Fast Mesh Similarity Measuring Based on CUDA

Jie Tang, Gangshan Wu, Bo Xu, Zhongliang Gong
National Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, China
tangjie@nju.edu.cn

Abstract— **This paper presented a fast algorithm which could measure similarity between two meshes interactively. The algorithm was based on CUDA (Compute Unified Device Architecture) technology. In order to fully utilize the computing power of GPU, we developed parallel method to construct uniform grid for fast space indexing of triangles. Special data structure was designed on device end to overcome the disadvantage of CUDA that it does not support dynamic allocation of memory. Lots of experiments were carried out and the results verified the effectiveness and efficiency of our algorithm.**

Keywords-Hausdorff distance; CUDA; Mesh

I. INTRODUCTION

Measuring the similarity between different polygonal objects is often needed as a first step for other geometric computations in diverse fields including computer graphics, computer games, virtual environment, geometric modeling, and robotics. Thus the development of efficient algorithms for distance computation is very important for improving the performance of related geometric problems. Various types of similarity measures have been extensively investigated and efficient algorithms have been proposed over the past two decades. Among them, the Hausdorff distance has attracted considerable research attention.

Using Hausdorff, however, distance to measure the similarity between meshes has two disadvantages. Firstly, the efficiency of the algorithm is quite low. For polygonal models in R^3 with n polygons, the expected time taken by the best-known algorithm to exactly evaluate the Hausdorff distance is $O(n^3+\varepsilon)$, where $\varepsilon > 0$ [2]. Thus, due to the high computational complexity and difficult implementation of proposed approaches, very few algorithms exist to compute Hausdorff distance for polygonal models in R^3. Secondly, The Hausdorff distance between two models is the maximum deviation between them. Hence it cannot reflect the overall shape similarity and is easy to be affected by noise. Using the summation of the squared Hausdorff distance of all sample points is a resonable solution. But the efficiency is still a bottleneck.

The recent introduction of the CUDA programming model, along with the advancement in GPU hardware design, made GPUs an attractive architecture for implementing parallel algorithms such as ray tracing [7][11][13][15]. Current NVIDIA GPUs are many-core processor chips, scaling from 8 to 240 cores. Increasing parallelism, rather than increasing clock rate, has become the primary engine of processor performance growth. This degree of hardware parallelism reflects the fact that GPU architectures evolved to fit the needs of those problems with tremendous inherent parallelism. However, this also raised many important questions about how to develop efficient parallel programs. Exist methods cannot be shifted to many-core system directly.

In this paper, we present a parallel method to compute the similarity between meshes. The whole process is divided into three steps: uniform grid construction, sampling on meshes and Hausdorff distance computing. Each step is converted into a kernel function which could be executed paralleled on GPU. The main contributions of our work are as follows:

- Present a better way to measure the similarity between meshes.
- Implemented a parallel method to construct a uniform grid containing triangles for fast space indexing of a triangle.
- Designed efficient data structure on device end, overcame the shortcoming of CUDA which does not support pointers and dynamic allocation of memory.

We have implemented the algorithm on graphics card using CUDA which gained a significant speedup over the CPU implementation and other exist methods. Actually, we could measure the similarity between two meshes with more than 100, 000 triangles in real time.

II. RELATED WORK

Since the seminal work by Atallah [4], different algorithms for mesh similarity measuring have been proposed in the literature, we give only a short overview of the most related methods.

For simple, non-convex polygons with n and m vertices, Alt [1] presented a method to compute the Hausdorff distance between two point sets in R^2 based on the Voronoi diagram which requires $O((n+m)log(n+m))$ running time. For R^3, Alt [2] proposed a randomized algorithm with $O((n+m+(nm)3/4) log(n+m))$ expected time.

