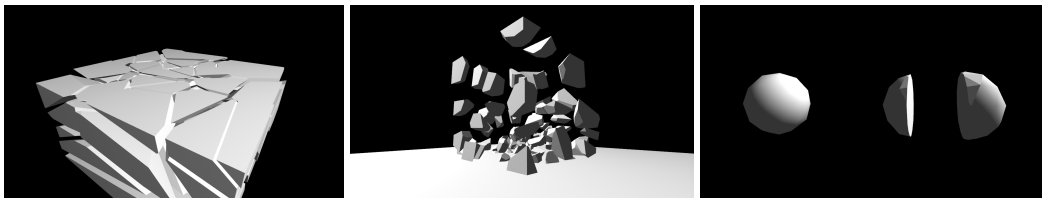
BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2018
http://excel.fit.vutbr.cz

Brittle Body Simulation on GPU

Tomáš Chlubna*



Abstract

While rigid body simulations are common topic of many papers and tutorials, brittle body simulation techniques are not that well described. One reason is that the brittle body simulation extends the former one and is more complex. This situation is even more visible in the world of GPGPU programming. The role of GPU is getting more and more significant even for non-rendering computations such as physics. The main feature of brittle body is the ability to shatter into pieces due to physical interactions with the scene. The main topic discussed here is the way to compute a Voronoi diagram and to use it in order to simulate the object shattering, all on GPU. This paper also mentions some additional techniques related to scene representation on GPU. The goal was to utilize the GPU parallelism as much as possible to make the simulation run in real-time. The object shattering should be accurate and the number of fragments is limited by the GPU architecture such as maximal workgroup count or local memory size. The simulation runs in real-time when dealing with a normal game-like 3D model with up to 100 fragments even in this alpha version. This article describes an efficient way to build a brittle body simulation engine, mainly on GPU and focuses on the Voronoi computation, offering a highly parallel way to build the diagram and split the 3D model into fragments.

Keywords: Brittle body — physics — voronoi — GPU

Supplementary Material: *N/A*

*xchlub00@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

The GPGPU calculations are nowadays an important part of physical, game and other simulation based engines. GPU usually has to hold the simulation objects' surface representations in its memory in order to render them while CPU uses the same data for the physics calculations. CPU then has to update the scene representation on GPU according to the simulation step results. However, this GPU-CPU communication is the bottleneck of the whole process. In order to minimize this communication, most of the simulation calculations are moved to GPU. To be able to do that, some algorithms have to be redesigned from the CPU versions (mostly sequential running on a fast processor) to GPU versions (highly parallel running on many slower processors).

The topic of this work is a brittle body simulation. There are many problems that need to be solved in order to efficiently implement such simulation on GPU. This article describes a basic structure of such application, focusing on the problem of 3D model shattering based on Voronoi diagram. Some other related problems and their solutions are mentioned here as well such as the whole GPU scene representation, collision detection and optimization techniques.

Rigid body simulations are already being imple-

mented on GPU be it partially or fully. Some well-known libraries such as Bullet Physics [1], implement many calculations on GPU. Many games also used the PhysX system by Nvidia to accelerate their physical calculations. There are also ways to simulate some physical phenomenons such as fluids, but also brittle bodies by particles [2]. The commonly used approach to construct a Voronoi diagram is by extracting it from Delaunay triangulation, which can be parallelized to efficiently run on the GPU [3]. Some other algorithms such as Lloyd's can be implemented on GPU as well [4].

My work presents efficient, highly parallel set of algorithms for GPU brittle body simulations, optimizing the CPU-GPU communication to minimum, thus leaving the CPU almost free for additional computations such as AI in games (application structure in Fig. 1). Aside from this general concept, this paper also presents a new way of Voronoi diagram construction in 3D called Box Cutting Algorithm. While Delaunay approaches are originally incremental and the parallelization brings many problems, this algorithm distributes the work on GPU efficiently right from the beginning and it constructs Voronoi diagram directly without the need of Delaunay-Voronoi duality property. The Voronoi diagram cells are then used to split original geometry using the Box Cutting Algorithm principles. The communication with the GPU is implemented using OpenGL 4.6 compute shaders.

2. Scene representation on GPU

As mentioned above, the whole scene needs to be stored in the GPU's memory. Most of the data is stored in global memory since this memory is available to all the workgroups in a coherent way and is big enough.

