Highly Parallel Surface Reconstruction

Kun Zhou Minmin Gong Xin Huang Baining Guo
Microsoft Research Asia

Abstract

We present a parallel surface reconstruction algorithm that runs entirely on the GPU. Like existing implicit surface reconstruction methods, our algorithm first builds an octree for the given set of oriented points, then computes an implicit function over the space of the octree, and finally extracts an isosurface as a water-tight triangle mesh. A key component of our algorithm is a novel technique for octree construction on the GPU. This technique builds octrees in real-time and uses level-order traversals to exploit the fine-grained parallelism of the GPU. Moreover, the technique produces octrees that provide fast access to the neighborhood information of each octree node, which is critical for fast GPU surface reconstruction. With an octree so constructed, our GPU algorithm performs Poisson surface reconstruction, which produces high quality surfaces through a global optimization. Given a set of 500K points, our algorithm runs at the rate of about five frames per second, which is over two orders of magnitude faster than previous CPU algorithms. To demonstrate the potential of our algorithm, we propose a user-guided surface reconstruction technique which reduces the topological ambiguity and improves reconstruction results for imperfect scan data. We also show how to use our algorithm to perform on-the-fly conversion from dynamic point clouds to surfaces.

Keywords: surface reconstruction, octree, marching cubes, free-form deformation, boolean operation

1 Introduction

Surface reconstruction from point clouds has been an active research area in computer graphics. This reconstruction approach is widely used for fitting 3D scanned data, filling holes on surfaces, and remeshing existing surfaces. So far, surface reconstruction has been regarded as an off-line process. Although there exist a number of algorithms capable of producing high-quality surfaces, none of these can achieve interactive performance.

In this paper we present a parallel surface reconstruction algorithm that runs entirely on the GPU. Following previous implicit surface reconstruction methods, our algorithm first builds an octree for the given set of oriented points, then computes an implicit function over the space of the octree, and finally extracts an isosurface as a water-tight triangle mesh using the marching cubes. Unlike previous methods which all run on CPUs, our algorithm performs all computation on the GPU and capitalizes on modern GPUs' massive parallel architecture. Given a set of 500K points, our algorithm runs at the rate of about five frames per second. This is over two orders of magnitude faster than previous CPU algorithms.

The basis of our algorithm is a novel technique for fast octree construction of the GPU. This technique has two important features. First, it builds octrees in real-time by exploiting the fine-grained parallelism on the GPU. Unlike conventional CPU octree builders, which often construct trees by depth-first traversals, our technique is based on level-order traversals: all octree nodes at the same tree level are processed in parallel, one level at a time. Modern GPU architecture contains multiple physical multi-processors and requires tens of thousands of threads to make the best use of these processors [NVIDIA 2007]. With level-order traversals, our technique maximizes the parallelism by spawning a new thread for every node at the same tree level.

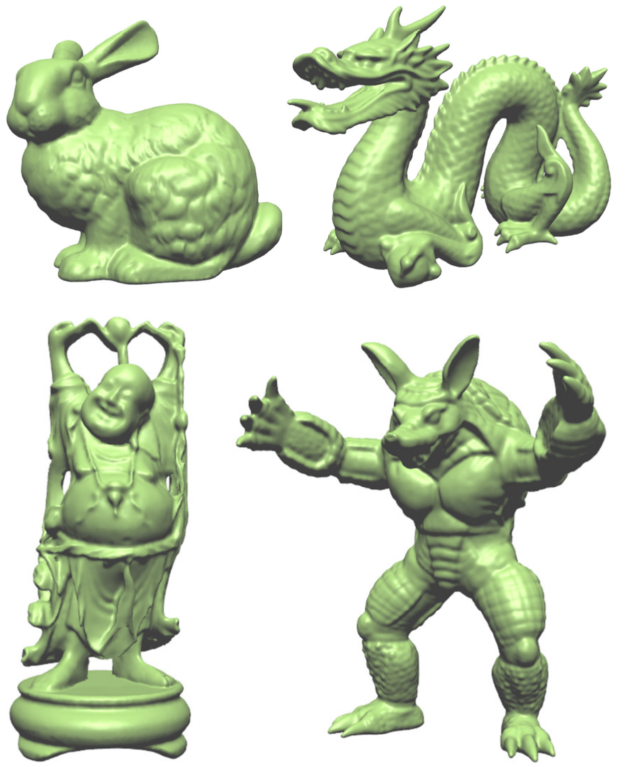


Figure 1: *Our GPU reconstruction algorithm can generate high quality surfaces with fine details from noisy real-world scans. The algorithm runs at interactive frame rates. Top left: Bunny, 350K points, 5.2 fps. Top right: Dragon, 1500K points, 1.3 fps. Bottom left: Buddha, 640K points, 4 fps. Bottom right: Armadillo, 500K points, 5 fps.*

The second feature of our technique is that it constructs octrees that supply the information necessary for GPU surface reconstruction. In particular, it is critical for the octree data structure to provide fast access to tree nodes as well as the neighborhood information of each node (i.e., links to all neighbors of the node). While information of individual nodes is relatively easy to collect, computing the neighborhood information requires a large number of searches for every single node. Collecting neighborhood information for all nodes of the tree is thus extremely expensive even on the GPU. To address this problem, we make the observation that a node's neighbors are determined by the relative position of the node with respect to its parent and its parent's neighbors. Based on this observation, we build two look up tables (LUT) which record the indirect pointers to a node's relatives. Unlike direct pointers, indirect pointers are independent of specific instances of octrees and hence can be precomputed. At runtime, the actual pointers are quickly generated by querying the LUTs.

Based on octrees built as above, we develop a GPU algorithm for the Poisson surface reconstruction method [Kazhdan et al. 2006]. We choose the Poisson method because it can reconstruct high quality surfaces through a global optimization. As part of our GPU algorithm, we derive an efficient procedure for evaluating the divergence vector in the Poisson equation and an adaptive marching cubes procedure for extracting isosurfaces from an implicit function defined over the volume spanned by an octree. Both of these procedures are designed to fully exploit modern GPUs' fine-grained parallel architecture and make heavy use of the octree neighborhood information. Note that GPU algorithms can also be readily designed for classical implicit reconstruction methods such as [Hoppe et al. 1992] by using our octree construction technique and the adaptive marching cubes procedure for extracting isosurfaces on the GPU.

Our GPU surface reconstruction can be employed immediately in existing applications. As an example, we propose a user-guided reconstruction algorithm for imperfect scan data where many areas of the surface are either under-sampled or completely missing. Similar to a recent technique [Sharf et al. 2007], our algorithm allows the user to draw strokes around poorly-sampled areas to reduce topological ambiguities. Benefiting from the high performance of GPU reconstruction, the user can view the reconstructed mesh immediately after drawing a stroke, while [Sharf et al. 2007] requires several minutes to update the reconstructed mesh.

GPU surface reconstruction also opens up new possibilities. As an example, we propose an algorithm for generating surfaces for dynamic point clouds on the fly. The reconstructed meshes may be directly rendered by the traditional polygon-based display pipeline. We demonstrate the application of our algorithm in two well-known modeling operations, free-form deformation and boolean operations. Since our algorithm is linearly scalable to the GPU's computational resources, real-time surface reconstruction will be realized in the near future with advancements in commodity graphics hardware. In view of this, our technique may be regarded as a bridging connection between point- and polygon-based representations.

2 Related Work

Surface reconstruction from point clouds has a long history. Here we only cover references most relevant to our work.

Early reconstruction techniques are based on Delaunay triangulations or Voronoi diagrams [Boissonnat 1984; Amenta et al. 1998] and they build surfaces by connecting the given points. These techniques assume the data is noise-free and densely sampled. For noisy data, postprocessing is often required to generate a smooth surface [Bajaj et al. 1995; Kolluri et al. 2004]. Most other algorithms reconstruct an approximating surface represented in implicit forms, including signed distance functions [Hoppe et al. 1992; Curless and Levoy 1996; Hornung and Kobbelt 2006; Sharf et al. 2007], radial basis functions [Carr et al. 2001; Turk and O'Brien 2002; Ohtake et al. 2004], moving least square surfaces [Alexa et al. 2001; Amenta and Kil 2004; Lipman et al. 2007], and indicator functions [Kazhdan et al. 2006; Alliez et al. 2007]. These algorithms mainly focus on generating high quality meshes to optimally approximate or interpolate the data points.