Klein [8] first used the Hausdorff distance between the original and simplified mesh to control the simplification error, although with significant computational effort. Cignoni [5] introduced a method dedicated exclusively to measurement of errors on simplified surfaces. Another method, presented by Aspert [3], is more efficient in terms of speed at the cost of higher memory use. Llanas [12] proposed an algorithm using

978-1-4244-6789-1/10/$26.00 ©2010 IEEE

Work on this paper was partially supported by the Natural Science Foundation of Jiangsu Province (No. BK2008262) and the National High-Tech Research and Development Plan of China under Grant No. 2007AA06A402

random covering, and also demonstrated implementation results for simple, convex ellipsoids. Güthe [6] use polygon subdivision to approximate the solution within an error bound. In a recent work, Tang [14] presented a real-time algorithm for approximating the Hausdorff distance between two triangular meshes within a given error bound. Due to the complexity of the exact computation, the performance of these algorithms is still too slow for interactive applications especially for large scale models.

III. Preliminaries and Overview

In this section, we present some preliminary concepts and theorems related to our mesh similarity measuring algorithm before presenting the algorithm itself.

A. Problem Formulation

The mesh similarity measuring problem could be formulized as: Given two meshes M and M', find a map E: $M \times M' \rightarrow R$, which could evaluate the similarity between two meshes quantitively. The lower the $E(M, M')$ is, the more similar the two meshes are.

Hausdorff distance was adopted by some reseachers to measure the similarity between meshes. The definition of Harsdorff distance between two meshes is as follows. Firstly, the distance from a point $\boldsymbol{p}$ to a mesh M is defined as:

$$d(p,M') = \min_{p' \in M'} d(p,p') \tag{1}$$

where $d(p,p')$ is the Euclidian distance between two points in R^3.

The one-sided Hausdorff distance between two meshes M and M' is then defined as:

$$d(M,M') = \max_{p \in M} d(p,M') \tag{2}$$

The symmetrical Hausdorff distance is defined as:

$$d_s(M,M') = \max(d(M,M'), d(M',M)) \tag{3}$$

Hausdorff distance is the max difference between two meshes. It, however, does not reflect the overall similarity between two meshes and is easy to be affected by noise. Hence, we proposed an improved metric as follows:

$$E(M,M') = \sum_{p \in M} d^2(p,M') \tag{4}$$

where $d(p, M')$ is defined in (1). And the symmetrical metric is defined as:

$$E_s(M,M') = \max(E(M,M'), E(M',M)) \tag{5}$$

Computing E_s is a time consuming task. Fortunately, with the adventure of GPU and CUDA, we could compute it paralleledly even in real time.

B. Overview of the Algorithm

Equation (4) could be computed paralleledly on GPU. Each sample point uses a single thread. Equation (1) is time consuming. So we create an uniform grid to accelerate the indexing of a triangle in R^3. Below is algorithm to compute (2) and (4). Here we mean device end as GPU and host end as CPU.

```
Algorithm 1: One-side Mesh Similarity Measuring
Input: $M$, $M'$
Output: mesh similarity
1: import $M$ and $M'$
2: compute the number of vertex and triangle of $M'$
3: compute the size of uniform grids
4: transfer $M'$ to device memory
5: compute the memory size $G$ of uniform grids of $M'$
6: transfer $G$ to host end
7: allocate the memory for the uniform grids of $M'$
8: construct the uniform grids of $M'$
9: sampling on $M$
10: compute the $d(p, M')$ for each sample point of $M$
11: compute $d(M, M')$ and $E(M, M')$
12: transfer the result to host end
```

Step 1-4, 7, 9 are carried out on CPU side and other steps are carried out on GPU side.

IV. Uniform Grids

In this section, we present some preliminary concepts and theorems related to our mesh similarity measuring algorithm before presenting the algorithm itself.

Uniform grids are used in a wide variety of applications including ray tracing of dynamic scenes. While other spatial structures, such as kd-trees and BVHs can provide better performance, their construction is very time consuming. GPU-algorithms that allow per-frame rebuild of hierarchical data structures in real time were introduced only recently Zhou [15]; Lauterbach [11] and build times are still considerably slower than those of grids.

During constructing uniform grids of triangles on CUDA device, we met some problems:

- Each triangle could belong to multiple grids, and the number of grids is unfixed.
- CUDA does not support dynamic allocation of memory.
- CUDA does not support pointers.

