SHIC: A View-Dependent Rendering Framework for Isosurfaces

Nan Zhang Huamin Qu Wei Hong Arie Kaufman

Center for Visual Computing (CVC) and Department of Computer Science*
Stony Brook University, Stony Brook, NY 11794-4400, USA

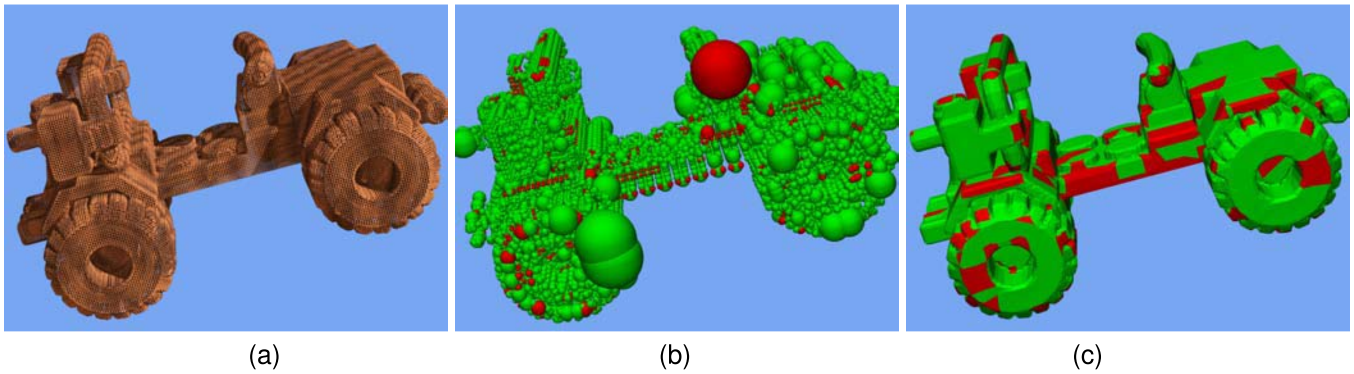


Figure 1: Illustration of our view-dependent isosurface rendering algorithm using a Lego car dataset. (a) The 241K isopoints (isovalue 120.5) extracted from a density volume. (b) The active isopoint set for the current viewpoint (Green: inherited from the previous frame, Red: newly refined or collapsed). The size of each isopoint is proportional to the size of the octree cell where it resides. (c) The isosurface constructed from the active isopoint set (Green: inherited, Red: newly extracted), 61K triangles.

Abstract

We present Selective and Hierarchical Isopoint Clustering (SHIC) as a framework for interactive isosurface visualization. SHIC is an octree-based vertex hierarchy where each surface component in a cell is represented by a vertex (isopoint) with encoded connectivity. We describe a novel connectivity encoding scheme, called Connectivity Encoding Bitmap, and two topology-preserving isopoint clustering algorithms to build the vertex hierarchy. Our framework is able to cluster surface components with up to four intersection points on a cell edge. During rendering, an incremental isosurface extraction algorithm is used to construct the isosurface dynamically. Events associated with vertex tree modifications and active vertices changes are unified using timestamps. Our framework is efficient, space-saving, and suitable for rendering large isosurface objects with local modifications.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Object hierarchies.

Keywords: isosurface simplification, vertex clustering, level-of-detail rendering

1 Introduction

For large volumetric datasets, a large amount of tiny triangles are usually generated by isosurface extraction algorithms. Visualizing them in real time is a challenging task. Level-of-detail (LOD) rendering techniques have been developed to address this problem. Using multiresolution techniques, LOD algorithms improve rendering performance without significantly losing image quality. In isosurface rendering, the LOD techniques can be classified into two categories: (1) isosurface pre-extraction, followed by LOD surface rendering [9, 15]; (2) isosurface extraction on a pre-simplified volume hierarchy. However, algorithms in both categories have shortcomings. For large volumes, isosurface extraction in full resolution is very time-consuming. The extracted mesh may be too large to fit into the main memory. If the volume is dynamically modified, the mesh has to be re-extracted and re-processed, which results in long latency. On the other hand, the pre-simplified volume hierarchy is hard to handle the topology problems arising from hierarchical cell merging. The cracks between different levels have to be patched [20, 22], which slows down the isosurfacing process.

