GLT (GPU Local Triangulation) - an interpolating surface reconstruction algorithm

1. We present a parallel surface reconstruction algorithm based on local triangulations and its implementation in the GPU. Surface reconstruction is a well known problem in geometry and many works have been done in this field. Surface reconstruction looks for the computation of a surface (usually a discrete representation such as a triangle mesh) given a set of points sampled from an original surface. A list of some of the most relevant works in the area is displayed on the slide. The way surface and points are related defines two kinds of reconstruction: interpolating and approximating; in the first case, the surface uses the input points as triangle vertices. The second generates a surface that is near to the points, but not necessarily adjusts tightly to them.

2. Approximating methods are more robust against noise than interpolating, but this usually leads to details loss. Although the most of recent works show very good results, it comes at the expense of very low computational times. Some other works show the power of modern graphic hardware to accelerate approximating methods.

3. Interpolating techniques guarantee that all the input points are included in the final surface, which may be expected in situations such as simulation results, preprocessed point clouds or implicit functions rendering. Usually these techniques are based in Delaunay triangulations… (and our is not the exception). On the other hand, the most common Delaunay triangulation algorithms are incremental or recursive, leading to high processing times too. Gopi et al. proposed an interesting method based on local Delaunay triangulations.

4. It is known that if for a set of local triangulations, each local fan is a whole Delaunay triangulation, then the global triangulation can be obtained merging all these fans. Further more (and here is where our work starts) local triangulations become independent to each other, so parallelization is possible. To achieve this parallelization, we have wanted to take advantage of the high processing power of GPUs without restricting to the most recent CUDA API. It is important to note that although there are some approximating methods implemented or specifically designed for GPU, there are no works on interpolating techniques. The use of shaders implies the development or adaptation of several algorithms and data structures to fit this programming scheme.

5. But, lets first see the overall process for the local Delaunay triangulation:
   a. The *k*-Nearest Neighbors for each point are computed. This parameter *k* has a strong influence in the final reconstruction, specially when dealing with non-uniformly sampled surfaces; *k* values from 16 to 32 are usually good enough.
   b. After the *k*-NN, for each point:
      i. The normal is estimated.
      ii. Every neighbor is sorted by angle around the central point and normal.
      iii. Neighbors are projected onto the tangent plane of the central point.

iv. Neighbors are tested to verify if they belong to the Delaunay neighborhood of the central points. Invalid points are discarded iteratively until no more invalid points left.

v. The local fan is created using the sorted valid neighbors.

c. Finally, all local fans are joined and duplicated triangles are eliminated.

6. It can be seen that the second phase can be executed in parallel… and that is what we have done implementing it in the GPU; phases one and three are executed in the CPU.

7. In order to fit GPU's data structures and graphic memory constraints, the input points are divided in smaller data sets and stored in textures. As for each point a set of maximum *k* neighbors is attached, there is another texture, the neighborhood texture, which stores it. As it is shown in the slide, the texture stores fixed size neighborhoods in each row. This scheme also allows the algorithm to simplify neighbors references, using only a single index as with one-dimensional arrays. Neighborhoods with less than *k* points are filled with virtual points, so we always have a full set, but this is an unusual case.

8. To approximate the normal of the neighborhood, we employ the same method that the proposed by Hoppe et al., which uses Principal Component Analysis, but, at opposite as Hoppe, we estimate the tangent plane, rather than the normal directly, to avoid numerical errors, more probable in the eigenvector related to the smaller eigenvalue. For our GPU implementation we use the Power Method for eigenvectors computation. This method provides robust results in few iterations. Gopi's method performs the local Delaunay triangulation in a two-dimensional space, the tangent plane: neighbors are projected onto the tangent plane, preserving its distance to the central point. As explained before, the triangulation method creates an initial fan with all the neighbors of a point. Then, iteratively it removes the non-Delunay neighbors. This fan is created by simply linking angle-consecutive neighbors, starting with the nearest point (which is always a valid Delaunay neighbor). Angle computation is performed in the same pass that projection, since both are per-neighbor outputs. Once angles are computed, they must be sorted to create the fan. We employ the GPU Bitonic Merge Sorting algorithm because its good performance with small lists, such as the neighborhoods of our points. To avoid making as many passes as points are, we have made an small change: the original GPU algorithm performed the sorting of a single array; taking advantage of the per-row storing of each neighborhood and its fixed size, we can transform the algorithm to multi-list sorting, simply using the index of the neighbor as the offset in the U coordinate, instead of the U coordinate itself.

9. The Delaunay validation is displayed on the slide. Given three consecutive projected neighbors A, B and C, lets call B' the middle point of the segment Bp. If the intersection of the perpendicular bisectors of Ap and Cp intersect the segment B'p, then the point B is not a Delaunay neighbor of *p* (please refer to the paper for a full proof of this method). Further more, this validation has the property to be completely independent to the each other, so they can be done in parallel too.

10. One of the major disadvantages of Gopi's method is the use of a recursive procedure for Delaunay neighbors verification. As it is known, GPUs do not support recursion, and the iterative equivalent algorithm doesn't take advantage of the multiple processors they have. To solve this issue, our method makes use of a ring structure to store the neighborhood, and tests each point against its predecessor and successor, invalidating it or not. When no more points are invalidated, the local triangulation is finished. A 1024^2 texture can store 64K neighborhood sets, so for bigger point clouds, several passes are necessary. Again, these passes are independent to each other, allowing multi-GPU or grid computing.
11.
12.
13.
14. After all passes are finished, results are merged into a single mesh and duplicated triangles are removed. In a final step, normal sign is unified; a random point is taken as the reference orientation and then it is propagated through the mesh connectivity.

15. On the slide can be seen the time for tests. These tests were performed using an nVidia 7950 with 1GB of VRAM. The 'passes' column refers to the number of sets needed to reconstruct the model. The Dragon set, for example, needed a bigger neighborhood (32 points) to reconstruct correctly, it leads of course to a higher number of passes. All the other models were reconstructed with $k = 16$.

16. We have compared our results to a CPU implementation of the original method, running in the same machine (Intel Core2Duo 1.83 with 1GB of RAM). We can appreciate a speed gaining of up to a grade of magnitude. On the following slides we'd like to show some of the results of our work.

17. Here's the Hand model. This model presents topology complexity at the wrist, due to the small distance between bones. The point cloud is also non-uniform. Our method reconstruct correctly the model.

18. This is a preprocessed version of the Dragon model, where noise has been removed. Even though, the model presents non-uniform zones and regions with frequency changes. This model needed a bigger neighborhood to reconstruct properly.

19. As a final example, the Blade is a more challenging model with almost 900K points and different topology challenges such as the holes at the top of the Blade. Currently we are working on some enhancements to the method, such as noise reduction to the input points, parameters estimation for the k-NN, and topology analysis.

Thanks