Lagae [10] proposed a parallel method to construct uniform grids which runs pretty fast on CPU. Its data structure, however, does fit into GPU device. We designed a proper data structure and proposed a parallel method to construct uniform grids on GPU which is similar to Kalojanov [9] 's work.

A. Data Structure

The data structure used during the constructing of uniform grids of triangle mesh has 4 components: (1) Vertex array (vertexs): the coordinates of vertices. (2) Triangle array (triangles): the indices of each triangle of the mesh. (3) Pair array (pairs): each element in this array contains two symbols. The first is grid index, and the second is the index of the triangle distributed to this grid. A grid could have multiple triangles. (4) Grid array (cells): this array store the start index and end index of a grid in pair array. If a grid has no triangle, then store -1 as its start and end index.

Fig. 1 gives an example of our data structure of uniform grids. In Fig. 1 we could find that grid 2 has 4 triangles which are triangle 2, 3, 6 and 8. As for grid 2 in grid array (cells), we record its start index in pair array (pairs) which is 7 and its end index which is 10.

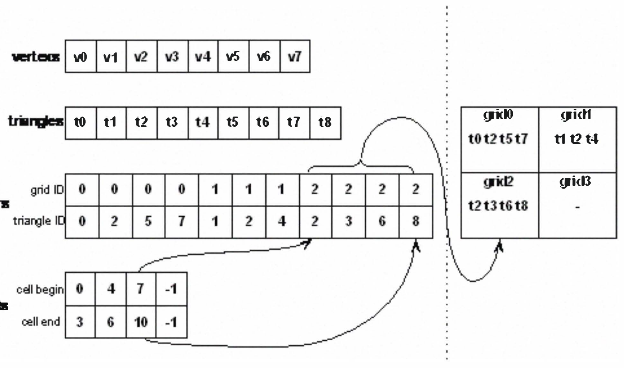

Figure 1. an example of uniform grid on CUDA

B. *Algorithm*

Among the four parts of the data structure mentioned above, the lengths of vertex array, triangle array and grid array could be precomputed when we have the mesh. But the size of pair array could not. This is because a triangle could overlap multiple grids. In CPU, we can use pointer and dynamic allocation of memory to solve the problem. But we cannot do it on CUDA. Therefore we have to know the size of the array that stores the primitive references in advance.

We present a CUDA kernel function to compute the size of pair array. Each thread computed how many grids a triangle intersect and write the number into a shared memory variable. After all thread finished, all numbers are summed and the result is the length of pair array. The kernel function is shown below, in which THREAD_NUM must be 2^n, *block_num* is (*tnum*-1)/THREAD_NUM+1. *tnum* is the number of triangles.

Algorithm 2: Computing overlap triangle numbers

```
1: __global__ ComputeOverlap(int vnum, int tnum, float3
     *vbuf, int3 *tbuf, int *ref){
2:    __shared__ int ret[THREAD_NUM];
3:    int t_idx←blockIdx.x*blockDim.x+threadIdx.x;
4:    if(t_idx>=tnum)
          ret[t_idx] = 0;
5:    else {
6:        get the vertices of the triangle
7:        compute the normal and the plane p of the triangle
8:        find the possible intersected grids S
9:        int n←0;
10:       for each grid s in S do{
11:           if(8 vertices of s lies in diferent side of p)
12:               n++; //triangle overlap this grid
13:       }
14:       ret[t_idx] = n;
15:   }
16:   __syncthreads(); //make sure all thread finished
17:   sum all ret;
18:   __syncthreads();
19:   if(threadIdx.x==0) ref[blockIdx.x] =ret[0];
20:}
```

After all threads finished, we sum all values in ref array, and return it to host end as the total pair array length. When we allocate the device memory for pair array, we run a second kernel to scan the triangle array once again. Each thread loads a triangle, computes again how many grids it overlaps and for each overlapped grid writes a pair consisting of the grid and triangle indices. We can use the shared memory to write the pair counts and perform a prefix sum to determine output locations in pair array. After being written, the pairs are sorted by the cell index via radix sort. We used the radix sort implementation from the CUDA SDK examples for this step of the algorithm.