Here is the layout of global memory for a simple scene (note that all the buffers below need to have some additional free space to be able to add new objects in runtime):

- Vertex buffer {position: 3xfloat32, normal: 3xfloat32, texture coordinates: 2xfloat32} - vertices of all scene models
- Element buffer {1xuint32} - indices for all scene models (indexed rendering)
- Draw commands {5xuint32} - draw command for each model (indirect rendering, ideally with one API function call from CPU per frame)
- Object information {21xfloat32, 3xuint32 (might differ according to the desired attributes)} - holds each object's attributes (position, velocity, mass...)
- Model matrices {4x4xfloat32} - used in vertex shader, one matrix per object

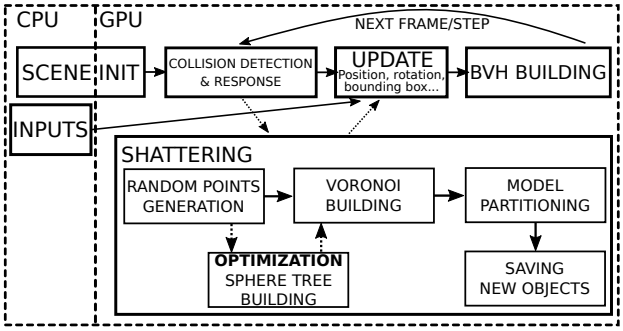


Figure 1. The whole scene is loaded and initialized on CPU, then transferred to the GPU's global memory along with an additional free space forward allocations. After that, CPU only controls the main rendering loop and handles the user inputs. Object informations on GPU contain some basic physical attributes such as velocity, position, mass etc. They get updated in each frame solely on GPU. This update also involves a bounding box calculations and BVH construction [5]. When needed, a special kernel generates random future fragments' centers (according to a collision contact point and strength), then it generates the actual 3D Voronoi diagram cells using the Box Cutting Algorithm discussed below and it divides the original geometry into pieces which are then included in the simulation as new objects (which can be shattered again).

- Bounding boxes {3x2xfloat32 (xyz bounds)} - for collision detection etc., one box per object
- General buffer - Free space that can be reused by multiple compute shaders and holds all the additional structures (in our case it might contain Voronoi cell centers, tree structures, morton codes...)

3. Voronoi diagram with Box Cutting Algorithm kernel

Voronoi diagram simulates the shattering and other physical phenomena pretty well which is why it's used for this type of problems. It partitions the space into cells where each point of the space gets assigned to its closest cell center, also called Voronoi site. Here follows the formal definition:

$$V(s_i) = \{x \in R^n : |s_i - x| \leq |s_l - x| \forall s_l \in S\} \quad (1)$$

where

- $V(s_i)$ is the Voronoi cell (part of the space aka set of points) assigned to the site s_i
- S is set of the Voronoi sites (cell centers)
- R^n is set of n-dimensional space points

Voronoi construction is a crucial part of the whole shattering process. When the object gets into a collision that is powerful enough to make it break into pieces a set of voronoi sites is generated. In order to avoid certain degeneracies later (Fig. 10), it's necessary to generate the sites inside the model geometry. 2 Uniform distribution pseudo-random generator is using the sinus error sampling method described in code 1.

```
A = 12.9898;
B = 78.233;
C = 43758.5453;
fract(sin(dot(coord.xy ,vec2(A,B))) * C);
```

Listing 1. Pseudo-random number generator for GPU [6]