Existing fast surface reconstruction methods are limited to simple smooth surfaces or height fields. [Randrianarivony and Brunnett 2002] proposed a parallel algorithm to approximate a point set with NURBS surfaces. [Borghese et al. 2002] presented a real-time reconstruction algorithm for height fields. For CAD applications, [Weinert et al. 2002] used a parallel multi-population algorithm to find a CSG representation that best fits data points. None of these techniques is appropriate for reconstructing complex surfaces from point clouds.

Recently, Buchart et al. [2007] proposed a GPU interpolating reconstruction method by using local Delaunay triangulation [Gopi et al. 2000]. First, the k-nearest neighbors to each point are computed on the CPU. Then, for each point on the GPU, its neighbors are ordered by angles around the point and the local Delaunay triangulation is computed. The authors reported $6 \sim 18$ times speed ups over the CPU implementation. For a moderate-sized data set (e.g., 250K points), their algorithm needs over 10 seconds, which is still far from interactive performance. Moreover, the algorithm can only handle noise-free and uniformly-sampled point clouds. For noisy data, this method may fail to produce a water-tight surface.

With real-world scan data, some areas of the surface may be undersampled or completely missing. Automatic techniques will fail to faithfully reconstruct the topology of the surface around these areas. Recently [Sharf et al. 2007] introduced a user-assisted reconstruction algorithm to solve this problem. It asks the user to add local inside/outside constraints at weak regions of unstable topology. An optimal distance field is then computed by minimizing a quadric function combining the data points, user constraints, and a regularization term. This system allows the user to interactively draw scribbles to affect the distance field at a coarse resolution, but the final surface reconstruction at finer resolutions takes several minutes, prohibiting immediate viewing of the reconstructed mesh.

Octree is an important data structure in surface reconstruction algorithms. It is used for representing the implicit function [Ohtake et al. 2004; Kazhdan et al. 2006] and for adaptively extracting isosurfaces [Wilhelms and Gelder 1992; Westermann et al. 1999]. Creating an octree for point clouds directly on the GPU, however, is very difficult, mainly because of memory allocation and pointer creation. Recently, [DeCoro and Tatarchuk 2007] proposed a realtime mesh simplification algorithm based vertex clustering on the GPU. A probabilistic octree is built on the GPU to support adaptive clustering. This octree, however, does not form a complete partitioning of the volume and only contains node information. More importantly, other elements such as faces, edges, and the neighborhood information are missing. Such octrees are not suitable for fast surface reconstruction.

Our parallel surface reconstruction algorithm is implemented using NVIDIA's CUDA [NVIDIA 2007]. In addition to providing a general-purpose C language interface to the GPU, CUDA also exposes new hardware features which are very useful for data-parallel computation. One important feature is that it allows arbitrary gather and scatter memory access from GPU programs. Based on CUDA's framework, researchers have developed a set of parallel primitives, such as *scan*, *compact* and *sort* [Harris et al. 2007a]. Our GPU algorithm makes heavy use of these parallel primitives.

3 GPU Octree Construction

In this section, we describe how to build an octree $\mathcal{O}$ with maximum depth D from a given set of sample points $Q = \{q_i \mid i = 1, ...N\}$. We first explain the design of the octree data structure. Next we present a procedure for the parallel construction of an octree with only individual nodes. Then we introduce an LUT-based technique for efficiently computing the neighborhood information of every octree node in parallel. Finally, we discuss how to collect information of vertices, edges, and faces of octree nodes.

3.1 Octree Data Structure

The octree data structure consists of four arrays: vertex array, edge array, face array, and node array. The vertex, edge, and face arrays record the vertices, edges, and faces of the octree nodes respectively. These arrays are relatively simple. In the vertex array, each vertex v records $v.nodes$, the pointers to all octree nodes that share vertex v. Following $v.nodes$ we can easily reach related elements such as all edges sharing v. In the edge array, each edge records the pointers to its two vertices. Similarly in the face array each face records the pointers to its four edges.

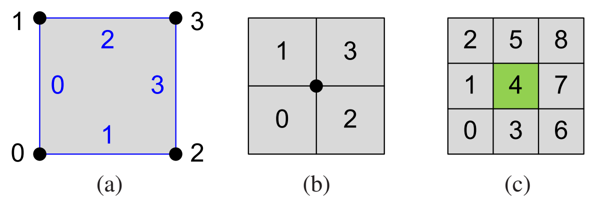


Figure 2: *Element ordering for quadtrees. (a) the ordering of vertices and edges (in blue) in a node; (b) the ordering of a node's children as well as the ordering of nodes sharing a vertex; (c) the ordering of a node's neighboring nodes.*

The node array, which records the octree nodes, is more complex. Each node t in the node array $NodeArray$ contains three pieces of information:

- The shuffled xyz key [Wilhelms and Gelder 1992], $t.key$.
- The sample points contained in t.
- Pointers to related data including its parent, children, neighbors, and other information as explained below.

Shuffled xyz Key: Since each octree node has eight children, it is convenient to number a child node using a 3-bit code ranging from zero to seven. This 3-bit code encodes the subregion covered by each child. We use the xyz convention: if the x bit is 1, the child covers an octant that is "right in x"; otherwise the child covers an octant that is "left in x". The y and z bits are similarly set. The shuffled xyz key of a node at tree depth D is defined as the bit string

$$x_1y_1z_1x_2y_2z_2\cdots x_Dy_Dz_D,$$

indicating the path from the root to this node in the octree. Therefore a shuffled xyz key at depth D has $3D$ bits. Currently we use 32 bits to represent the key, allowing a maximum tree depth of 10. The unused bits are set to zero.

Sample Points: Each octree node records the sample points enclosed by the node. The sample points are stored in a point array and sorted such that all points in the same node are contiguous. Therefore, for each node t, we only need to store the number of points enclosed, $t.pnum$, and the index of the first point, $t.pidx$, in the point array.

Connectivity Pointers: For each node we record the pointers to the parent node, 8 child nodes, 27 neighboring nodes including itself, 8 vertices, 12 edges, and 6 faces. All pointers are represented as indices to the corresponding arrays. For example, t's parent node is $NodeArray[t.parent]$ and t's first neighboring node is $NodeArray[t.neighs[0]]$. If the pointed element does not exist, we set the corresponding pointer to -1. Since each node has 27 neighbors at the same depth, the array $t.neighs$ is of size 27.

For consistent ordering of the related elements, we order these elements according to their shuffled xyz keys. For example, t's first child node $t.children[0]$ has the smallest key among t's eight children and the last child $t.children[7]$ has the largest key. For a vertex, we define its key value as the sum of the keys of all nodes sharing the vertex. This way vertices can also be sorted. Similarly, edges and faces can be sorted as well. Fig. 2 illustrates the ordering of the related elements for quadtrees; the case with octrees is analogous.

3.2 Building Node Array

We build the node array using a reverse level-order traversal of the octree, starting from the finest depth D and moving towards the root, one depth at a time.

Listing 1 Build the Node Array

```
1: // Step 1: compute bounding box
2: Compute $Q$'s the bounding box using Reduce primitive

3: // Step 2: compute shuffled $xyz$ key and sorting code
4: $code \leftarrow$ new array
5: for each $i = 0$ to $N-1$ in parallel
6:     Compute $key$, $q_i$'s shuffled $xyz$ key at depth $D$
7:     $code[i] = key << 32 + i$

8: // Step 3: sort all sample points
9: $sortCode \leftarrow$ new array
10: Sort($sortCode$, $code$)
11: Generate the new point array according to $sortCode$

12: // Step 4: find the unique nodes
13: $mark \leftarrow$ new array
14: $uniqueCode \leftarrow$ new array
15: for each element $i$ in $sortcode$ in parallel
16:     if $sortCode[i].key \neq sortCode[i-1].key$ then
17:         $mark[i] =$ true
18:     else
19:         $mark[i] =$ false
20: Compact($uniqueCode$, $mark$, $sortCode$)
21: Create $uniqueNode$ according to $uniqueCode$

22: // Step 5: augment $uniqueNode$
23: $nodeNums \leftarrow$ new array
24: $nodeAddress \leftarrow$ new array
25: for each element $i$ in $uniqueNode$ in parallel
26:     if element $i-1$ and $i$ share the same parent then
27:         $nodeNums[i] = 0$
28:     else
29:         $nodeNums[i] = 8$
30: Scan($nodeAddress$, $nodeNums$, +)

31: // Step 6: create $NodeArray_D$
32: Create $NodeArray_D$
33: for each element $i$ in $uniqueNode$ in parallel
34:     $t = uniqueNode[i]$
35:     $address = nodeAddress[i] + t.x_Dy_Dz_D$
36:     $NodeArray_D[address] = t$
```

At Depth D: Listing 1 provides the pseudo code for the construction of $NodeArray_D$, the node array at depth D. This construction consists of six steps. In the first step, the bounding box of the point set Q is computed. This is done by carrying out parallel reduction operations [Popov et al. 2007] on the coordinates of all sample points. The **Reduce** primitive performs a scan on an input array and outputs the result of a binary associative operator, such as min or max, applied to all elements of the input array.