Volume simplification algorithms can be classified into field simplification and isosurface simplification. In the first category, He et al. [6] have used low-pass filters to convolve a 3D volume buffer. This tends only to blur the objects. Tetrahedra refinement [4, 5, 17] is a good strategy to generate a coarse-to-fine hierarchy of the volume, where the longest edge bisection is used. Gerstner and Pajarola [4] have presented a table-based method for topology-preserving volume simplification, where the critical points are extracted and preserved. In isosurface simplification, there are two major types: MC-based [20, 22] and isosurface clustering [11]. The former algorithms decimate the isosurface on an octree-based grid. One major issue is crack patching for different levels. Another is that these algorithms can't handle more than one intersection point on a cell edge. Although point-based clustering [1, 7, 18] has been studied and decimation algorithms, such as principal component analysis, have been presented, isosurface clustering hasn't gained significant attention. Ju et al. [11] have described a topology-preserving algorithm for clustering cells of different materials using one representative vertex per cell. We have used the *Enhanced Cell* representation for isosurface simplification [25]. In terms of representation power, the enhanced cell is equivalent to the method presented in this paper. However, the representation in this paper is more compact. Furthermore, our previous simplification algorithm is limited to at most two intersection points on a cell edge.

*e-mail: {nanzhang|huamin|weihong|ari}@cs.sunysb.edu

IEEE Symposium on Volume Visualization and Graphics 2004
October 11-12, Austin, Texas, USA
0-7803-8781-3/04/$20.00 ©2004 IEEE

Based on the Progressive Mesh (PM) [8] framework for simplifying arbitrary manifold meshes, Xia and Varshney [24] and Hoppe [9] have independently extended PM to support LOD rendering, where vertices in the mesh are organized into a forest. In Hoppe's View-Dependent Progressive Mesh (VDPM) [9], the system maintains a list of active vertices. Using *vertex split* and *edge collapse*, these vertices are merged or refined, according to view parameters. Later, Hoppe [10] has elaborated his algorithm with output-sensitive memory consumption and a geomorphing technique to smoothly interpolate two successive meshes. Luebke and Erikson [15] have presented a dynamic vertex clustering algorithm on a tight octree. Some specialized algorithms have been presented for terrain rendering, such as height field based algorithms [2, 13]. The ROAMing algorithm [2] uses longest edge bisection to subdivide height field cells into crack-free triangles and a dual-queue technique to refine and merge cells. Besides, the tetrahedra refinement strategy [5] is efficient enough for view-dependent isosurface rendering.

In this paper, we present Selective and Hierarchical Isopoint Clustering (SHIC), a view-dependent framework for interactively rendering large isosurfaces. For a user-defined isovalue, we extract the points that lie on the isosurface, or isopoints, as the representation primitives. Each isosurface component in a cell is represented by an isopoint. The isopoints are pre-simplified using our novel isopoint clustering algorithms with topology-preservation and are organized into an octree. During rendering, a set of isopoints are dynamically selected for constructing a crack-free, view-dependent isosurface. Our framework provides an approach for interactive rendering large isosurfaces. It also attempts to tackle the problem of LOD isosurface rendering of volume datasets while permitting local modifications, which is rarely reported in literature.

The primary contributions of this paper are:

- A novel connectivity encoding scheme for isopoints. Each isopoint is associated with an edge bitmap, called Connectivity Encoding Bitmap (CEB), to provide compact encoding.
- Two topology-preserving isopoint clustering algorithms to build the vertex hierarchy. In the hierarchical clustering algorithm, the vertices from different surface components are clustered independently using connectivity bitmaps. The selective clustering algorithm can handle up to four intersection points on a coarse cell edge.
- A view-dependent isosurface extraction algorithm. Our system doesn't store the full mesh. Triangles are constructed on-the-fly in view-dependent rendering.