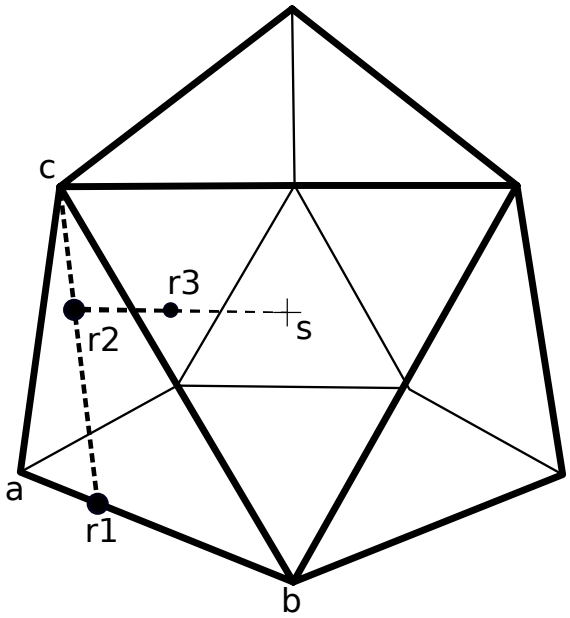


Figure 2. To generate a random point inside model, one random triangle is chosen. Line segment *ab* is sampled for a random point $r1$, lying on it. The same is done to $cr1$ segment and subsequently to $sr2$ (where s is bounding box center), resulting in point $r3$ which is the result. In order to generate non-uniform set of points (sites are closer to the collision point), utilizing the GPU parallelism, it's possible to generate 8 candidates per site and choose the closest one to the wanted position with a probability derived from the collision strength. (having 256 threads per workgroup, resulting in 32 points (256/8) which can be chosen from the candidates in one warp)

Each workgroup computes the shape of one Voronoi cell which is a list of edges, associated with connected planes (each edge has exactly two adjacent planes). So first thread loads it's previously generated center. After that the cell shape is initialized as the model bounding box. From now on, the algorithm works with cell-space coordinates (Fig. 3). The cutting planes can be stored in local memory as 3D vectors as:

$$\vec{p} = \{x, y, z\}, plane = \{\vec{p}, \frac{\vec{p}}{|\vec{p}|}\} \tag{2}$$

where

- *plane* is represented as position of one of its points and normal, pointing outside the cell

A nearest neighbouring point from the generated Voronoi sites is loaded and the plane between the cell center and this point is initialized (lies in the middle between them with normal having the same direction as the vector defined by those two points like in Fig. 3). This plane is then used to cut the cell (initially the bounding box) (Fig. 4).

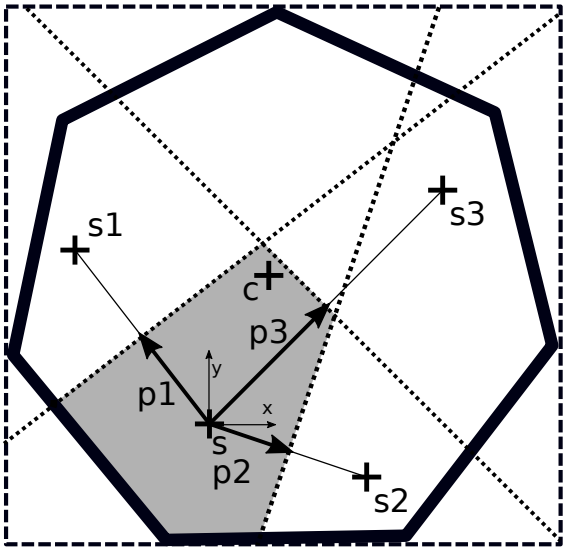


Figure 3. An object (thick polygon) in its bounding box (dashed rectangle). The figure shows the planes of the Voronoi cell (gray area), represented as vectors *pn*. The plane cuts in half the line connecting current Voronoi site s with the neighbouring site *sn*. All coordinates are in cell-space with origin in s, while the model-space has it's origin usually in the middle of the geometry c. The dotted lines, perpendicular to the plane vectors are the actual planes, defining the cell shape.

The algorithm 1 doesn't have to go through the whole set of Voronoi sites. It keeps a track of the so called maximal distance which is the distance from the cell center to the furthest cell vertex. This distance can be calculated exploiting the parallel reduction pattern. If the new plane lies outside the maximal distance radius, the plane is too far to cut anything and no other points can give us such plane. The maximal distance is recalculated after each cut and might decrease as in Fig. 5.

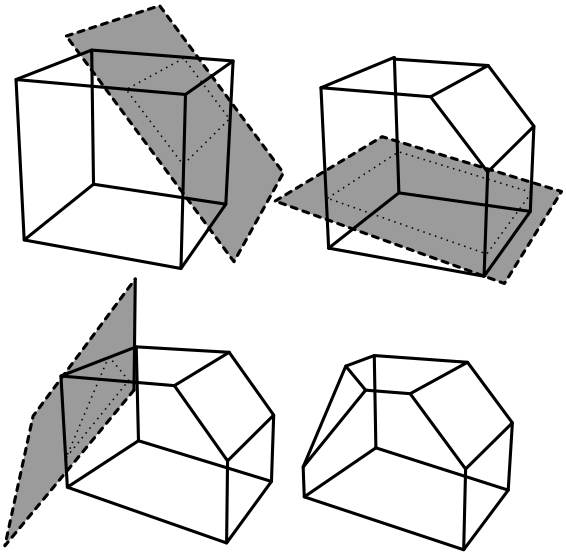


Figure 4. Three iterations of Box Cutting Algorithm. Starting from the model bounding box it iteratively creates the Voronoi cell, cutting the shape by the plane separating the current site and its nth nearest neighbour in the middle between them.