In the second step, we compute the 32-bit shuffled xyz keys at depth D for all sample points in parallel. Given a point p, its shuffled xyz key is computed in a top-down manner. The x bit at depth d, $1 \leq d \leq D$, is computed as:

$$x_d = \begin{cases} 0, & \text{if } p.x < C_d.x, \\ 1, & \text{otherwise,} \end{cases}$$

where C_d is the centroid of the node that contains p at depth $d-1$. The y and z bits y_d and z_d are similarly computed. All unused bits are set to zero. We also concatenate the shuffled xyz key and the 32-bit point index to a 64-bit code for the subsequent sorting operation.

In the third step, all sample points are sorted using the sort primitive in [Harris et al. 2007a]. This primitive first performs a split-based radix sort per block and then a parallel merge sort of blocks [Harris et al. 2007b]. After sorting, points having the same key are contiguous in the sorted array. Then the index of each sample point in the original point array is computed by extracting the lower 32 bits of the point's code. The new point array is then constructed by copying the positions and normals from the original point array using the extracted indices.

In the fourth step, a unique node array is generated by removing duplicate keys in the sorted array, as follows. First, for each element of the sorted array, the element is marked as invalid if its key value equals that of its preceding element in the array. Then, the compact primitive from [Harris et al. 2007a] is used to generate the unique node array which does not contain invalid elements. During this process, the relationship between the point array and the node array can be easily built. Specifically, for each element of the node array, we record the number of points contained by this node and the index of the first point in the point array.

In the fifth step, the unique node array obtained in the last step is augmented to ensure that each node's seven siblings are also included, since each octree node has either eight or zero children. In lines 25 $\sim$ 29 of the pseudo code, each element in the unique node array is checked to see if it shares the same parent with the preceding element. This is done by comparing their keys. If the result is yes, $nodeNums[i]$ is set to zero; otherwise it is set to eight. Then a parallel prefix sum/scan primitive is performed on the array $nodeNums$, and the result is stored in the array $nodeAddress$. Each element of $nodeAddress$ thus holds the sum of all its preceding elements in $nodeNums$. In other words, $nodeAddress$ contains the starting address of every unique node in the final node array.

In the final step, the node array $NodeArray_D$ is created. For each node that is added in the fifth step, only the key value is computed and the number of points contained is set to zero. For each node in $uniqueNode$, we locate its corresponding element in $NodeArray_D$ through $nodeAddress$ and its 3-bit $x_D y_D z_D$ key, and copy the node's data to this element. For each sample point in the point array, we also save the pointer to the octree node that contains it.

At Other Depths: The node array at depth $D - 1$ can be easily built from $NodeArray_D$. Recall that the eight siblings having the same parent are contiguous in $NodeArray_D$. For every eight sibling nodes in $NodeArray_D$, a parent node is generated by setting the last three bits of the keys of these nodes to zero. Again, the resulting parent nodes are augmented as in the fifth step above to generate the final array $NodeArray_{D-1}$. At this point, each node in $NodeArray_D$ can get the index of its parent node in $NodeArray_{D-1}$. For each node t in $NodeArray_{D-1}$, the pointers to its children are saved. The number of points $t.pnum$ is computed as the sum of that of t's children, and the index of the first point $t.pidx$ is set to be that of t's first child.

The node arrays at other depths can be built the same way. The node arrays of all depths are then concatenated to form a single node array, $NodeArray$. Another array $BaseAddressArray$ is also created, with each element of the array recording the index of the first node at each depth in $NodeArray$.

3.3 Computing Neighborhood Information

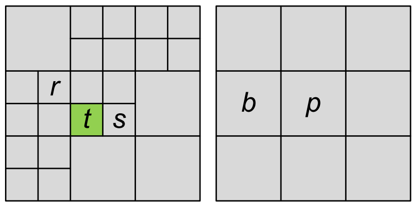
For each octree node in $NodeArray$, we wish to find its neighboring octree nodes at the same depth. This neighborhood information is not only critical for computing the implicit function and running the marching cubes procedure as described in Section 4, but also important for building the vertex, edge, and face arrays.

Listing 2 Compute Neighboring Nodes

```
for each node $t$ at depth $d$ in parallel
    for each $j = 0$ to 26 in parallel
        $i \leftarrow t$'s 3-bit $xyz$ key
        $p \leftarrow NodeArray[t.parent]$
        if $p.neighs[LUTparent[i][j]] \neq -1$ then
            $h \leftarrow NodeArray[p.neighs[LUTparent[i][j]]]$
            $t.neighs[j] = h.children[LUTchild[i][j]]$
        else
            $t.neighs[j] = -1$
```

```
LUTparent[4][9] = {
  {0, 1, 1, 3, 4, 4, 3, 4, 4},
  {1, 1, 2, 4, 4, 5, 4, 4, 5},
  {3, 4, 4, 3, 4, 4, 6, 7, 7},
  {4, 4, 5, 4, 4, 5, 7, 7, 8} };
LUTchild[4][9] = {
  {3, 2, 3, 1, 0, 1, 3, 2, 3},
  {2, 3, 2, 0, 1, 0, 2, 3, 2},
  {1, 0, 1, 3, 2, 3, 1, 0, 1},
  {0, 1, 0, 2, 3, 2, 0, 1, 0} };
```

(a) LUTs for quadtrees



(b) compute node t's neighboring nodes

Figure 3: *Compute neighboring nodes for quadtrees.*

Each node has up to 26 neighbors at the same depth, distributed among its sibling nodes and the child nodes of its parent's neighbors. A naive approach for computing the neighbors is to enumerate all these candidate nodes, which requires $26 \times 27 \times 8 = 5616$ searches for each node (26 neighbors, its parent and 26 neighbors of its parent, each neighbor having 8 children). Our observation is that a node's neighbors are determined by the relative position of the node with respect to its parent and its parent's neighbors. Based on this observation we precompute two look up tables to significantly speed up this neighborhood computation. These two LUTs are defined as follows

Parent Table The parent table $LUTparent$ is a 2D array providing the following information: For an octree node t whose parent is p, if t's index (or xyz key) in $p.children$ is i, then the index of $t.neighs[j]$'s parent in $p.neighs$ is $LUTparent[i][j]$.

Child Table The child table $LUTchild$ is a 2D array with the following information: For the node t with parent p and index i in $p.children$ as above, if node t's j-th neighbor $t.neighs[j]$, whose parent node is h, the index of $t.neigh[j]$ in $h.children$ is $LUTchild[i][j]$.

The size of both tables is 8×27. For convenience we regard a node as a neighbor of itself with index 13 in $neighs$.

Note that we distinguish two kinds of pointers. The *direct* pointers are those represented as indices into one of the "global" arrays: the node, vertex, edge, and face arrays. For example, $t.parent$ is a direct pointer. The *indirect* pointers are those represented as indices into one of the "local" arrays of a node: $t.neighs$, $t.children$, $t.vertices$, $t.edges$, and $t.faces$. The above two tables both record only indirect pointers, which are independent of specific instances of octrees and hence can be precomputed.

Listing 2 provides the pseudo code for computing the neighboring nodes for each node t at depth d in parallel. First, we fetch t's parent p and its xyz key, which is t's index in $p.children$. To compute t's j-th neighbor $t.neighs[j]$, we get this neighbor's parent node h by querying $LUTparent$ and then get the neighbor using a second query to $LUTchild$. Compared with the naive enumeration approach, our technique only needs 27 searches and is over two orders of magnitude faster.

