```
Copyright (c) 2011, Los Alamos National Security, LLC
                    reserved
Copyright 2011. Los Alamos National Security, LLC. This software was produced under U.S. Government contract DE-AC52-06NA25396 for Los Alamos National Laboratory (LANL), which is operated by Los Alamos National Security, LLC for the U.S. Department of Energy. The U.S. Government has rights to use, reproduce, and distribute this software.
 NEITHER THE GOVERNMENT NOR LOS ALAMOS NATIONAL SECURITY, LLC MAKES ANY WARRANTY, EXPRESS OR IMPLIED, OR ASSUMES ANY LIABILITY FOR THE USE OF THIS SOFTWARE.
 If software is modified to produce derivative works, such modified software should be clearly marked, so as not to confuse it with the version available from LANL.
Additionally, redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

Neither the name of Los Alamos National Security, LLC, Los Alamos National Laboratory, LANL, the U.S. Government, nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.
THIS SOFTWARE IS PROVIDED BY LOS ALAMOS NATIONAL SECURITY, LLC AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL LOS ALAMOS NATIONAL SECURITY, LLC OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 #ifndef MARCHING_CUBE_H_
 #define MARCHING_CUBE_H_
#include <thrust/copy.h>
#include <thrust/scan.h>
#include <thrust/transform_scan.h>
#include <thrust/binary_search.h>
#include <thrust/iterator/constant_iterator.h>
#include <thrust/iterator/zip_iterator.h>
 #include <piston/image3d.h>
#include <piston/piston_math.h>
#include <piston/choose_container.h>
#include <piston/hsv_color_map.h>
 #define MIN_VALID_VALUE -500.0
 namespace piston {
  template <typename InputDataSet1, typename InputDataSet2>
  class marching_cube
 public:
        typedef typename InputDataSet1::PointDataIterator InputPointDataIterator; typedef typename InputDataSet1::GridCoordinatesIterator InputGridCoordinatesIterator; typedef typename InputDataSet1::PhysicalCoordinatesIterator InputPhysCoordinatesIterator; typedef typename InputDataSet2::PointDataIterator ScalarSourceIterator;
        typedef typename thrust::iterator_difference<InputPointDataIterator>::type
        typedef typename thrust::iterator_space<InputPointDataIterator>::type space_type;
typedef typename thrust::iterator_value<InputPointDataIterator>::type value_type;
        \label{typedef}  \mbox{ typedef typename thrust::counting\_iterator<int, space\_type> CountingIterator;}
       VerticesContainer;
       typeder typename IableContainer::iterator
typedef typename VerticesContainer::iterator
typedef typename IndicesContainer::iterator
typedef typename NormalsContainer::iterator
typedef typename ScalarContainer::iterator
typedef typename ScalarContainer::iterator
TableIterator;
VerticesIterator;
IndicesIterator;
NormalsIterator;
ScalarIterator;
        static const int triTable_array[256][16];
static const int numVerticesTable_array[256];
                                                                         // scalar field for generating isosurface/cut geometry
// scalar field for generating interpolated scalar values
        InputDataSet1 &input:
        InputDataSet2 &source:
        value_type isovalue;
bool discardMinVals;
bool useInterop;
                                            triTable;  // a copy of triangle edge indices table in hostIdevice_vector
numVertsTable;  // a copy of number of vertices per cell table in hostIdevice_vector
        TableContainer
                                            case_index;  // classification of cells as indices into triTable and numVertsTable
num_vertices;  // number of vertices will be generated by the cell
        IndicesContainer
                                             valid_cell_enum; // enumeration of valid cells
                                             valid_cell_indices;
                                                                                         // a sequence of indices to valid cells
        IndicesContainer
        IndicesContainer
                                            output_vertices_enum; // enumeration of output vertices, only valid ones
#ifdef USE_INTEROP
  value_type minIso, maxIso;
  bool colorFlip;
  float4 *vertexBufferData;
  float3 *normalBufferData;
        float4 *colorBufferData;
int vpoSize;
Gluint vboBuffers[3];
struct cudaGraphicsResource* vboResources[3]; // vertex buffers for interop
#endif
                                           vertices;  // output vertices, only valid ones
normals;// surface normal computed by cross product of triangle edges
scalars;// interpolated scalar output
        VerticesContainer
        NormalsContainer
        unsigned int num_total_vertices;
#endif
               {}
        void freeMemory(bool includeInput=true)
               if (includeInput) {
                       case_index.clear();
                      num_vertices.clear():
                      valid_cell_enum.clear();
                 valid cell indices.clear():
               output_vertices_enum.clear();
vertices.clear();
               normals.clear();
               scalars.clear():
       }
        void operator()()
               const int NCells = input.NCells;
               case_index.resize(NCells)
               num_vertices.resize(NCells);
               "/thrust::copy(input.point_data_begin(), input.point_data_begin()+20, std::ostream_iterator<float>(std::cout, " ")); std::cout << std::endl; //std::cout << std::endl;
               // no valid cells at all, return with empty vectors.