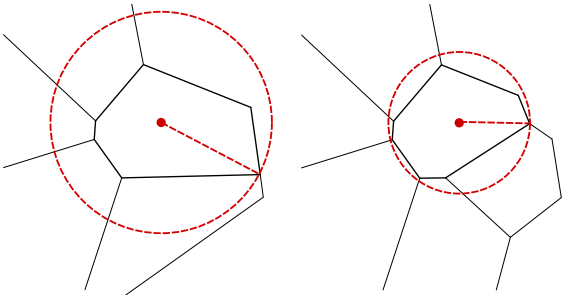


Figure 5. Maximal cell distance shrinking when the cell gets smaller due to a cut in 2D

The actual cutting is performed in parallel where each thread handles one cell edge. All the cell vertices are checked in parallel testing on which side of the plane they lie. For each edge the thread decides if both it's vertices lie outside the new cutting plane which would mark the edge as eliminated. If both edge vertices are on the other side of the plane (aka inside the cell), the edge is kept and if one of them is outside, then the splitting is performed. Both operations are implemented in listing 2.

```
//input: vertex v, plane p
bool orientPositive(vec3 v, vec3 p)
{return dot( normalize(p), v-p ) > 0.0;}

//input: edge points a,b, plane p
vec3 splitPoint(vec3 a, vec3 b, vec3 p)
{
        vec3 n = normalize(p);
        float u = dot(n, p-a)/dot(n, b-a);
        return a + u*(b-a);
}
```

Algorithm 1: BCA pseudocode to create the Voronoi cell

```
Data: set of generated Voronoi sites, model
      bounding box
Result: one Voronoi cell represented as list of
        edges, vertices and planes
site = getCellCenter(WorkGroupId);
initBoundingBoxCell();
avaliablePlanes = true;
while avaliablePlanes do
    maxDist = cellFurthestPoint();
    neighbor = nextNeighbour(site,maxDist);
    if neighbor != null then
        plane = (neighbor-site)/2.0;
        checkCellVerticesParallel(plane);
        splitEdgesParallel();
        createHullEdgesParallel();
    else
        avaliablePlanes = false;
    end
end
```

}

Listing 2. Vertex-plane orientation predicate and edge splitting function returning the split point

In the split case, a new vertex is calculated. All new vertices are stored and when all edges are checked, the newly created vertices need to be connected by new edges. In other words, a convex hull is created of those new vertices. Please note that there is no need to utilize any convex hull algorithms working with the coordinates since it's possible to connect the vertices with a common plane according to the edges (Fig. 6). A synchronization is necessary after the cut in a form of shared (local) memory barrier. In order to prevent a memory overflow, cleaning of the eliminated edges and vertices is performed in parallel after the cut by reinserting the active ones in the array getting their new indices by incrementing an atomic counter.

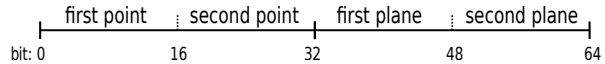


Figure 6. The edge representation stored in local memory as one uvec2 (2x32 unsigned int). Each 16bit short value is an index to the vertex/plane array.

4. Nearest neighbour search

When searching for the next neighbour (Algorithm 1) the algorithm starts looking for a point from the input sites that is further from the current cell center than

the last one and in the radius defined by the maximal cell distance. When the number of sites is sufficiently small, optimal way is to load them in local memory and go through them in a linear way. This however can't be used for hundreds of points in order to keep the simulation run in real time. Thus a spatial partitioning-like structure is needed. K-d tree or octree construction might be too complex and just the construction might slow the process down a lot. A fast alternative is a lightweight sphere tree [7] that is not as accurate as the former mentioned ones but might cull large parts of the searched space.