For clarity we use quadtrees to illustrate Listing 2. The two tables for quadtrees, $LUTparent$ and $LUTchild$, are of size 4×9 as shown in Fig. 3(a). As shown in Fig. 3(b), the quadtree node t's parent is p, and t's index in $p.children$ is 0 (i.e., $i = 0$). To compute t's 2-th neighbor (i.e., $j = 2$), we first get p's 1-th neighbor, which is b, according to $LUTparent[0][2] \equiv 1$. Since $LUTchild[0][2] \equiv 3$, b's 3-th child, which is r, is the neighboring node we want. Therefore, $t.neighs[2] = b.children[3] = r$.

To compute t's 7-th neighbor (i.e., $j = 7$), we first get p's 4-th neighbor, which is p itself, according to $LUTparent[0][7] \equiv 4$. Since $LUTchild[0][7] \equiv 1$, p's 1-th child, which is s, is the node we want. Therefore, $t.neighs[7] = p.children[1] = s$.

When computing a node's neighbors, its parent's neighbors are required. For this reason we perform Listing 2 for all depths using a (forward) level-order traversal of the octree. If node t's j-th neighbor does not exist, $t.neighs[j]$ is set as -1. For the root node, all its neighbors is -1 except its 13-th neighbor which is the root itself.

3.4 Computing Vertex, Edge, and Face Arrays

Vertex Array: Each octree node has eight corner vertices. Simply adding the eight vertices of every node into the vertex array will introduce many duplications because a corner may be shared by up to eight nodes. A simple way to create a duplication-free vertex array is to sort all the candidate vertices by their keys and then remove duplicate keys, just as we did for the node array in Section 3.2. This approach, however, is inefficient due to the large number of nodes. For example, for the Armadillo example shown in Fig. 1, there are around 670K nodes at depth 8 and the number of candidate vertices is over 5M. Sorting such a large array takes over 100ms.

We present a more efficient way to create the vertex array by making use of node neighbors computed in Section 3.3. Building the vertex array at octree depth d takes the following steps. First, we find in parallel a unique *owner node* for every corner vertex. The owner node of a corner is defined as the node that has the smallest shuffled xyz key among all nodes sharing the corner. Observing that all nodes that share corners with node t must be t's neighbors, we can quickly locate the owner of each corner from t's neighbors. Second, for each node t in parallel, all corner vertices whose owner is t itself are collected. The unique vertex array is then created. During this process, the vertex pointers $t.vertices$ are saved. For each vertex v in the vertex array, the node pointers $v.nodes$ are also appropriately set.

To build the vertex array of all octree nodes, the above process is performed at each depth independently, and the resulting vertex arrays are concatenated to form a single vertex array. Unlike the node array, the vertex array so obtained still has duplicate vertices between different depths. However, since this does not affect our subsequent surface reconstruction, we leave these duplicate vertices as they are in our current implementation.

Other Arrays: The edge and face arrays can be built in a similar way. For each edge/face of each node, we first find its owner node. Then the unique edge/face array is created by collecting edges/faces from the owner nodes.

4 GPU Surface Reconstruction

In this section we describe how to reconstruct surfaces from sample points using the octree constructed in the last section. The reconstruction roughly consists of two steps. First, an implicit function φ over the volume spanned by the octree nodes is computed using Poisson surface reconstruction [Kazhdan et al. 2006]. Then, an adaptive marching cubes procedure extracts a watertight mesh as an isosurface of the implicit function.

Note that, instead of Poisson surface reconstruction, we may use other methods (e.g., [Hoppe et al. 1992] and [Ohtake et al. 2004]) for GPU surface reconstruction. We chose the Poisson approach because it can reconstruct high quality surfaces through a global optimization. In addition, the Poisson approach only requires solving a well-conditioned sparse linear system, which can be efficiently done on the GPU.

Specifically, we perform the following steps on the GPU:

1. Build a linear system $\mathbf{Lx} = \mathbf{b}$, where $\mathbf{L}$ is the Laplacian matrix and $\mathbf{b}$ is the divergence vector;
2. Solve the above linear system using a multigrid solver,
3. Compute the isovalue as an average of the implicit function values at sample points,
4. Extract the isosurface using marching cubes.

The mathematical details of Poisson surface reconstruction (Step 1 and 2) are reviewed in Appendix A. In the following, we describe the GPU procedures for these steps.

4.1 Computing Laplacian Matrix L

As described in Appendix A, the implicit function φ is a weighted linear combination of a set of blending functions $\{F_o\}$ with each function F_o corresponding to a node of the octree. An entry of the Laplacian matrix $\mathbf{L}_{o,o'} = \langle F_o, \Delta F_{o'} \rangle$ is the inner product of blending function F_o and the Laplacian of $F_{o'}$.

The blending function F_o is given by a fixed basis function F:

$$F_o(q) = F\left(\frac{q - o.c}{o.w}\right)\frac{1}{o.w^3},$$

where $o.c$ and $o.w$ are the center and width of the octree node o. F is non-zero only inside the cube $[-1, 1]^3$. As explained in Appendix A, F is a separable function of x, y and z. As a result, the blending function F_o is separable as well and can be expressed as:

$$F_o(x, y, z) = f_{o.x,o.w}(x) f_{o.y,o.w}(y) f_{o.z,o.w}(z).$$

Given the definition of Laplacian $\Delta F_{o'} = \frac{\partial^2 F_{o'}}{\partial x^2} + \frac{\partial^2 F_{o'}}{\partial y^2} + \frac{\partial^2 F_{o'}}{\partial z^2}$, the Laplacian matrix entry $\mathbf{L}_{o,o'}$ can be computed as:

$$\mathbf{L}_{o,o'} = \left\langle F_o, \frac{\partial^2 F_{o'}}{\partial x^2} \right\rangle + \left\langle F_o, \frac{\partial^2 F_{o'}}{\partial y^2} \right\rangle + \left\langle F_o, \frac{\partial^2 F_{o'}}{\partial z^2} \right\rangle =$$
$$\langle f_{o.x,o.w}, f''_{o'.x,o'.w} \rangle \langle f_{o.y,o.w}, f_{o'.y,o'.w} \rangle \langle f_{o.z,o.w}, f_{o'.z,o'.w} \rangle +$$
$$\langle f_{o.x,o.w}, f_{o'.x,o'.w} \rangle \langle f_{o.y,o.w}, f''_{o'.y,o'.w} \rangle \langle f_{o.z,o.w}, f_{o'.z,o'.w} \rangle +$$
$$\langle f_{o.x,o.w}, f_{o'.x,o'.w} \rangle \langle f_{o.y,o.w}, f_{o'.y,o'.w} \rangle \langle f_{o.z,o.w}, f''_{o'.z,o'.w} \rangle.$$

All the above inner products can be efficiently computed by looking up two precomputed 2D tables: one for $\langle f_o, f_{o'} \rangle$ and the other for $\langle f_o, f''_{o'} \rangle$. These two tables are queried using the x-bits, y-bits, or z-bits of the shuffled xyz keys of node o and o'. This reduces the table size significantly. For a maximal octree depth 9, the table size is $(2^{10} - 1) \times (2^{10} - 1)$. The table size may be further reduced because the entries of the tables are symmetric.