2 Problem Statement and Design

Our framework is inspired by the Hierarchical Dynamic Simplification (HDS) [15] algorithm for rendering surfaces. However, our system has two unique characteristics: (1) Our clustering algorithm preserves isosurface topology by using multiple representative vertices in a cell. (2) We don't store an explicit mesh of the finest resolution. Instead, the isopoint connectivity is encoded and the mesh used for rendering is dynamically constructed. Our motivation for such an algorithm design is stated as follows:

Vertex Clustering: In this algorithm, the space is partitioned into cells and vertices that fall into the same cell are replaced with one representative vertex. To apply this algorithm, for a given isovalue, the isosurface is extracted from the implicit function $f(x,y,z)$ on the volume grid first. The grid lines split the space into cells. The surface components existing in a cell may be complex. To merge these cells into a coarse cell p, the surface components in p can be even more complex. We only consider the vertex clustering algorithm in a cubic cell environment. Since the isopoints are the vertices used for clustering, we don't distinguish them in this paper. Figure 2 shows a large cell ABCD, which contains 16 child cells. Two surface components, S_1 and S_2, pass this cell. In each child cell, there is at most one surface component. To cluster the surface components, we need to answer three questions:

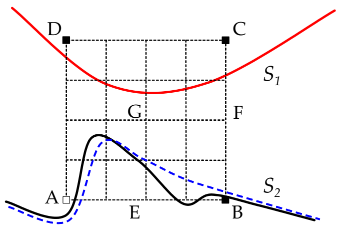


Figure 2: Simplifying two surface components in the volume grid.

- Shall we allow more than one representative vertex in a cell? In the large cell ABCD, there is no doubt that keeping the two surface components is better than merging them into one.
- How to recover the vertex connectivity for cell ABCD? If we use the Marching Cubes (MC) algorithm [14], only one surface component can be extracted. The connection on edge BC is lost since B and C have the same sign.
- How to handle the cases where a surface component is split into several segments by a grid line? For example, in cell EBFG, edge EB cuts S_2 into two segments. How can we reconnect them? Can it be simplified as the blue dash line?

Surface-based vertex clustering algorithms provide no answers to these questions. These algorithms use one vertex per cell and are topology-blind. Existing topology-preserving simplification algorithms [4, 11] are unable to solve these problems, too. These algorithms maintain all critical points. Instead, we use multiple representative vertices to overcome these problems.

Memory and Latency Considerations: The basic rendering elements of an isosurface are triangles, which are usually tiny and vast in amount, since the volume representation has to use high resolution to sample the original object. CSG operations also require the sampling resolution to be high enough to reconstruct the high frequencies created in modeling. The emergence of the feature-sensitive volume modeling techniques [11, 12, 21] require much more memory to store the directed distances. For example, a possible data structure for octree cells can be:

```
struct OctreeCell {
   OctreeCell* child[8];          //pointer to the child cells
   QEF qef;                       //quadric error metric
   HermiteData* edge[12];         //intersection points and normals
   byte sign[8];                  //signs at the corner points
};
```

The above data structure implies that space requirement for storing cells is huge. At the same time, LOD surface rendering algorithms also have significant space consumption, since they only work on pre-extracted meshes. For large isosurfaces, pre-extracting the whole isosurface might be impractical, both in storage and latency. For example, the connectivity space takes a large portion in storage. It may also be unnecessary since not all regions are visualized in full detail. Furthermore, the volumetric scenes can be dynamically modified. Most existing LOD algorithms are suitable for rendering static scenes only and a time-consuming preprocessing stage is required. Therefore, we use encoded connectivity and dynamic surface construction to address these issues.

3 Isosurface Encoding and Clustering

Given a user-defined isovalue, our framework extracts the isosurface components in cells and converts them into the isopoint representation. Each surface component is represented by an isopoint with an associated connectivity bitmap. The isopoints are positioned using the MC algorithm or dual contouring [11]. Two clustering strategies are developed to build the vertex hierarchy: hierarchical clustering and selective clustering.