if (num_valid_cells == 0) {
   vertices.clear();
   normals.clear();
   scalars.clear();
   return;
}
              // get the total number of vertices
               num_total_vertices = num_vertices[valid_cell_indices.back()] + output_vertices_enum.back();
                if (useInterop) {
#if USE_INTEROP

if (num_total_vertices > vboSize)

{
                        glBindBuffer(GL_ARRAY_BUFFER, vboBuffers[0]);
glBufferData(GL_ARRAY_BUFFER, num_total_vertices*sizeof(float4), 0, GL_DYNAMIC_DRAW);
if (glGetError() == GL_OUT_OF_MEMORY) { std::cout << "Out of VBO memory" << std::endl; exit(-1); }
glBufferData(GL_ARRAY_BUFFER, vboBuffers[1]);
glBufferData(GL_ARRAY_BUFFER, num_total_vertices*sizeof(float4), 0, GL_DYNAMIC_DRAW);
if (glGetError() == GL_OUT_OF_MEMORY) { std::cout << "Out of VBO memory" << std::endl; exit(-1); }
glBufferData(GL_ARRAY_BUFFER, vboBuffers[2]);
glBufferData(GL_ARRAY_BUFFER, num_total_vertices*sizeof(float3), 0, GL_DYNAMIC_DRAW);
if (glGetError() == GL_OUT_OF_MEMORY) { std::cout << "Out of VBO memory" << std::endl; exit(-1); }
glBindBuffer(GL_ARRAY_BUFFER, 0);
vboSize = num_total_vertices;
                       \begin{array}{c} {\rm cudaGraphicsMapResources(1,\ \&vboResources[2],\ 0);} \\ {\rm cudaGraphicsResourceGetMappedPointer((void\ **)\ \&normalBufferData,\ \&num\_bytes,\ vboResources[2]);} \\ \end{array} 
              } else {
                     vertices.resize(num_total_vertices);
normals.resize(num_total_vertices);
               //scalars.resize(num_total_vertices);
 // do edge interpolation for each valid cell
if (useInterop) {
#if USE_INTEROP
                      num_valid_cells,
                                                                                                                                    thrust::make_permutation_iterator(num_vertices.begin(), valid_cell_indices.begin()) +
num_valid_cells)),
                                                   thrust::transform(scalars.begin(), scalars.end()
                      } else {
                     num_valid_cells,
                                                                                                                                   thrust::make_permutation_iterator(num_vertices.begin(), valid_cell_indices.begin()) +
 num_valid_cells)),
                                                   }
        struct classify_cell : public thrust::unary_function<int, thrust::tuple<int, int> >
               // FixME: constant iterator and/or iterator to const problem.