4.2 Evaluating Divergence Vector b

As described in Appendix A, the divergence coefficients b_o can be computed as:

$$b_o = \sum\nolimits_{o' \in \mathcal{O}^D} \vec{v}_{o'} \cdot \vec{u}_{o,o'},$$

where $\vec{u}_{o,o'} = \langle F_o(q), \nabla F_{o'} \rangle$. $\mathcal{O}^D$ is the set of all octree nodes at depth D. The inner product $\langle F_o(q), \nabla F_{o'} \rangle$ can be quickly com-

Listing 3 Compute Divergence Vector $\mathbf{b}$

```
// Step 1: compute vector field
for each node $o$ at depth $D$ in parallel
    $\vec{v}_o = 0$
    for $j = 0$ to 26
        $t \leftarrow NodeArray[o.neighs[j]]$
        for $k = 0$ to $t.pnum$
            $i = t.pidx + k$
            $\vec{v}_o += \vec{n}_i F_{q_i,o.w}(o.c)$
// Step 2: compute divergence for finer depth nodes
for $d = D$ to 5
    for each node $o$ at depth $d$ in parallel
        $b_o = 0$
        for $j = 0$ to 26
            $t \leftarrow NodeArray[o.neighs[j]]$
            for $k = 0$ to $t.dnum$
                $idx = t.didx + k$
                $o' \leftarrow NodeArray[idx]$
                $b_o += \vec{v}_{o'} \vec{u}_{o,o'}$
// Step 3: compute divergence for coarser depth nodes
for $d = 4$ to 0
    $divg \leftarrow$ new array
    for node $o$ at depth $d$
        for each depth-$D$ node $o'$ covered by all nodes in
        $o.neighs$ in parallel
            $divg[i] = \vec{v}_{o'} \vec{u}_{o,o'}$
        $b_o =$ Reduce($divg$, +)
```

puted using a precomputed look up table for $\langle f_o, f'_{o'} \rangle$ as in the computation of $\mathbf{L}_{o,o'}$. As for $\vec{v}_{o'}$, it is computed as

$$\vec{v}_{o'} = \sum\nolimits_{q_i \in Q} \alpha_{o',q_i} \vec{n}_i, \tag{1}$$

where α_{o,q_i} is the weight by which each sampling point q_i distributes the normal $\vec{n}_i$ to its eight closest octree nodes at depth-D.

Listing 3 provides the pseudo code for computing the divergence vector $\mathbf{b}$. This computation takes three steps. In the first step, the vector field $\vec{v}_{o'}$ is computed for each octree node o' according to Eq. (1). Since Eq. (1) essentially distributes sample point q_i's normal $\vec{n}_i$ to its eight nearest octree nodes at depth D, vector $\vec{v}_{o'}$ is only affected by the sample points that are contained in either node o' or its 26 neighbors. The pointers to the node neighbors as recorded in Section 3.3 are used to locate these neighbors.

In the second step, the divergence at every finer depth, which is defined as any depth greater than four, is computed in parallel for all nodes, as shown in Step 2 of Listing 3. The most obvious way to accumulate b_o for each octree node o is to iterate through all nodes o' at depth D. However, this costly full iteration is actually not necessary. Since the basis function F's domain of support is the cube $[-1, 1]^3$, $\vec{u}_{o,o'}$ equals zero for a large number node pairs (o, o'). Specifically, we can easily prove that, for node o, only the depth-D nodes whose ancestors are either o or o's neighbors have nonzero $\vec{u}_{o,o'}$. These nodes can be located by iterating over o's neighbors. Note that $t.dnum$ and $t.didx$ are the number of depth-D nodes covered by t and the pointer to t's first depth-D node respectively. These information can be easily obtained and recorded during tree construction.

In the third step, the divergence at every coarser depth, which is defined as any depth no greater than four, is computed. For nodes at a coarser depth, the approach taken in the second step is not appropriate because it cannot exploit the fine-grained parallelism of GPUs.

Listing 4 Compute Implicit Function Value φ_q for Point q

```
$\varphi_q = 0$
$nodestack \leftarrow$ new stack
$nodestack$.push($proot$)
while $nodestack$ is not empty
    $o \leftarrow NodeArray[nodestack$.pop()$]$
    $\varphi_q += F_o(q)\varphi_o$
    for $i = 0$ to 7
        $t \leftarrow NodeArray[o.children[i]]$
        if $q.x - t.x < t.w$ and $q.y - t.y < t.w$ and $q.z - t.z < t.w$
        then
            $nodestack$.push($o.children[i]$)
```

The node number at coarser depths is much smaller than that at finer depths, and the divergence of a node at a coarser depth may be affected by many depth-D nodes. For example, at depth zero, there is only one root node and all depth-D nodes contribute to its divergence. To maximize parallelism, we parallelize the computation over all covered depth-D nodes for nodes at coarser depths. As shown in Step 3 of Listing 3, we first compute the divergence contribution for each depth-D node in parallel and then perform a reduction operation to sum up all contributions.

4.3 Multigrid Solver and Implicit Function Value

The GPU multigrid solver is rather straightforward. For each depth d from coarse to fine, the linear system $\mathbf{L}^d \mathbf{x}^d = \mathbf{b}^d$ is solved using a conjugate gradient solver for sparse matrices [Bolz et al. 2003]. $\mathbf{L}^d$ contains as many as 27 nonzero entries in a row. For each row, the values and column indices of nonzero entries are stored in a fixed-sized array. The number of the nonzero entries is also recorded.

Note that the divergence coefficients at depth d need to be updated using solutions at coarser depths according to Eq. (7) in Appendix A. For the blending function F_o of an arbitrary octree node o, it can be easily shown that only the blending functions of o's ancestors and their 26 neighbors may overlap with F_o. Therefore, we only need to visit these nodes through the pointers stored in $parent$ and $neighs$ fields of node o.

To evaluate the implicit function value at an arbitrary point q in the volume, we need to traverse the octree. Listing 4 shows the pseudo code of a depth-first traversal for this purpose. A stack is used to store the pointers to all nodes to be traversed. For this traversal, a stack size of $8D$ is enough for octrees with a maximal depth D.

Note that the implicit function value of a sample point q_i can be evaluated in a more efficient way, because we already know the depth-D node o where q_i is located. In other words, we only need to traverse octree nodes whose blending function may overlap with that of o. These nodes include o itself, o's neighbors, o's ancestors, and the neighbors of o's ancestors. Once we get the implicit function values at all sample points, the isovalue is computed as an average: $\bar{\varphi} = \sum_i \varphi(q_i)/N$.

4.4 Isosurface Extraction

We use the marching cubes technique [Lorensen and Cline 1987] on the leaf nodes of the octree to extract the isosurface. The output is a vertex array and a triangle array which can be rendered directly.

As shown in Listing 5, the depth-D nodes are processed in five steps. First, the implicit function values are computed for all octree vertices in parallel. As in the case with the sample points, each vertex v's implicit function value can be efficiently computed by traversing only the related nodes, which can be located through the

Listing 5 Marching Cubes

```
// Step 1: compute implicit function values for octree vertices
$vvalue \leftarrow$ new array
for each octree vertex $i$ at depth-$D$ in parallel
    Compute the implicit function value $vvalue[i]$
    $vvalue[i] -= \bar{\varphi}$
// Step 2: compute vertex number and address
$vexNums \leftarrow$ new array
$vexAddress \leftarrow$ new array
for each edge $i$ at depth-$D$ in parallel
    if the values of $i$'s two vertices have different sign then
        $vexNums[i] = 1$
    else
        $vexNums[i] = 0$
Scan($vexAddress$, $vexNums$, +)
// Step 3: compute triangle number and address
$triNums \leftarrow$ new array
$triAddress \leftarrow$ new array
for each node $i$ at depth-$D$ in parallel
    Compute the cube category based the values of $i$'s vertices
    Compute $triNums[i]$ according to the cube category
Scan($triAddress$, $triNums$, +)
// Step 4: generate vertices
Create $VertexBuffer$ according to $vexAddress$
for each edge $i$ at depth-$D$ in parallel
    if $vexNums[i] == 1$ then
        Compute the surface-edge intersection point $q$
        $VertexBuffer[vexAddress[i]] = q$
// Step 5: generate triangles
Create $TriangleBuffer$ according to $triAddress$
for each node $i$ at depth-$D$ in parallel
    Generate triangles based on the cube category
    Save triangles to $TriangleBuffer[triAddress[i]]$
```

pointers stored in $v.nodes$. Second, the number of output vertices is computed with a single pass over the octree edges and the output address is computed by performing a scan operation. Third, each node's cube category is calculated and the number and addresses of output triangles are computed. Finally, in Step 4 and 5 the vertices and triangles are generated and saved. During this process, for each face of each node, if one of its four edges has a surface-edge intersection, the face is deemed to contain surface-edge intersections and we mark the face. This information is propagated to the node's ancestors.

For all leaf nodes at other depths, we first filter out nodes that do not produce triangles in parallel. For each node, if the implicit function values at its eight corners have the same sign and none of its six faces contain surface-edge intersections, the node does not need any further processing. Otherwise, we subdivide the node to depth D. All the depth-D nodes generated by this subdivision are collected to build the new node, vertex and edge arrays. Then, we perform Listing 5 to generate vertices and triangles. This procedure is carried out iteratively until no new triangles are produced. Note that in each iteration, we do not need to handle the nodes subdivided in previous iterations.

