Regulus : Degenerate Triangulations on the GPU

Christian Mazakas

Abstract

We present a method for creating tetrahedral meshes on the GPU using Nvidia's CUDA technology.

Much of the behavior of the code is an emulation of gFlip3D's core triangulation routine. The paper can be found at : `http://www.comp.nus.edu.sg/~tants/gdel3d.html` and is definitely worth a good reading.

1 Introduction

When I was still an undergraduate, my thesis advisor mentioned a new simulation software in the cosmological field. Its name was Arepo and once I read the paper, I knew what I had to do. I was to channel my growing love of computer science into something that I thought was visually appealling and challenging to solve.

Tetrahedral meshes have very niche use. They are a common choice for space-discretized solvers (finite-element and -volume methods) for various physics equations. Arepo does this with how it treats the flow of gases in galactic evolution though it uses a different type of finite-element/-volume. They are also useful for reconstructing surfaces and imaging. Much to my own amusement, the red rabbit sculpture, Leap, hanging in the Sacramento Airport is a reconstruction done via a tetrahedral mesh.

My interests have never truly lied with the application of meshes however. Instead, I think they are incredibly fun to create and doing so on the GPU posed an interesting challenge that attracted me.

2 Tetrahedra

A tetrahedron is a 3d shape defined by 4 points. It has 4 faces, each being a triangle, and 6 edges.

2.1 Orientation

We define a tetrahedron in Regulus as *abcd* or *0123*. This is key because we use this as our reference frame or view of the tetrahedron. In Regulus, we say that a tetrahedron abcd is positively oriented if d is above the plane spanned by the points abc where abc is ordered in a counter-clockwise fashion.

What this translates to is, if the vertices of a tetrahedron were (0, 0, 0), (9, 0, 0), (0, 9, 0) and (0, 0, 9), the face abc would be drawn with (0, 0, 0) to (9, 0, 0) and then from (9, 0, 0) to (0, 9, 0) and from there back to (0, 0, 0). (0, 0, 9) would be "above" the plane spanned by the triangle which in this case is simply the xy-plane and any point with a positive z-coordinate value is intuitively "above" the plane.

There are methods for determining whether or not a tetrahedron has a positive orientation. We use the result of the following determinant :

$$orientation(abcd) = \begin{vmatrix} 1 & a_x & a_y & a_z \\ 1 & b_x & b_y & b_z \\ 1 & c_x & c_y & c_z \\ 1 & d_x & d_y & d_z \end{vmatrix} \tag{1}$$

If the result is greater than 0, the orientation is posive and is negative if the result is less than 0. If the determinant is exactly 0, it is on the plane itself. Using our previous example, if vertex d was (3, 3, 0), then the orientation of abcd would be 0 which makes sense considering d would live directly in the triangle abc. Note that orientation and volume are connected.

We choose to represent all of our tetrahedra in Regulus as positively oriented. Note that it doesn't ultimately matter whether we choose positive or negative so long as we are consistent, i.e. we shouldn't mix positive and negative tetrahedral representations in the same mesh.

2.2 Tetrahedral Fractures

With meshing, one of the first questions to arise is, "How do we construct a mesh in the first place?" There many ways of doing this though

the most popular one in computing is to "insert" points into an already existing mesh which then produces more tetrahedra. This commonly referred to as "flipping" but I've always prefered the term "fracture" because I feel it's more descriptive of the event.

A point is inside a tetrahedron if it is above all 4 faces of the tetrahedron or rather, the point is positively oriented with regard to each face. Each face of a tetrahedron is represented in Regulus as 321, 023, 031 and 012. The faces are then stored in the order of the point opposite them. For example, vertex 0 is the point opposite face 321, vertex 1 is opposite 023, etc.

When we "insert" a point into the mesh, the enclosing tetrahedron is split apart or "fractured" by the point and 4 new tetrahedra are created, one for each face the point is above. The point being inserted becomes a vertex for new tetrahedra.