InputPointDataIterator point_data;
const float jovalue;
const bool discardMinVals;
               TableIterator
                                                           numVertsTable;
               const int xdim;
                const int ydim;
               const int cells_per_layer;
const int points_per_layer;
              discardMinVals(discardMinVals),
                                               numVertsTable(numVertsTable),
xdim(input.dim0), ydim(input.dim1), zdim(input.dim2),
cells_per_layer((xdim - 1) * (ydim - 1)),
points_per_layer (xdim*ydim) {}
               _host__ _device__
thrust::tuple<int, int> operator() (int cell_id) const {
    // FIXME: this integer division/modulus is repeated at every
    // instance of the input iterator when the scalars are computed
                      // on the fly.

const int x = cell_id % (xdim - 1);

const int y = (cell_id / (xdim - 1)) % (ydim -1);

const int z = cell_id / cells_per_layer;
                     // FIXME: there is too much redundant computation to get
// triple (col, row, layer) in the input iterator when data
// is calculated on the fly
const float f0 = *(point_data + i0);
const float f1 = *(point_data + i1);
const float f2 = *(point_data + i2);
const float f3 = *(point_data + i3);
const float f4 = *(point_data + i4);
const float f5 = *(point_data + i5);
const float f6 = *(point_data + i6);
const float f7 = *(point_data + i7);
                     unsigned int cubeindex = (f0 > isovalue);
cubeindex += (f1 > isovalue)*2;
cubeindex += (f2 > isovalue)*4;
cubeindex += (f3 > isovalue)*8;
cubeindex += (f4 > isovalue)*16;
cubeindex += (f5 > isovalue)*32;
cubeindex += (f6 > isovalue)*64;
cubeindex += (f7 > isovalue)*128;
                      //bool valid = (!discardMinVals) || ((f0 > MIN_VALID_VALUE) && (f1 > MIN_VALID_VALUE) && (f2 > MIN_VALID_VALUE) && (f3 > MIN_VALID_VALUE) && (f4 > MIN_VALID_VALUE) && (f6 > MIN_VALID_VALUE) && (f7 > MIN_VALID_VALUE));
                     return thrust::make_tuple(cubeindex, /*valid*/numVertsTable[cubeindex]);
             }
       };
        struct is_valid_cell : public thrust::unary_function<int, bool>
               _host_ _device_
bool operator()(int numVertices) const {
    return numVertices != 0;
       };
        struct\ isosurface\_functor\ :\ public\ thrust::unary\_function<thrust::tuple<int,\ int,\ int,\ int>,\ void>\{
               // FixME: constant iterator and/or iterator to const problem.
               // HIXME: Constant iterator and/or iterator inputPointDataTierator point_data; InputPhhysCoordinatesIterator physical_coord; ScalarSourceIterator scalar_source; const float isovalue; TableIterator triangle_table;
               typedef typename InputPhysCoordinatesIterator::value_type grid_tuple_type;
               float4 *vertices_output;
                float3 *normals_output;
               float *scalars_output;
               const int xdim;
const int ydim;
const int zdim;
const int cells_per_layer;
              TableIterator triangle_table,
float4 *vertices,
float3 *normals/*,
float *scalars*/)
                       : point_data(input.point_data_begin()),
physical_coord(input.physical_coordinates_begin()),
scalar_source(source.point_data_begin()),
                         isovalue(isovalue),
triangle_table(triangle_table),
vertices_output(vertices), normals_output(normals), /*scalars_output(scalars),*/
xdim(input.dim0), ydim(input.dim1), zdim(input.dim2),
cells_per_layer((xdim - 1) * (ydim - 1)) {}
               Tloat3 vertex_interp(float3 p0, float3 p1, float t) const {
   return lerp(p0, p1, t);