Finally, to remove duplicate surface vertices and merge vertices located closely to each other, we compute the shuffled xyz key for each vertex and use the keys to sort all vertices. Vertices having the same key values are merged by performing a parallel compact operation. The elements in the triangle array are updated accordingly and all degenerated triangles are removed. Each triangle's normal is also computed.

Discussion Besides the Poisson method, we can also design GPU algorithms for other implicit reconstruction methods. For example, an early technique [Hoppe et al. 1992] calculates a signed distance field and reconstructs a surface by extracting the zero set of the distance field using the marching cubes. With the octrees we construct, the distance field can be quickly estimated on the GPU: processing each octree vertex in parallel, we locate its nearest sample point by traversing the octree using a procedure similar to that shown in Listing 4 and compute the signed distance between the vertex and a plane defined by the position and normal of this sample point. Then our adaptive marching cubes procedure is applied to extract the zero set surface. As noted in [Kazhdan et al. 2006], the quality of surfaces reconstructed this way is not as good as those produced by the Poisson method.

5 Results and Applications

We have implemented the described surface reconstruction algorithm on an Intel Xeon 3.7GHz CPU with a GeForce 8800 ULTRA (768MB) graphics card.

Implementation Details: The G80 GPU is a highly parallel processor working on many threads simultaneously. CUDA structures GPU programs into parallel thread blocks of up to 512 parallel threads. We need to specify the number of thread blocks and threads per block for those parallel primitives (e.g., `Sort`, `Compact` and `Scan`) and the programs marked **in parallel**. In our current implementation, we use 256 threads for each block. The block number is computed by dividing the total number of parallel processes by the thread number per block. For example, in Step 2 (line 5) of Listing 1, the block number is $N/256$.

Reconstruction Results: We tested our algorithm on a variety of real-world scan data. As a preprocess, normals are computed using Stanford's Scanalyze system. As shown in Fig. 1, our GPU algorithm is capable of generating high quality surfaces with fine details from noisy real-world scans, just like the CPU algorithm in [Kazhdan et al. 2006].

In terms of performance, the GPU algorithm is over two orders of magnitude faster than the CPU algorithm. For example, for the Stanford Bunny, the GPU algorithm runs at 5.2 frames per second, whereas the CPU algorithm takes 39 seconds for a single frame.

As summarized in Table 1, the GPU algorithm achieves interactive performance for all examples shown in the paper. Currently, the implicit function computation, especially the stage of building the linear system, is the bottleneck of our algorithm. The time for octree construction occupies a relatively small fraction. Compared with the CPU octree construction algorithm, our GPU octree builder is also over two orders of magnitude faster.

One limitation of our GPU reconstruction is that currently it can only handle octrees with a maximal depth of 9 due to the limited memory of current graphics cards. This limitation, however, can be solved with the rapid improvements in graphics hardware.

5.1 User-Guided Surface Reconstruction

Using our GPU reconstruction technique, we develop a user-guided surface reconstruction algorithm for imperfect scan data. The algorithm allows the user to draw strokes to reduce topological ambiguities in areas that are under-sampled or completely missing in the input data. Since our GPU reconstruction technique is interactive, the user can view the reconstructed surface immediately after drawing a stroke. Compared with a previous user-assisted method

Model	# Points	Tree Depth	# Triangles	Memory	T_{ot}	T_{func}	T_{iso}	T_{total}	FPS	T^{ot}_{cpu}	T_{cpu}
Bunny	353272	8	228653	290MB	40ms	144ms	6ms	190ms	5.26	8.5s	39s
Buddha	640735	8	242799	320MB	50ms	167ms	35ms	252ms	3.97	16.1s	38s
Armadillo	512802	8	201340	288MB	43ms	149ms	5ms	197ms	5.06	12.8s	42s
Elephant	216643	8	142197	391MB	46ms	209ms	41ms	296ms	3.38	5.5s	34s
Hand	259560	8	184747	253MB	36ms	143ms	27ms	206ms	4.85	6.4s	26s
Dragon	1565886	9	383985	460MB	251ms	486ms	23ms	760ms	1.31	39.1s	103s

Table 1: *Running time and memory performance for some examples shown in the paper. # Triangles is the number of triangles in the reconstructed surface. T_{ot}, T_{func}, T_{iso} and T_{total} are the time for building octree, implicit function computation (including both linear system building and solving), isosurface extraction and total time respectively, using our GPU algorithm. FPS is the frame rates of our algorithm. For comparison, T^{ot}_{cpu} and T_{cpu} are the octree building time and total time using the CPU algorithm [Kazhdan et al. 2006].*

[Sharf et al. 2007] which takes several minutes to update the reconstructed mesh, our approach is more effective and provides better user experience.

Our basic idea is to first add new oriented sample points to the original point cloud based on user interaction. Then a new isosurface is generated for the augmented point cloud. Suppose Q is the original point set and Q' is the current point set after each user interaction. After the user draws a stroke, our system takes the following steps to generate the new surface:

1. Compute the depth range of Q's bounding box under the current view.
2. Iteratively extrude the stroke along the current view direction in the depth range, with a user-specified interval w. For each extruded stroke, a set of points are uniformly distributed along the stroke, also with interval w. Denote this point set as S.
3. For points in S, compute their implicit function values in parallel using the procedure in Listing 4.
4. Remove points from S whose implicit function values are not less than the current isovalue $\bar{\varphi}$.
5. Compute normals for all points in S.
6. Add S to the current point set Q'.
7. Perform GPU reconstruction with Q' as input and generate the new isosurface.

In Step 2, the interval w is set to be the width of an octree node at depth D by default. Step 4 removes points outside of the current reconstructed surface because we only wish to add new points in inner regions, where topological ambiguity is found. This scheme works well for all tested data shown in this paper. Note that unwanted points may be accidentally introduced in some inner regions. When this happens, the user can remove those points manually. In Step 7, the new isovalue is always computed as the average of the implicit function values of points in the original point set Q because we want to restrict the influence of newly-added points to local areas. The new points are only used to change the local vector field.

Our current system provides two ways to compute the normals for points in S in Step 5. One is based on normal interpolation. For each point $s_i \in S$, we traverse the octree of Q' and find all points of Q' which are enclosed by a box centered at s_i. Then s_i's normal is computed as an interpolation of the normals of these points. The interpolation weight of a point q' is proportional to the reciprocal of the squared distance between q' and s_i. The box size is a user-specified parameter. If no point is found given the current box size, the algorithm automatically increases the box size and traverses the octree again. The other scheme for computing the normals is relatively simple. The normals are restricted to be orthogonal to both the current viewing direction and the tangents of the stroke. We always let the normals point to the right side of the stroke.

Note that for the first normal computation scheme, the user's interaction is not limited to drawing strokes. We also allow users to draw a rectangle or any closed shape to define an area where they want to insert new points. This shape is then extruded and a set of points is uniformly distributed inside the extruded shape. After that, Steps $3 \sim 7$ are performed to generate a new isosurface.

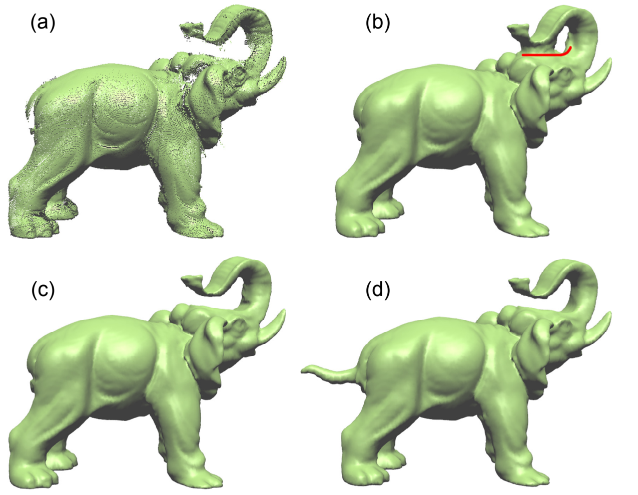


Figure 4: *User-guided reconstruction of a scanned elephant model. (a) The input scan. (b) The result from automatic reconstruction. The head and trunk are mistakenly connected. (c) The improved surface after the user draws the stroke shown in (b). (d) A tail copied from the Armadillo is added around the rear end of the elephant. A new elephant surface with the new tail is immediately reconstructed. See the companion video for live demos.*

User-Guided Reconstruction Results: We tested our algorithm on a variety of complex objects including the Buddha (Fig. 1), Elephant (Fig. 4), and Hand (Fig. 5). For all examples, we were able to generate satisfactory results after several strokes. See the companion video for examples of user interaction sessions.