Here is the construction algorithm (Fig. 7):

1. Choose two random points which create a root
2. For each point in parallel save the root into it's path buffer
3. Do until there are points left:
   (a) For each point in parallel take the node in the path buffer
   (b) Decide which of the two node's points is closer to the current point
   (c) Now every point is assigned to a node and to one of its points, if there are multiple points choose one for each side (by minimal ID for deterministic way or random)
   (d) Save each point's assignment to the path buffer (to avoid traversal from the root in the next iteration)
   (e) The maximal number of new nodes is nth power of two where n is the number of current tree level. The assigned points are inserted into the nodes resulting in one or two new child nodes per parent.
   (f) The internal points of the new nodes are always the initial one from the parent and the inserted one.
4. Calculate the distance from the initial point of each node to it's furthest child's point

When searching for the next neighbour only the parts of the tree intersecting a search sphere with radius of the maximal cell distance are being searched while this radius is shrinking when a closer point is found. The algorithm stops when it reaches the inner limit sphere with radius of the distance from the cell center to the previously found point. When the searching stops, the last but one point that caused search sphere shrinking is returned.

5. Model geometry splitting

The naive way of creating one fragment would be to load all the model's vertices and iterate over cell planes.

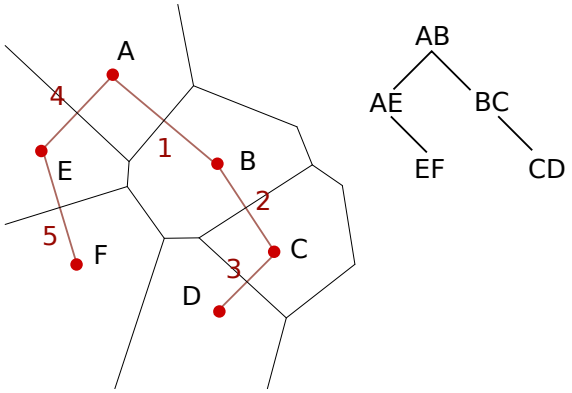


Figure 7. 2D visualization of the point sphere tree, Voronoi on the left and corresponding tree on the right (the numbers describe the order of insertions)

Each plane would split the model in two halves (checking on which side of the plane does each triangle lie). The triangles intersecting the plane would be simply cut resulting in one or two new triangles. This approach would not be efficient on GPU because of the need to transfer a lot of data from the global memory when copying the model geometry (when dealing with a dense triangulations it would also be impossible to use the local memory). It would also be necessary to triangulate the hole after each cut so the resulting triangulation might not be optimal either due to multiple cuts in the already triangulated segments.

The parallel approach discussed here is based on a bit map stored in a local memory, where each bit represents one triangle. In this way, it's possible to make each thread check one triangle (when the number of triangles is higher than workgroup size, it's necessary to do this in multiple iterations) and decide if it's outside or inside the cell, iterating over all cell planes. When the triangle is marked as outside, there is no need to keep iterating over the rest of the planes since this triangle is excluded. Inside triangles are copied to the cell geometry and intersecting ones are being cut and retriangulated like in Fig. 8.

All that is left is to modify the cell shape in order to prevent it to overlap the model surface. In other words, to cut the cell by the planes defined by the triangles that were split in the previous phase. This is now fairly straightforward since all that is needed is to insert those additional planes in the Box Cutting Algorithm (while marking them as invalid to prevent their triangulation). The following equation shows how to get the needed cell-space triangle plane:

$$\vec{n} = (b-a) \times (c-a), \hat{n} = \frac{\vec{n}}{|\vec{n}|}, \vec{p} = |-a \cdot \hat{n}|\hat{n} \quad (3)$$

where

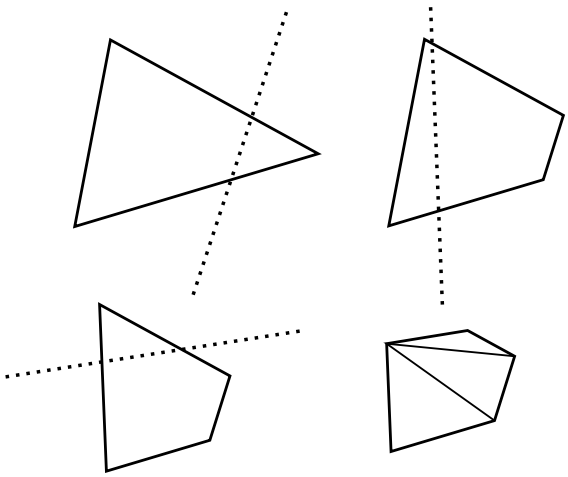


Figure 8. When the triangle intersects the plane it's necessary to decide the resulting shape of the polygon that will come out of this triangle due to possible multiple plane cuts. A way to achieve that is similar to the BCA where a set of edges is being kept (three at first for the original triangle) and each plane intersection might split those edges, resulting in one new edge (can be imagined as a 2D polygon-line intersection). At the end a triangulation of this resulting polygon is needed. This figure shows three plane intersections and the final triangulation. (floating point precision errors might be avoided by using the EPS tollerance)