3.1 Connectivity Encoding Bitmaps

Our encoding scheme is inspired by the SurfaceNets algorithm [19], where a quad is constructed around an edge **e** which shows a sign change. The sign change indicates that the isosurface passes that edge once. Therefore, all of the 4 vertices in the cells that share **e** can be associated with **e**. Given a list of vertices, the quad can be recovered by finding the 4 vertices. For each isopoint representing a surface component S, its associated edges are the edges that S intersects with. We also need to decide the bits required for each edge in the encoding. Following the assumption of [21], the isosurface may pass an edge twice. Since the cell edges are axis-aligned, we order the end points of each edge in its axis direction. Each edge must be one of the following four cases:

- ○ : No isosurface passing;
- + : End point 1 inside and end point 2 outside (1 passing);
- - : End point 1 outside and end point 2 inside (1 passing);
- +-: The isosurface passing the edge twice and both end points showing the same classification value.

We distinguish the sign changes of each edge along its axis direction. If end point 1 is inside the object and end point 2 is out, we call it a + sign change. If reversed, we call it a - sign change. The two cases are called directed sign changes in general. Distinguishing them allows us to setup the correspondence of vertices when two intersection points are found on one edge. We use 2 bits to encode the above 4 cases. Totally a bitmap of 24 bits is required for the 12 edges. We name this scheme Connectivity Encoding Bitmap (CEB). Figure 3 shows the structure of the CEB. Besides building a CEB for each isopoint, we can also construct a CEB for each cell. The CEB of a cell describes the connectivity of all vertices in the cell and is computed by the OR operation on all the vertex CEBs.

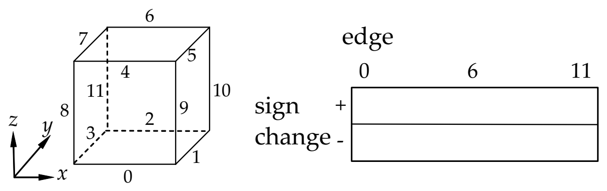


Figure 3: The indexing sequence of the 12 cube edges (left), and the structure of a CEB (right).

The CEBs provide us a way to encode more complex connectivity than the cube-based encoding. A CEB can fully encode the connectivity of an isosurface component. Figure 4 shows some examples, where the cube-based encoding can't represent Figures 4b and 4c. Especially, Figure 4c has singular connectivity. Our motivation of using powerful encoding is to reduce the amount of cells necessary to preserve the isosurface topology. Although more than one representative vertex may appear in a cell, these vertices are temporarily stored and are waiting for chances to be merged in other levels.

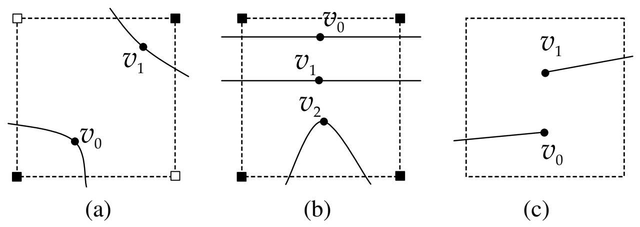


Figure 4: Our encoding scheme can represent: (a) Two disconnected surface components; (b) Two intersection points on an edge, including loops; and (c) Singular connectivity.

3.2 Hierarchical Clustering

In our hierarchical clustering algorithm, the octree cells are merged in a bottom-up manner. A cell p is constructed from its eight child cells, unless the CEB of a vertex in p or the CEB of p can't be constructed. There are two tasks: generating new representative vertices for each cell and creating a new CEB for each vertex.

To generate new representative vertices, we need to determine the surface components first. Vertices in the same surface component of the coarse cell are merged. Vertices are treated as connected if they form quads or triangles. Since the CEBs encode the connectivity, the vertices that share one cell edge and have the same directed sign change must be connected. The connected vertices are collected into a set. To generate the representative vertex of this set, the quadric error metrics [3] are applied. The quadric error functions associated with those vertices are added into $Q(x)$. A representative vertex is computed by minimizing the error of $Q(x)$. To get a robust result, we use the Singular Value Decomposition (SVD) method.