While the user-specified inside/outside constraints in [Sharf et al. 2007] only correct the local topology, our system also allows the user to specify the geometry of missing areas of the surface. The user first copies a set of points from another point cloud and places the points around the target area. The new isosurface can be then generated. Note that in this case, we do not remove the points outside of the surface as in Step 4 above. Fig. 4(d) shows such an example.

5.2 Point Cloud Modeling

Our GPU reconstruction algorithm can also be integrated into point cloud modeling tools to generate meshes for dynamic point clouds on the fly. The reconstructed meshes can be directly rendered using conventional polygon-based rendering methods.

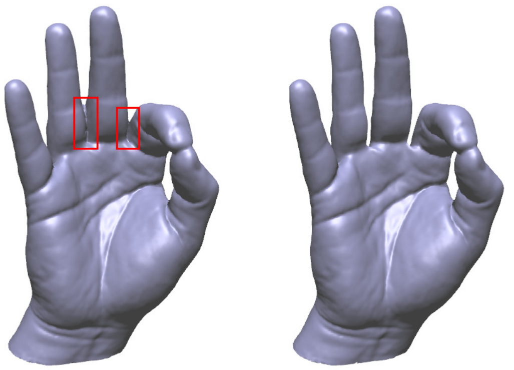


Figure 5: *User-guided reconstruction of a scanned hand model. Left: the automatic reconstruction result. Several fingers are mistakenly connected. Right: the improved surface after the user draws two rectangles.*

Free-Form Deformation: We first implemented the free-form deformation tool described in [Pauly et al. 2003]. The GPU reconstruction is performed on the deformed point cloud at each frame to produce a triangular mesh. As shown in Fig. 6 and the companion video, our system is capable of generating high quality surfaces for large deformation at interactive frame rates, even as dynamic sampling is enabled.

Boolean Operations: Boolean operations are useful for combining several shapes to build complex models. Suppose Q_1 and Q_2 are two point clouds. First, two implicit functions (φ_1 and φ_2) are computed for Q_1 and Q_2 respectively and two isosurfaces M_1 and M_2 are extracted. Second, for each point $q_2^i \in Q_2$ in parallel, the implicit function value $\varphi_1(q_2^i)$ is computed using the pseudo code in Listing 4. Similarly, for each point $q_1^i \in Q_1$, $\varphi_2(q_1^i)$ is computed. Third, the inside/outside classification is done by comparing each $\varphi_1(q_2^i)$ with $\bar{\varphi}_1$, and each $\varphi_2(q_1^i)$ with $\bar{\varphi}_2$. Fourth, based on the inside/outside classification, a new point cloud Q is produced by collecting points from Q_1 and Q_2 according to the definition of the specific Boolean operation being performed. At this point, we may need to flip point normals for some Boolean operations. For example, for $M_1 - M_2$, the new point set consists of Q_1's points that are outside of M_2, plus Q_2's points that are inside of M_1. The normals of Q_2's points need to be flipped. Finally, GPU reconstruction is performed on Q to generate a surface for the Boolean operation.

Fig. 6 shows some results generated using our algorithm. Please refer to the companion video for interactive demos.

6 Conclusion and Future Work

We have presented a parallel surface reconstruction algorithm that runs entirely on the GPU. For moderate-sized scan data, this GPU algorithm generates high quality surfaces with fine details at interactive frame rates, which is over two orders of magnitude faster than CPU algorithms. Our GPU reconstruction algorithm not only enhances existing applications but also opens up new possibilities. To demonstrate its potential, we integrate the algorithm into a user-guided reconstruction system for imperfect scan data and thus enable interactive reconstruction according to user input. We also show how to employ the algorithm in point cloud modeling tools for generating polygonal surfaces from dynamic point clouds on the fly.

For future work, we are interested in exploring the scenario with unreliable normals given at the sample points. In this case, a possible approach is to use the inside/outside constraints [Sharf et al. 2007] instead of normal constraints in implicit function optimization. We

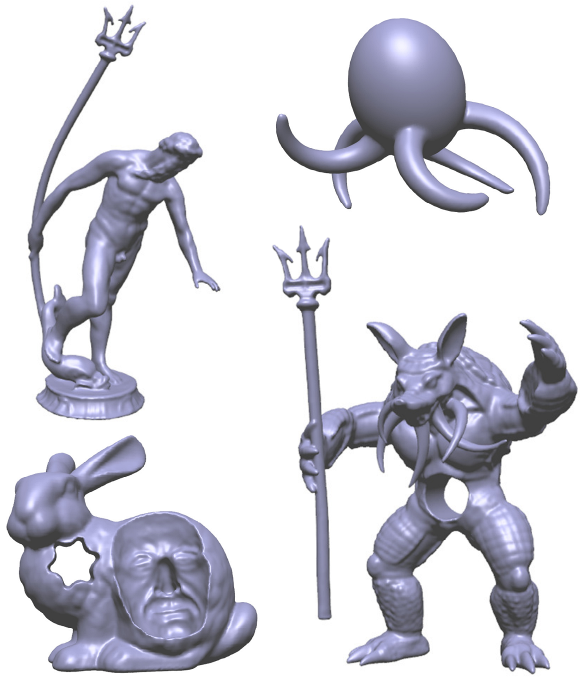


Figure 6: *Free form deformation and boolean operations. Top left: a Neptune model is bent. Top right: several tentacles are pulled out from an ellipsoid. Bottom left: a hole and a face mask are created on the bunny's surface. Bottom right: an interesting creature is created from the armadillo using free-form deformation and boolean operations.*

are also interested in enhancing our user-guided surface reconstruction by developing an automatic method for detecting problematic regions as in [Sharf et al. 2007]. Such a method will save the user the trouble of having to locate these topologically unstable regions. Finally, we believe that our GPU technique for real-time octree construction could be useful in a wide variety of graphics applications since the constructed octrees provide fast access to all tree nodes as well as their neighborhood information. Therefore, an important part of our future plan is to investigate the applications of our octree technique.

References

ALEXA, M., BEHR, J., COHEN-OR, D., FLEISHMAN, S., LEVIN, D., AND SILVA, C. T. 2001. Point set surfaces. In *Proceedings of IEEE Visualization'01*, 21–28.

ALLIEZ, P., COHEN-STEINER, D., TONG, Y., AND DESBRUN, M. 2007. Voronoi-based variational reconstruction of unoriented point sets. In *Proceedings of SGP'07*, 39–48.

AMENTA, N., AND KIL, Y. J. 2004. Defining point-set surfaces. *ACM Trans. Graph. 22*, 3, 264–270.

AMENTA, N., BERN, M., AND KAMVYSSELIS, M. 1998. A new Voronoi-based surface reconstruction algorithm. In *Proceedings of SIGGRAPH'98*, 415–421.

BAJAJ, C. L., BERNARDINI, F., AND XU, G. 1995. Automatic reconstruction of surfaces and scalar fields from 3d scans. In *Proceedings of SIGGRAPH'95*, 109–118.

BOISSONNAT, J.-D. 1984. Geometric structures for three-dimensional shape representation. *ACM Trans. Graph. 3*, 4, 266–286.

BOLZ, J., FARMER, I., GRINSPUN, E., AND SCHRÖDER, P. 2003. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Trans. Graph. 22*, 3, 917–924.

BORGHESE, N. A., FERRARI, S., AND PIURI, V. 2002. Real-time surface reconstruction through HRBF networks. In *Proceedings of the Fourth IEEE International Workshop on Haptic Virtual Environments and Their Applications*, 19–24.

BUCHART, C., BORRO, D., AND AMUNDARAIN, A. 2007. A GPU interpolating reconstruction from unorganized points. In *ACM SIGGRAPH 2007 posters*.

CARR, J. C., BEATSON, R. K., CHERRIE, J. B., MITCHELL, T. J., FRIGHT, W. R., MCCALLUM, B. C., AND EVANS, T. R. 2001. Reconstruction and representation of 3d objects with radial basis functions. In *Proceedings of SIGGRAPH'01*, 67–76.