- a, b, c are triangle vertices in cell-space (current site coordinates are subtracted from the model-space vertex coordinates)
- $\hat{n}$ is normalized triangle normal $\vec{n}$
- $\vec{p}$ is the resulting plane

The whole process is illustrated in Fig. 9. Then the valid cell faces are being triangulated. A simple way to triangulate a convex polygon is to chose one of its vertices and connect it with all the other edges in the polygon that are not containing this vertex.

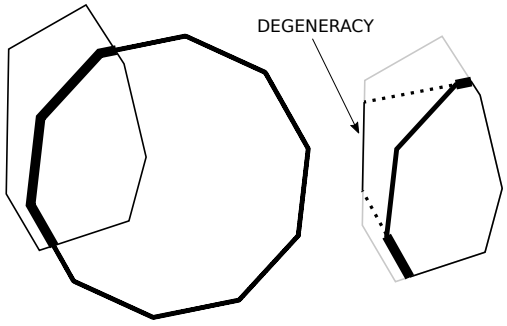


Figure 9. Integrating the model surface in the Voronoi cell (2D view): thick lines define the original model surface, thin lines are the Voronoi cell faces, dashed lines are the split triangle planes

In the figure 9 there is one cell face still valid but apparently outside the model surface. This might happen only if some sites are generated outside the model geometry. Such sites can cause degeneracies (Fig. 10) which can be fixed locally but also avoided by their proper generation (Fig 2). The initial bounding box faces are already marked to avoid their triangulation.

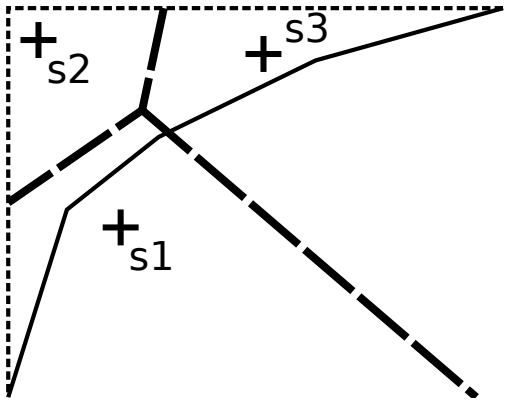


Figure 10. Degeneracies caused by sites outside the model. The solid line represents the model geometry, dashed thin line represents the bounding box and dashed thick line is the boundary between the Voronoi cells. $s1$ is a valid site but the whole cell defined by $s2$ is outside the model so a test would be needed to eliminate this cell. While cell defined by $s3$ is penetrating the surface, the site itself is outside the geometry which would invert the results of the surface triangles' tests due to the cell-space conversion.

It also would be possible to avoid the triangle copying by insertion of all the surface triangles into the Box Cutting algorithm which would inherently integrate the mode surface in the cell. This might be good for simple models but when dealing with a bit higher resolution models, this might get slow because of the need to basically triangulate the surface triangles again and also the cell internal representation might get too big to fit the local memory.

Creating the new object in the global memory representing one fragment of the original model can be summed up in the following steps:

1. Each thread checks one triangle, stores the result in a bitmap
2. If needed the triangle splitting is performed, storing the new triangles in a temporary array
3. The split triangles' planes are inserted in BCA to modify the cell shape
4. The appropriate cell faces are triangulated and counted
5. A part of global memory is reserved in vertex and element buffer (Fig. 11)
6. Triangles are copied in the global memory in parallel

7. A new object info and draw command is created (incrementing draw and object counters)

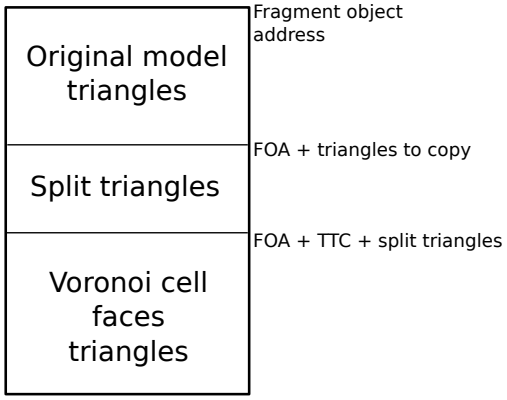


Figure 11. Memory layout in vertex and element buffer for one fragment of the original model

6. Physics

This is just a brief overview of the additional techniques implementing physics which is not the main topic of this paper. Please read the referenced material for more detailed explanation.