To build the CEB of a vertex, we merge the CEBs of its child vertices. In these CEBs, only the edges that lie on the edges of the coarse cell are used. The directed sign changes on each coarse cell edge is counted. If the count exceeds the limit, this CEB can't be constructed. Otherwise, the corresponding bits on the CEB are set.

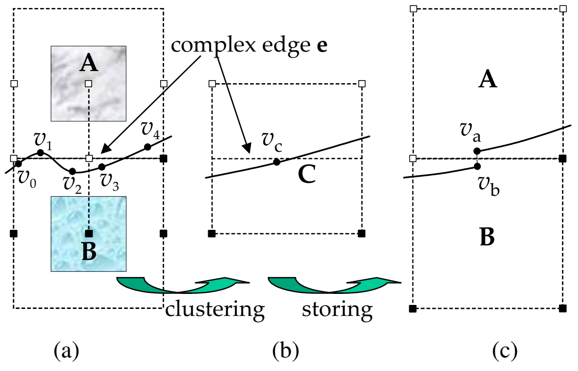
3.3 Selective Clustering

The hierarchical clustering algorithm is restricted by the following two factors:

- The representation power of the CEB.
- The mip-map style merging sequence.

In the first factor, the CEB imposes a limitation on cell merging: edges with more than two intersection points can't be represented. We name these edges *complex edges*. In the merging tree, this will stop the merging of the four coarse cells sharing a complex edge and their parents. In the second factor, the mip-map style hierarchical cell merging enforces that a coarse cell could only be built from child cells. This makes the first factor unavoidable. On the other hand, for the child cell edges which don't lie on the coarse cell edges, the total count of intersection points can exceed the limit. Based on this observation, we present a selective clustering algorithm. In this algorithm, a complex edge is no longer the bottleneck, as long as the intersection points on that complex edge is in one surface component.

This algorithm is illustrated in Figure 5. Figure 5a shows two coarse cells, A and B. The two cells share one complex edge **e**. The isosurface is cut by **e** and three intersection points are created. There are five surface components in the child cells, which are represented by vertices v_0 to v_4. To cluster them together, we create a virtual



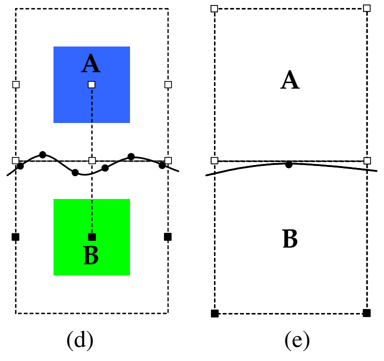


Figure 5: Selective vertex clustering on a complex edge **e**. (a) The isosurface has three intersection points on **e**, which is shared by cells A and B. (b) Virtual cell C is created by merging the four child cells sharing **e**. C is mergable since **e** is an inner edge now. (c) The representative vertex v_c in C is copied to A and B. (d) The four-intersection-point case. (e) The simplified result.

coarse cell C from the 4 child cells sharing **e**, shown in Figure 5b. C contains only v_0 to v_4. In C, edge **e** becomes an inner edge and v_0 to v_4 are in one surface component. Since there is no restriction on the count of intersection points on an inner edge during clustering, C can be merged safely. The clustering process generates one representative vertex v_c. The connectivity bitmap of v_c, CEB_{v_c}, can be constructed from its child vertices using the method in Section 3.2. Note that CEB_{v_c} contains the connectivity from both A and B. Finally, we need to store v_c into the real coarse cells. In each coarse cell, we create a copy of v_c. Suppose it is v_a in A and v_b in B. The connectivity of the two vertices is computed by splitting CEB_{v_c} into two parts: one part that goes from A and another part from B. CEB_{v_a} is former part, while CEB_{v_b} is the latter.