2.2.1 Types of Fractures

There are 3 different types of fractures, depending on where the point being inserted lies relative to the tetrahedron.

A point is inside a tetrahedron if it is above all 4 faces. A point is on a face of a tetrahedron if it is on 1 face and above 3. A point is on an edge if it is on 2 faces and above 2. A point is on top of a vertex if it is on 3 faces and above 1. A point is outside a tetrahedron if it is below any face (negatively oriented).

We only insert a point if it is inside or on a face or edge of a tetrahedron and the number of faces the point is above determines the amount of new tetrahedra to create. This means a point inside yields a 1-to-4 fracture, a point on a face yields a 1-to-3 fracture and an edge fracture is 1-to-2.

Consider the following,

$$0123 + p = \{321p, 023p, 031p, 012p\} \tag{2}$$

This a full 1-to-4 flip with tetrahedron 0123 and p. 1-to-3 and 1-to-2 fractures are similar except that select tetrahedra will be omitted from the above list (the ones where p is on that face).

3 Algorithm

3.1 Preliminaries

To understand how the point insertion algorithm works, we have to begin with how bookkeeping will be done. We choose a 5-tuple of integer arrays. We denote the elements of the tuple with arrays representing point association, tetrahedron association, location association, fracture size and a nominated switch.

Internally, these are stored as pa, ta, la, fs and nm. The data is meant to be read as, point pa[i] is inside/on tetrahedron ta[i] with location association la[i], potential fracture size fs[i] with a nomination flag determined by nm[i] (i.e. is this a point we will insert into the mesh this round?).

It is also worth noting that we use the Shewchuk predicates adapted to CUDA by the creators of gFlip3D for all of our orientations tests. See the source code (either mine or theirs).

Regulus also begins with an all-encompassing tetrahedron that contains the entire point set to be inserted. In our case, we use a Cartesian grid distribution as our point set.

3.2 Data Structures

The structures that we use in Regulus are relatively simple and are seemingly standard of meshing algorithms. Our tetrahedron structure stores an internal array of 4 integers and our point structure stores an array of either 3 floats or 3 doubles. In the code these look like :

```
struct tetra
{
        int v[4];
};

struct point
{
        real c[3];
};
```

Here the real type in the point structure is assumed to be the result of a typedef somewhere off-screen.

Each integer in the tetrahedron structure represents the location of the point in the point buffer so a tetrahedron of value { 0, 1, 2, 3 } is a tetrahedron with vertices at 0, 1, 2, and 3 in the point buffer.

3.3 Nominating Points

One of the first challenges of creating a mesh is how to decide which points we will insert. We want to maximize the inherent parallelism of the GPU so the natural idea is to insert as many points as possible into the mesh at once. This can be problematic, however.

If a point is on a face, there's a good chance that face will be shared between two tetrahedra. This means that if we insert points without any form of checking, we're likely to fracture the same tetrahedron twice and not in the correct serialized order either. This is even worse in the case of an edge because any number of tetrahedra can share an edge (this is most easily visualized as a bouquet of tetrahedra).

To solve this in Regulus, we allocate two arrays. We use one array to store a series of locks that we check and set with atomic compare-and-swap operations that are built into CUDA and an integer array that stores tuple indices.

In order to effectively use these, we elect to use the bucket hashing technique presented by Alcantara (`http://idav.ucdavis.edu/~dfalcant/downloads/dissertation.pdf`). We use thrust::sort_by_key() to sort our tuple by pa (or the point id). This means we treat points as buckets and the tetrahedra they fracture as the bucket contents.

We launch a kernel with one thread per bucket and begin iterating the contents. We perform an atomicCAS() at the index of our lock array taken from the bucket contents. In the code, this looks like :

```
for (int i = begin; i < end; ++i)
{
        // get id of tetrahedron
        const int ta_id = ta[i];

        // check the lock...
        const int old = atomicCAS(locked_tetra + ta_id, 0, 1);

        // use this to filter out "bad" threads (i.e. the bucket
        // contents intersect)
        if (old == 1)
                return;
}
```

It may not be particularly obvious but Regulus chooses points based on when the atomicCAS is executed so point nomination is effectively random. Whichever thread gets to a lock first determines which points will be nominated. In practice, however, this proves to have little impact on the runtime or performance of the algorithm.