After pair array is calculated, the next step is to deduce the grid array (cells in Fig. 1). We also designed a parallel method to fulfill this task. Each thread checks if two neighboring pairs have different cell indexes. If such exists, the corresponding thread updates the range indexes in both grids. During the execution, each thread loads the element to share memory. Algorithm 3 shows the detail.

Algorithm 3: Computing the grid array

```
1: __global__ ExtractCell( int pairnum, uint2 * pairs,
2:                         int2 *cells){
3:    __shared__ uint pairs_cell[THREAD_NUM];
4:    int pairs_idx = blockIdx.x*blockDim.x+threadIdx.x;
5:    pairs_cell[pairs_idx]=
6:              pairs_idx<pairnum?pairs[pairs_idx].x:0;
7:    __syncthreads();
8:    if(pairs_idx==0){
9:      set the start index of the first elememnt;
10:     return;
11:   }
12:   if(thread_idx.x==0){//first thread of the block
13:     load pre pair p1 from global memory;
14:     load current pair p2 from share memory;
15:     if (the grid index of p1 and p2 are different){
16:       set the end index of the pre grid;
17:       set the start index of the current grid;
18:       return;
19:     }
20:   }
21:   if(pairs_idx<pairnum)
22:     load pre pair p1 from share memory;
23:     load current pair p2 from share memory;
24:     if (the grid index of p1 and p2 are different){
25:       set the end index of the pre grid;
26:       set the start index of the current grid;
27:       return;
28:     }
29:   }
30:   if(pairs_idx==pairnum-1){
31:     set the end index of the grid;
32:     return;
33:   }
34:}
```

After the grid array was constructed, the uniform grid creating process was finished. Using this structure, we could fast index a triangle in R^3 space. And the method has a linear complexity. Another advantage is that the algorithm avoided writing conflict and no atomic synchronization is required throughout the construction. Hence, the performance of the construction algorithm depends only on the number of triangles that are inserted into the grid.

V. Similarity Computing

Using metric described in (5) could better reflect the similarity between two meshes. But it needs massive computing. With the help of uniform grid structure and the great computing ability of GPU, we could achieve interactive results even processing large scale models. The atomic task is to compute the distance from a sample point to another mesh. Here, we proposed a parallel method to compute it. The algorithm of kernel function is showed below.

Algorithm 4 Computing the distance from a sample point to a mesh

```
1: __global__ Hausdorff_kernel(){
2:   int vidx ←blockIdx.x*blockDim.x+threadIdx.x;
3:   if(vidx>= sample_point_num) return;
4:   gridIdx ←grid index in which point vidx locates;
5:   set the current grid as the search area
6:   float hDis ←FLOAT_MIN
7:   int flag ←0;
8:   do{
9:      flag ←0;
10:     hDis ←the least distance from point vidx to all triangles of
11:              search area
12:     if(one of the distance from point vidx to 6 bounding planes of
13:                       search area < hDis){
12:        extend the search area in that direction by one grid
13:        flag ←1;
14:     }
15:  }while(flag)
16:}
```

The atomic operation of algorithm 4 is to compute the distance from a point $\boldsymbol{v}$ and a triangle $\boldsymbol{v_0v_1v_2}$, the point $\boldsymbol{v}$' is the projection of $\boldsymbol{v}$ on to the plane where the triangle lies (shown in Fig. 2). The distance d from $\boldsymbol{v}$ to triangle $\boldsymbol{v_0v_1v_2}$ could have 3 different cases according to the location of the $\boldsymbol{v}$':

- when $\boldsymbol{v}$' lies in area 1, d is the distance from point $\boldsymbol{v}$ to the plane on which triangle $\boldsymbol{v_0v_1v_2}$ lies.
- when $\boldsymbol{v}$' lies in area 2, d is the distance from point $\boldsymbol{v}$ to the relative edge.
- when $\boldsymbol{v}$' lies in area 3, d is the distance from point $\boldsymbol{v}$ to the relative vertex.