In Figure 5c we show the copied vertices and their connectivity. Note that the connectivity is singular, as there is only one connection for each vertex. This can only be represented by our encoding method. Our algorithm also guarantees that there will be no crack during surface extraction. Although v_a or v_b can only reconstruct a partial triangle fan centered at v_c, the union of them will be a crack-free triangle fan, since v_a and v_b are actually the same vertex. In terms of error measurement, since v_a and v_b are copies of v_c, they are assigned the same error of v_c. Furthermore, to avoid cracks, they should be processed specially in subsequent clustering operations since they are tied as one vertex.

The above algorithm can also be used to merge cells where four intersection points are found on a complex edge, as long as they are in one surface component, such as the case shown in Figure 5d. The difference is that since cell A has no contribution in connectivity (see Figure 5e), there is no need to copy the vertex to A. However, this algorithm can only handle one complex edge per surface component.

4 Our Rendering Framework

The following shows the structure of the representative vertices and octree cells used in the SHIC system:

```
struct RepVertex {
    bit active;                 //vertex activeness in rendering
    bit recursion;              //indicating a path to active decedents
    bit edgevector[24];         //connectivity encoding bitmap (CEB)
    vec repvert;                //representative vertex position
    RepVertex *pParentvert;     //parent vertex to be merged to
    QEF qef;                    //quadric error metrics
    float qerror;               //quadric error of repvert
};

struct OctreeCell {
    ....                        //other octree cell information

    byte vertcount;             //# of representative vertices
    RepVertex *pRepvert;        //pointer to representative vertices
    word timestamp;             //CSG & LOD rendering time stamp
};
```

In the vertex hierarchy, vertices can be classified into active and non-active, according to whether or not they participate in isosurface construction of the current frame. In the RepVertex structure, the *active* field is the flag of activeness. If not active, the *recursion* field indicates whether or not there is a path from this vertex to its active descendants. In additions, the parent vertex link and quadric errors associated with a vertex are stored in *pParentvert* and *qerror* fields, respectively. In the OctreeCell structure, since multiple representative vertices are allowed in a cell, the *qef* field in the previously-defined OctreeCell structure has to be moved to each vertex. A *timestamp* field is used for indicating cell CSG and LOD rendering events. The RepVertex structure costs 31 extra bytes per vertex. Since there is only a small portion of cells that contain multiple vertices, the cost per cell is also close to 31 bytes. This number can be reduced by quantizing some floating point fields.

Our framework contains two stages: a preprocessing stage that simplifies the input isopoint set into a static vertex hierarchy, and an interactive stage that extracts view-dependent isosurfaces from this vertex hierarchy frame by frame.

4.1 Preprocessing

The input isopoint sets are extracted from regular density or directed distance volumes. The hierarchical clustering and selective clustering algorithms are performed. Figure 6 shows our preprocessing algorithm. In this process, the quadric error is computed and stored in each vertex. To keep track of the parent-child relationship, the *pParentvert* pointer is also maintained. There are some restrictions to merge the parent cells of those cells containing complex edges. We must link the parent vertices of the copied vertices. Copied vertices from the same complex edge only vanish after they are all merged into one surface component.

```
procedure Preprocessing(OctreeCell root)
1. hierarchical_clustering(root);      //complex edges ⇒ QUEUE
2. while QUEUE $\neq \emptyset$
3.     edge e ⇐ HEAD_QUEUE();
4.     selective_clustering(e);
5.     for each coarse cell p sharing e
6.         hierarchical_clustering(p); //complex edges ⇒ QUEUE
```

Figure 6: Preprocessing algorithm.

4.2 CSG Updating

We use directed distances for feature-preserving CSG operations. For each cell edge, up to two intersection points with normals are stored. To decide the surface components in a cell, we group the intersection points using the algorithm described in [21]. The system keeps a global timestamp TS for CSG operations. Before a CSG operation, TS is increased by one. The affected region is indicated by the *timestamp* field of the octree cells. Each affected cell is assigned the new timestamp, and hereby its ancestors. After the CSG operations, we re-compute the representative vertices and perform a similar preprocessing on the affected region of the octree.