Our second array is an inverse of the array ta. Instead of ta[tuple_id] = ta_id, we use a new array perhaps improperly named nominated_tetra such that nominated_tetra[ta_id] = tuple_id. We write to this array in the code as follows:

```
// now, these are the only threads alive
for (int i = begin; i < end; ++i)
{
        const int ta_id = ta[i];

        // default value
        assert(nominated_tetra[ta_id] == -1);

        // we store the location of the nominated tetrahedron
        // in ta and index it by the tetrahedron id
        // nominated_tet[0] = 17; <-- tetrahedron 0 has assocation
        // array data stored at index 17
        // (pa[17], la[17], fs[17], ta[17] == 0)
        nominated_tetra[ta_id] = i;

        // then write to special nomination array
        nm[i] = 1;
}
```

Because ta_id is unique across all threads, all writes are safe.

3.4 Inserting Points

This is a much simpler portion of the code. We simply write 4 new tetrahedron structures to the stack of each thread and then load them all to global memory at once.

To account for degeneracies, we load in the properly indexed value of la and iterate the first 4 bits, checking if each one is set or not and then creating new tetrahedra with the proper faces and the insertion point.

The only difficult part is in determining the write indices for the mesh buffer. To do this, we use a modified prefix sum by multiplying

each element of nm with fs before adding it to the sum.

This is because the fracture size determines how far past the end of the buffer we will need to write. The first write is done in-place while all subsequent ones are done beyond the end of the mesh buffer. We pre-allocate the buffer to have a generous amount of room before even beginning the triangulation process. So far this has not proven to be overly prohibitive.

3.5 Point Redistribution

This part of the software absolutely took me the longest to figure out how to solve and I had to break it up into three kernels to do so.

The basic idea is, we want to only test all the points *not* nominated for insertion and test the tetrahedra that were fractured. We then need a way of writing this data back to our 5-tuple.

Our first kernel is relatively simple. We use one thread for each element of the tuple and check it this index was nominated or not. If it's not, we continue. We load in the tetrahedron id and using nominated_tetra, we can see if the tetrahedron of this tuple was fractured or not. If the tetrahedron was fractured, we load the fracture size and store it in a separate array.

We then perform a prefix sum of the aforementioned array. This gives us our write indices. We now need to perform the actual calculations which is what our third and final kernel is for. We load all the tetrahedra from the fracure and test them against the point. After all the tests are done, we write back to the 5-tuple array which was pre-allocated with sufficient storage.

We must perform 4 orientation tests to determine the location assocation and subsequently the fracture size. This an expensive but unavoidable task for the triangulation to successfully complete. We store no values if any of the orientation test return negative.

4 Wrapping Up

So, I guess that's about it, really. We nominate, insert and redistribute. Note, adjacency relations are not yet part of the code. This is because they are technically unnecessary for the triangulation routine to succeed which makes this algorithm unique in the sense that jump-and-walk algorithms absolutely require adjacency relations to work.

A simple brute-foce method I devised was to complete the triangulation process and then hash each face of every tetrahedron using MurmurHash3 and then sort relevant data (tetrahedral id and face id) by its associated face-hash. This way, we can easily establish which tetrahedra are linked to each other (the same faces yield the same hash) by checking neighboring values once the sorting is done. If a hash matches, we have a matching faces and the adjacency relation can be created.

4.1 The Future Is Always Bright!

I want to eventually add Delaunay refinement but that is far off in the future. As of now, I want to focus on the speed of just the raw triangulation so I'm always looking for feedback from other developers.

I would also like to eventually add a Voronoi mesh construction routine on top of all of that but I'm beginning to doubt whether or not this is advisable to do on the GPU.