The broad-phase of collision detection decides what objects might collide using bounding volume hierarchy. An effective way to build a BVH on GPU is well described in many papers [5]. Here is a possible approach (Fig. 12):

1. Calculate the bounding box center for each object in parallel
2. Transform the center position into Morton code by bitwise interleaving of the xyz coordinates
3. Sort the codes using a parallel radix sort (neighbouring codes represent objects that are also near each other in 3D space)
4. Build binary radix tree, in parallel for N-1 nodes (where N is the number of objects):
   (a) Take the initial code on index *n* (the index corresponds to the node index)
   (b) Define the direction of the node by the neighbouring code with greater common prefix
   (c) Get the length of the node, aka find the code that has smaller common prefix with the initial code than the lower bound (the code excluded during the direction test)
   (d) Find the split point of the node (furthest code from the initial one having greater mutual common prefix than with the end of the sequence)
   (e) The indices of the children are the indices of the two codes at the split point

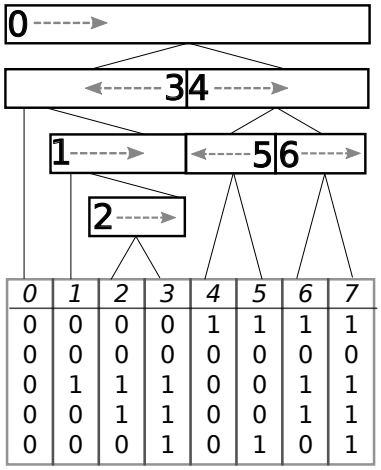


Figure 12. BVH constructed over an ordered list of Morton codes, each node is defined by the beginning code with the same index as the node, sequence direction and split point with children

In the narrow-phase it's possible to use the separating axis theorem [8] in parallel over both objects' normals or over predefined number of axes. The approach is quickly described here:

1. Project the models on each axis in parallel (dot product of each model vertex position and axis vector, gives us a range)
2. Find an axis where the projections don't overlap
   (a) If the axis is found, finish with no collision
   (b) If it's not found, return the minimal translation vector (minimal amount of overlap and corresponding collision normal)

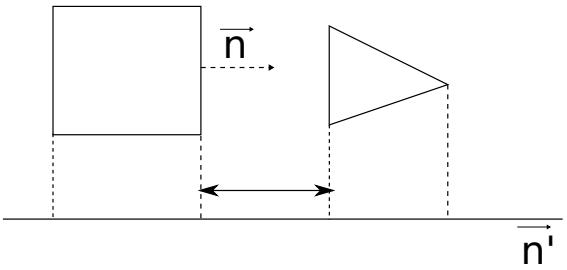


Figure 13. Separating axis theorem - situation where the axis of separation was found and no collision happens

This can be used also for moving objects to predict the collision. The time interval of collision can be calculated on each axis and collision only happens when there is a non empty time interval intersection on all axes. Rotations might however cause errors. When the objects are already overlapping, it's possible to push them away from the collision using the MTV, mentioned above.

As for physics and collision response, the equations derivation would exceed the nature of this paper.

It is possible to use a combination of the basic physics formulas for elastic/inelastic collisions and angular velocity or the all-in-one impulse based approach [9].

7. Conclusions

This paper has introduced a possible set of algorithms that can be used to efficiently implement a brittle body simulation. The main focus was on the parallel way to construct Voronoi diagram and to use the result to partition the model into fragments. During my research I have found only Delaunay based, discrete voxel based algorithms or possibly inaccurate approximations so this might be a nice new alternative for GPU applications or highly parallel environments.

There are many optimizations that can be additionally done including some implementation details that are out of the scope of this paper. I'd love to keep working on this project and present an updated and improved approach. Because of the complexity of this thesis, I am presenting only the early alpha version here with a basic performance measurement (Fig. 14) and performance distribution (Fig. 15).