                  host
                                  device
               float scalar_interp(float s0, float s1, float t) const {
    return lerp(s0, s1, t);
               // FixME: the type of the grid coordinates may not be 3-tuple of ints
               template <typename Tuple>
__host__ __device__
              __host___device__
void operator()(thrust::tuple<int, int, int, int> indices_tuple) {
   const int cell_id = thrust::get<0>(indices_tuple);
   const int outputVertId = thrust::get<1>(indices_tuple);
   const int cubeindex = thrust::get<2>(indices_tuple);
   const int numVertices = thrust::get<3>(indices_tuple);
                     const int x = cell_id % (xdim - 1);
const int y = (cell_id / (xdim - 1)) % (ydim -1);
const int z = cell_id / cells_per_layer;
                      i[4] = i[0] + xdim * ydim;
i[5] = i[1] + xdim * ydim;
i[6] = i[2] + xdim * ydim;
i[7] = i[3] + xdim * ydim;
                      float f[8];

f[0] = *(point_data + i[0]);

f[1] = *(point_data + i[1]);

f[2] = *(point_data + i[2]);

f[3] = *(point_data + i[3]);

f[4] = *(point_data + i[4]);

f[5] = *(point_data + i[5]);

f[6] = *(point_data + i[6]);

f[7] = *(point_data + i[7]);
                     // TODO: Reconsider what GridCoordinates should be (tuple or float3) float3 p[8]; p[0] = tuple2float3(*(physical_coord + i[0])); p[1] = tuple2float3(*(physical_coord + i[1])); p[2] = tuple2float3(*(physical_coord + i[2])); p[3] = tuple2float3(*(physical_coord + i[3])); p[4] = tuple2float3(*(physical_coord + i[4])); p[5] = tuple2float3(*(physical_coord + i[5])); p[6] = tuple2float3(*(physical_coord + i[6])); p[7] = tuple2float3(*(physical_coord + i[7]));
                       /*float s[8];
                      /*floot s[8];

s[0] = *(scalar_source + i[0]);

s[1] = *(scalar_source + i[1]);

s[2] = *(scalar_source + i[2]);

s[3] = *(scalar_source + i[4]);

s[4] = *(scalar_source + i[4]);

s[5] = *(scalar_source + i[5]);

s[6] = *(scalar_source + i[6]);

s[7] = *(scalar_source + i[7]);*/
                      // interpolation for vertex positions and associated scalar values
for (int v = 0; v < numVertices; v++) {
    const int edge = triangle_table[cubeindex*16 + v];
    const int v0 = verticesForEdge[2*edge];
    const int v1 = verticesForEdge[2*edge + 1];
    const float t = (isovalue - f[v0]) / (f[v1] - f[v0]);
    *(vertices_output + outputVertId + v) = make_float4(vertex_interp(p[v0], p[v1], t), 1.0f);
    /*(scalars_output + outputVertId + v) = scalar_interp(s[v0], s[v1], t);
}</pre>
                      // generate normal vectors by cross product of triangle edges
for (int v = 0; v < numVertices; v += 3) {
    const float4 *vertex = (vertices_output + outputVertId + v);
    const float3 edge0 = make_float3(vertex[1] - vertex[0]);
    const float3 edge1 = make_float3(vertex[2] - vertex[0]);
    const float3 normal = normalize(cross(edge0, edge1));
    *(normals_output + outputVertId + v) =
    *(normals_output + outputVertId + v + 1) =
    *(normals_output + outputVertId + v + 2) = normal;
}</pre>
                    }
            }
        VerticesIterator vertices_begin() {
               return vertices.begin();
        VerticesIterator vertices_end() {
               return vertices.end();
       NormalsIterator normals_begin() {
    return normals.begin();
        NormalsIterator normals_end() {
               return normals.end();
       }
```

ScalarIterator scalars\_begin() {
 return scalars.begin();
}
ScalarIterator scalars\_end() {
 return scalars.end();

void set\_isovalue(value\_type val) {

#endif /\* MARCHING\_CUBE\_H\_ \*/

};