CURLESS, B., AND LEVOY, M. 1996. A volumetric method for building complex models from range images. In *SIGGRAPH'96*, 302–312.

DECORO, C., AND TATARCHUK, N. 2007. Real-time mesh simplification using the gpu. In *Proceedings of I3D'07*, 161–166.

GOPI, M., KRISHNAN, S., AND SILVA, C. 2000. Surface reconstruction based on lower dimensional localized Delaunay triangulation. In *Proceedings of Eurographics'00*, 467–478.

HARRIS, M., OWENS, J., SENGUPTA, S., ZHANG, Y., AND DAVIDSON, A., 2007. CUDPP homepage. http://www.gpgpu.org/developer/cudpp/.

HARRIS, M., SENGUPTA, S., AND OWENS, J. 2007. Parallel prefix sum (scan) in CUDA. In *GPU Gems 3*, Addison Wesley, H. Nguyen, Ed., Ch.31.

HOPPE, H., DEROSE, T., DUCHAMP, T., MCDONALD, J., AND STUETZLE, W. 1992. Surface reconstruction from unorganized points. In *Proceedings of SIGGRAPH'92*, 71–78.

HORNUNG, A., AND KOBBELT, L. 2006. Robust reconstruction of watertight 3d models from non-uniformly sampled point clouds without normal information. In *Proceedings of SGP'06*, 41–50.

KAZHDAN, M., BOLITHO, M., AND HOPPE, H. 2006. Poisson surface reconstruction. In *Proceedings of SGP'06*, 61–70.

KOLLURI, R., SHEWCHUK, J. R., AND O'BRIEN, J. F. 2004. Spectral surface reconstruction from noisy point clouds. In *SGP'04*, 11–21.

LIPMAN, Y., COHEN-OR, D., AND LEVIN, D. 2007. Data-dependent MLS for faithful surface approximation. In *Proceedings of SGP'07*, 59–67.

LORENSEN, W. E., AND CLINE, H. E. 1987. Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of SIGGRAPH'87*, 163–169.

NVIDIA, 2007. CUDA programming guide 1.0. http://developer.nvidia.com/object/cuda.html.

OHTAKE, Y., BELYAEV, A., ALEXA, M., TURK, G., AND SEIDEL, H.-P. 2004. Multi-level partition of unity implicits. *ACM Trans. Graph. 22*, 3, 463–470.

PAULY, M., KEISER, R., KOBBELT, L. P., AND GROSS, M. 2003. Shape modeling with point-sampled geometry. In *Proceedings of SIGGRAPH'03*, 641–650.

POPOV, S., GÜNTHER, J., SEIDEL, H.-P., AND SLUSALLEK, P. 2007. Stackless kd-tree traversal for high performance GPU ray tracing. In *Proceedings of Eurographics'07*, 415–424.

RANDRIANARIVONY, M., AND BRUNNETT, G., 2002. Parallel implementation of surface reconstruction from noisy samples. Preprint Sonderforschungsbereich 393, SFB 393/02-16.

SHARF, A., LEWINER, T., SHKLARSKI, G., TOLEDO, S., AND COHEN-OR, D. 2007. Interactive topology-aware surface reconstruction. *ACM Trans. Graph. 26*, 3, Article 43, 9 pages.

TURK, G., AND O'BRIEN, J. F. 2002. Modelling with implicit surfaces that interpolate. *ACM Trans. Graph. 21*, 4, 855–873.

WEINERT, K., SURMANN, T., AND MEHNEN, J. 2002. Parallel surface reconstruction. In *Proceedings of the 5th European Conference on Genetic Programming*, 93–102.

WESTERMANN, R., KOBBELT, L., AND ERTL, T. 1999. Real-time exploration of regular volume data by adaptive reconstruction of isosurfaces. *The Visual Computer 15*, 2, 100–111.

WILHELMS, J., AND GELDER, A. V. 1992. Octrees for faster isosurface generation. *ACM Trans. Graph. 11*, 3, 201–227.

A Review of Poisson Surface Reconstruction

Given an input point cloud Q with each sample point q_i having a normal vector $\vec{n}_i$, the Poisson surface reconstruction technique [Kazhdan et al. 2006] computes an implicit function φ whose gradient best approximates a vector field $\vec{V}$ defined by the samples, i.e., $\min_\varphi \|\nabla\varphi - \vec{V}\|$. This minimization problem can be restated as solving the following Poisson equation:

$$\Delta\varphi = \nabla \cdot \vec{V},$$

i.e., compute a scalar function φ whose Laplacian (divergence of gradient) equals the divergence of $\vec{V}$. The algorithm first defines a set of blending functions based on octree $\mathcal{O}$. For every node $o \in \mathcal{O}$, a blending function F_o is defined by centering and stretching a fixed basis function F:

$$F_o(q) \equiv F_{o.c,o.w}(q) = F\left(\frac{q - o.c}{o.w}\right)\frac{1}{o.w^3}, \tag{2}$$

where $o.c$ and $o.w$ are the center and width of node o.

The vector field $\vec{V}$ is then defined as:

$$\vec{V}(q) \equiv \sum_{q_i \in Q} \sum_{o \in \mathcal{O}^D} \alpha_{o,q_i} F_o(q)\vec{n}_i = \sum_{o \in \mathcal{O}^D} \vec{v}_o F_o(q), \tag{3}$$

where $\mathcal{O}^D$ are the octree nodes at depth D, α_{o,q_i} is the interpolation weight. Each sample point q_i only distributes its normal to its eight closest octree nodes at depth D. This works well for all scan data we tested, although it is preferable to also "splat" the samples into nodes at other depths for non-uniformly distributed point samples.

The implicit function φ is also expressed in the function space spanned by $\{F_o\}$:

$$\varphi(q) = \sum_{o \in \mathcal{O}} \varphi_o F_o(q). \tag{4}$$

The Poisson equation thus reduces to a sparse linear system:

$$\mathbf{L}\mathbf{x} = \mathbf{b}, \tag{5}$$

where $\mathbf{x} = \{\varphi_o\}$ and $\mathbf{b} = \{b_o\}$ are $|\mathcal{O}|$-dimensional vectors. The Laplacian matrix entries are the inner products $\mathbf{L}_{o,o'} = \langle F_o, \Delta F_{o'}\rangle$, and the divergence coefficients are

$$\begin{aligned} b_o &= \sum_{o' \in \mathcal{O}^D} \langle F_o, \nabla \cdot (\vec{v}_{o'} F_{o'})\rangle = \sum_{o' \in \mathcal{O}^D} \langle F_o, (\vec{v}_{o'} \cdot \nabla F_{o'})\rangle \\ &= \sum_{o' \in \mathcal{O}^D} \int F_o(q)(\vec{v}_{o'} \cdot \nabla F_{o'}(q))dq = \sum_{o' \in \mathcal{O}^D} \vec{v}_{o'} \cdot \vec{u}_{o,o'}, \end{aligned}$$

where $\vec{u}_{o,o'} = \int F_o(q)\nabla F_{o'}(q)dq$.

The linear system can be transformed into successive linear systems

$$\mathbf{L}^d\mathbf{x}^d = \mathbf{b}^d, \tag{6}$$

one per octree depth d. Since $\mathbf{L}^d$ is symmetric and positive definite, each linear system can be solved using a conjugate gradient solver. The divergence at finer depths is updated as:

$$b_o^d \leftarrow b_o^d - \sum_{d' < d} \sum_{o' \in \mathcal{O}^{d'}} \mathbf{L}_{o,o'}\varphi_{o'}, \tag{7}$$

where $\mathcal{O}^d$ is the set of octree nodes at depth d.

The basis function F used in [Kazhdan et al. 2006] is the n-th convolution of a box filter with itself:

$$F(x,y,z) \equiv (B(x)B(y)B(z))^{*n}, \quad B(t) = \begin{cases} 1, & \text{if } |t| < 0.5; \\ 0, & \text{otherwise.} \end{cases} \tag{8}$$

Kazhdan et al. used $n = 3$ in their implementation. In our implementation we choose $n = 2$ instead. This reduces the support of F to the domain $[-1,1]^3$ without noticeable degradation of the reconstructed surfaces. Note that F is a separable function and can be expressed as:

$$F(x,y,z) = f(x)f(y)f(z), \tag{9}$$

where f is the n-th convolution of B with itself.