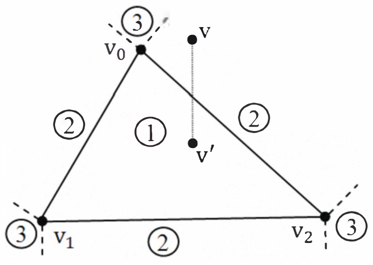

Figure 2. The projection of v on the triangle v0v1v2 (the dashline is perpendicular to the relative edge)

VI. Results

We have implemented the algorithm on CUDA 2.3. The testing systems are:

system 1: Windows XP, Inter Core 2 Duo E6550@2.33GHz(2 CPUs), 2GB RAM, NVIDIA GeForce 9800GT which has 112 CUDA core and 512MB Memory.

We have test our method using many models including Bunny, Armadillo and Buddha from Stanford 3D library as well as some models we build. The models are list in table 1. Table 2 shows the time results on system 1.

TABLE I. The models used for mesh similarity measuring

model	***Bunny***	***Bunny_a***	***Bunny_b***
Vert num	35947	739	13030
Tri num	69451	1421	26069
model	***Bunny_a***	***Armad***	***Armad_a***
Vert num	220	171550	57892
Tri num	441	343096	115780
model	***Armad_b***	***Cube***	***Ball***
Vert num	108141	21633	20897
Tri num	216278	43247	41790

TABLE II. The time results of mesh similarity measuring(ms).

model A	***Bunny_a***	***Armad_a***	***Armad_a***
model B	***Bunny***	***Armad_b***	***Armad***
sampling time	0.717	1.31	1.65
time 1	4.08	13.32	13.32
time 2	9.64	21.41	34.20
time 3	37.9	305.9	399.2
time 4	19.5	143.1	200.2
time 5	7.2	23.8	28.8
rendering time	1.05	1.05	2.18
Total time	80	510	680

In table 2, time 1 is the time to create the uniform grid for model A, time 2 is the time to create the uniform gird for model B, time 3 is the time to compute the shortest distance from a point of A to the model B, time 4 is the time to compute the shortest distance from a point of B to model A and time 5 is the time to compute the similarity according to (5). From the table, we could find that the most time consuming task is to compute the shortest distance from a point of A to model B. This process involves lots of distances from a point to a triangle. And since the coordinates of a vertex and the vertex indices of a triangle are stored on global memory of GPU, visit them usually is slow.

Fig. 3 shows the comparison between our method and those of Güthe's [6] and Tang's [14] methods when computing the Hausdorff distance between Bunny_a and Bunny. From the figure, we could find that our method run the fastest. Also what we get is more than Hausdorff distance and could better reflect the similarity of two meshes.

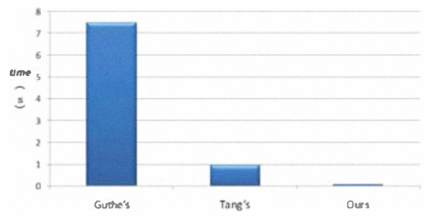

Figure 3. The comparison of running time between our method and Guthe's and Tang's methods

In order to verify the effectiveness of our method, we change the computed distance into colorful output. Fig. 4 shows the result of a similarity between a box and a sphere. The barycenters of both models are coincident. The edge length of the box is 17, and the radius of the sphere is 10. In Fig. 4, the blue color means the distance is 0, and the red color means the distance is much larger.

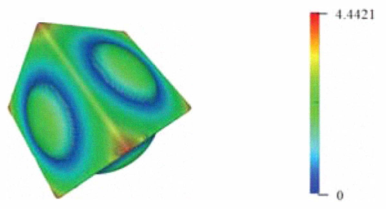

Figure 4. The colorful result of similarity between a box and a sphere

Fig. 5 shows the result of Bunny model. The top left one is the original model. The top middle is Bunny_b. The top right is the result of Bunny_b to Bunny. The bottom middle model is Bunny_c. The bottom right is the result of Bunny_c to Bunny.