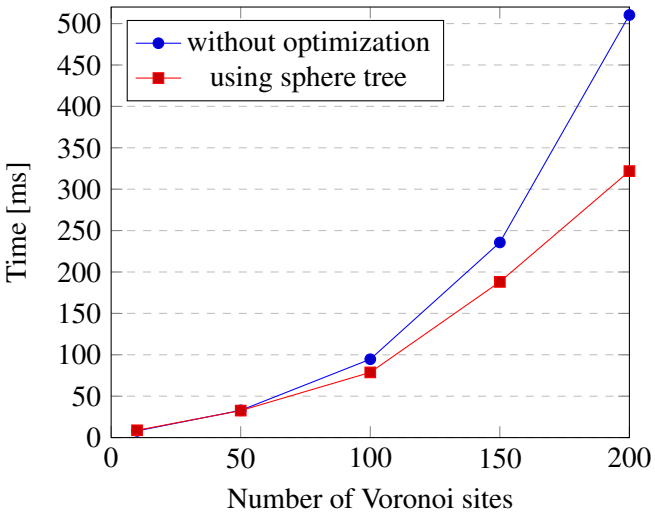


Figure 14. Measuring the performance with a simple 3D model (icosphere with 42 vertices). Please note that this is an early version and many optimizations, which might bring a significant speedup (hopefully), are yet to be done. The sites were generated in uniform distribution. When concentrating more sites around a contact point, the times are increasing and the tree optimization impact is also more significant. Measured using GeForce GTX 550 Ti

Acknowledgements

I would like to thank my supervisor Ing. Tomáš Milet for his help and literally hours of consultations.

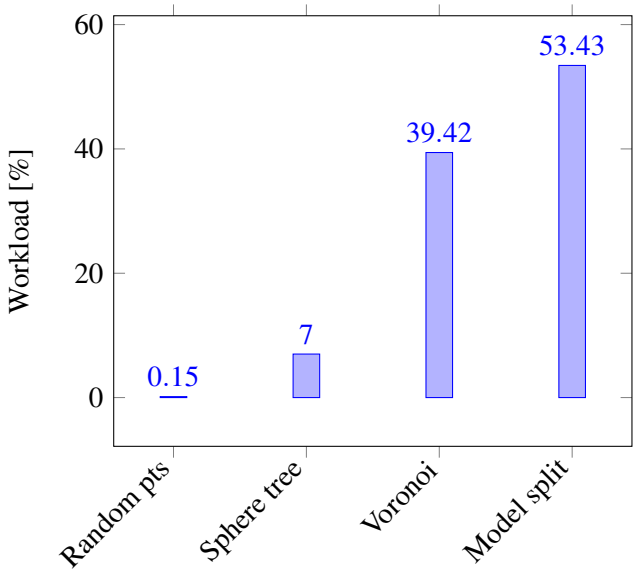


Figure 15. Distribution of the workload between the parts of the whole calculation. The model split part includes also storing of the new objects in the memory.

References

[1] Erwin Coumans. Gpu rigid body simulation. In *Game Developer Conference*, volume 2, 2013.

[2] Jiangfan Ning, Huaxun Xu, Liang Zeng, and Sikun Li. Particle-based fracture simulation on the gpu. In *Transactions on edutainment VI*, pages 193–205. Springer, 2011.

[3] Ashwin Nanjappa. *Delaunay Triangulation in R3 on the GPU*. PhD thesis, 2012.

[4] Cristina N Vasconcelos, Asla Sá, Paulo Cezar Carvalho, and Marcelo Gattass. Lloyd's algorithm on gpu. In *International Symposium on Visual Computing*, pages 953–964. Springer, 2008.

[5] Pedro Lousada, Vasco Costa, and João M Pereira. Bandwidth and memory efficiency in real-time ray tracing. 2017.

[6] Pierre L'Ecuyer and Richard Simard. A software library in ansi c for empirical testing of random number generators. Technical report, Technical report, Département d'Informatique et de Recherche Opérationnelle Université de Montréal, 2002.

[7] Blackpawn. Making cellular textures. Available at `http://blackpawn.com/texts/cellular/default.html`.

[8] David Eberly. Intersection of convex objects: The method of separating axes. *Geometric Tools, LLC http://www. geometrictools. com,(1998- 2008)*, 2001.

[9] Chris Hecker. Physics, part 4: The third dimension.
Game Developer, pages 15–26, 1997.