vdb");

// add the grid pointer to a container (since we can write multiple grids out).
openvdb::GridPtrVec grids;
grids.push_back(grid);

// write out the container and close the file.
file.write(grids);
file.close();

/*** File Input ***/
// open the file for reading
openvdb::io::File fileR("spherels.vdb");
fileR.open();

// we need to use the generic GridBase type to temporarily track
// the grid until we convert it over to its proper type.
openvdb::GridBase::Ptr base;

// iterate over all grids using their name in the file
// and retrieve the one named "sphereLevelSet".
for(openvdb::io::File::NameIterator iter = fileR.beginName(); iter != fileR.endName(); ++iter){
    std::cout << "Grid Name (iterator): " << iter.gridName()  << std::endl;
    if(iter.gridName() == "sphereLevelSet") base = fileR.readGrid(iter.gridName());
  }

  // we are done with the file, so close it and deregister types with the system
  fileR.close();
  openvdb::uninitialize();

  // convert the baseGrid over to the proper grid (assuming fog volume is a float grid).
  grid = openvdb::gridPtrCast<FloatGrid>(base);

  std::cout << "grid name (file): " << grid->getName() << std::endl;
  std::cout << "grid class (file): " << grid->getGridClass() << std::endl;

  return 0;
}
```

VolumeRayIntersector

The VolumeRayIntersector class is a useful class to quickly figure out intersting ranges along a given ray in a grid. Due to the way OpenVDB stores the grids, you might have either a tile or a series of leaves. A tile is an region of space in the grid that is a constant value. A leaf is stored value at that position. This class also performs a ray/bounding box intersection test, telling you if the ray intersects the volume at all.

```
#include <openvdb/tools/LevelSetSphere.h>
#include <openvdb/tools/LevelSetUtil.h>
#include <openvdb/tools/RayIntersector.h>
#include "Types.h"
```

```
int main(int argc, char** argv)
{
  // create a sphere fog volume
  FloatGrid::Ptr grid = openvdb::tools::createLevelSetSphere<FloatGrid>(50.0, Vec3f(0,0,0), 0.5);
  openvdb::tools::sdfToFogVolume<FloatGrid>(*grid);

  // construct volume ray intersector object
  openvdb::tools::VolumeRayIntersector<FloatGrid> vri(*grid);

  // construct a ray with origin (-100, 0, 0) and the default direction (1, 0, 0)
  openvdb::math::Ray<double> ray(Vec3d(-100,0,0));

  // before we begin a march we set the ray to march along
  // this function returns true if it intersects
  // this should intersect
  std::cout << ray << " intersect volume? " << vri.setWorldRay(ray) << std::endl;

  // march along the ray (note, an openvdb::Real = double)
  double t0, t1;

  // integrate between t0 and t1
  // note, return value tells you if there is any media along the ray
  // if return = 1 then value is constant between t0 and t1
  // if return = 2 then value is variable between t0 and t1
  // each consecutive call to march will give you the next bounding
  // region in the volume (ie, region that is variable or the same)
  while(int n = vri.march(t0, t1))
    std::cout << n << " " << t0 << " " << t1 << std::endl;

  return 0;
}
```


Category: OpenVDB