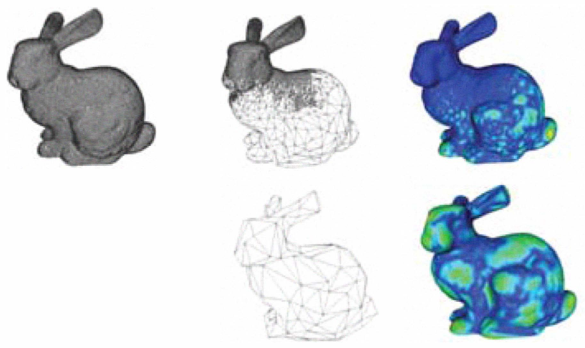

Figure 5. The colorful output of Bunny model

VII. RESULTS

We presented a fast algorithm to measure the similarity of two meshes based on CUDA technology. Instead of using Hausdorff distance, we proposed a new metric to measure the similarity of two meshes which could better reflect the overall difference. In order to fully utilize the computing power of GPU, we developed parallel solutions to construct uniform grid for fast space indexing of triangles. Special data structure was designed on device end to overcome the disadvantage of CUDA that it does not support pointer and dynamic allocation of memory. Our method could compute the Hausdorff distance interactively even dealing with large scale models. The results verified the effectiveness and efficiency of our algorithm.

The possible future work including mesh shape matching using CUDA, similarity measuring between meshes with multiple properties.

REFERENCES

[1] H. Alt, B. Behrends, J. Blömer, Approximate matching of polygonal shapes. Ann. Math. Artif. Intell. 13 (1995), 251–266.

[2] H. Alt, P. Brass, M. Godau, C. Knauer, C. Wenk, Computing the hausdorff distance of geometric patterns and shapes. Discrete and Computational Geometry 25 (2003), 65–76.

[3] N. Aspert, D. Santa-Cruz, Mesh: measuring errors between surfaces using the hausdorff distance. In Proc. of the IEEE International Conference on Multimedia and Expo (2002), pp. 705–708.

[4] M. J. Atallah, A linear time algorithm for the hausdorff distance between convex polygons. Inf. Process. Lett. 17 (1983), 207–209.

[5] Cignoni P., Rocchini C., Scopigno R., Metro: measuring error on simplified surfaces. Computer Graphics Forum 17, 2 (1998), 167–174.

[6] Güthe M., Borodin P., Klein R., Fast and accurate hausdorff distance calculation between meshes. Journal of WSCG 13, 2 (Feb. 2005), 41–48.

[7] Günther J., Popov S., Seidel H.-P., Slusallek P.: Realtime ray tracing on GPU with BVH-based packet traversal. In Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007 (sep 2007), pp. 113–118.

[8] Klein R., Liebich G., Strasserw., Mesh reduction with error control. In Proc. of IEEE Visualization 96 (1996), pp. 311–318.

[9] Kalojanov J., Slusallek P., A parallel algorithm for construction of uniform grids. In Proceedings of the Conference on High Performance Graphics (2009), pp. 23–28.

[10] Lagae A., Dutré P., Compact,fast and robust grids for ray tracing. In Proceedings of the 19th Eurographics Symposium on Rendering (Jun 2008), pp. 1235–1244.

[11] Lauterbach C., Garland M., Sengupta S., Luebke D., Manocha D., Fast bvh construction on gpus. Computer Graphics Forum 28, 2 (2009), 375–384.

[12] Llanas B., Efficient computation of the hausdorff distance between polytopes by exterior random covering. Comput. Optim. Appl. 30, 2 (2005), 161–194.

[13] Popov S., Gunther J., Seidel H.-P., Slusallek P., Stackless kd-tree traversal for high performance gpu ray tracing. Computer Graphics Forum 26, 3 (Sept. 2007).

[14] Tang M., Lee M., Kim Y. J. Interactive hausdorff distance computation for general polygonal models. ACM Trans. Graph. 28, 3 (2009), 1–9.

[15] Zhou K., Hou Q., Wang R., Guo B., Real-time kd-tree construction on graphics hardware. ACM Trans. Graph. 27, 5 (2008), 1–11.