4.3 Screen-space Error Metric

The screen-space error metric provides an error measurement for each representative vertex in the vertex hierarchy when projected on to the screen. There have been many sophisticated error metrics [9, 13]. Due to lacking of the normal vector for each vertex, we use a simplified computation model. Figure 7 illustrates our error metric, where a representative vertex v has a distance r to the isosurface and d to the viewpoint. Suppose the view angle is $fovy$ and the screen error threshold is τ pixels. The maximum possible screen projection size $Proj_v$ of a line segment of length r is:

$$Proj_v = r \cdot h/(d \cdot fovy) \qquad (1)$$

The error $v.qerror$ is the sum of squared distances of v to a set of triangle planes represented by v. It is an over-estimation of r^2. By replacing r with $v.qerror^{1/2}$ and requiring $Proj_v < \tau$, we have:

$$v.qerror < d^2 \cdot (\tau \cdot fovy/h)^2 \qquad (2)$$

We define a *qrefine* function to compute whether or not a vertex should be refined. *qrefine* is similar to the function defined by Hoppe [9]. It uses two criteria: the view frustum and the screen-space error. This function returns true only if a vertex is inside the view frustum and its screen error satisfies Equation 2.

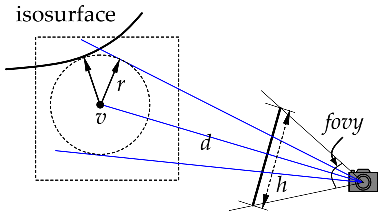


Figure 7: The screen-space error for a representative vertex.

4.4 Active Vertices and Triangles

A set of active vertices is maintained in the vertex hierarchy. The active vertices are not stored separately. Instead, they are just indicated by the *active* field in the RepVert structure. For each frame, constructing the isosurface from the vertex hierarchy contains two steps: (1) Updating the active vertex set. After each updating, the active vertex set is adapted to the current viewpoint. There are three possible actions for each active vertex from the previous frame: collapsing, refinement and no change. (2) Incremental isosurface extraction. The active vertex set is initialized using a view-independent error threshold δ. The *recursion* field of the non-active vertices are initialized to keep track of the paths to the active vertices in the vertex hierarchy.

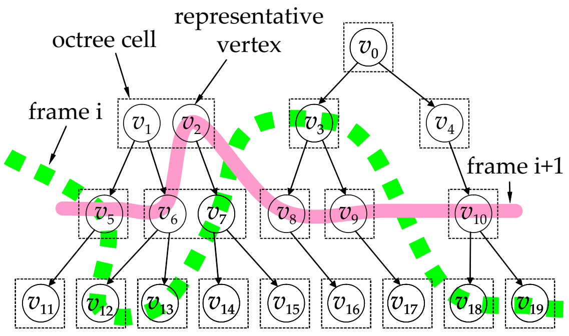


Figure 8: The active vertices of two consecutive frames. Although there may be multiple vertices in an octree cell, these vertices are raised or lowered independently.

Figure 8 shows the refining and collapsing of the active vertex set between two consecutive frames. These operations are efficient since there is no need to manage connectivity. In the adaptive refinement algorithm (see Figure 9), the octree is recursively visited. There may be multiple vertices in a cell. They may not be selected simultaneously, such as v_1 and v_2 in Figure 8. Therefore, we use the *recursion* field to ensure that no surface components are lost. To generate a smooth transition between frames, we specify that for each vertex, the coarsening and refining processes can only raise or lower the vertex by one level.

```
procedure adaptive_refinement(OctreeCell p)
    bool recursion=false;
    for each vertex v in p
        recursion|=v.recursion;
        if v.active and qrefine(v)
            expand_vertex(p, v);
    if recursion or p.pRepvert==null        //there are other paths
        for each child cell q in p
            adaptive_refinement(q);
    for each vertex v in p where !v.active
        if all of v's child vertices can be collapsed
            if !qrefine(v) collapse_vertex(p, v);
procedure expand_vertex(OctreeCell p, RepVertex v)
    v.active=false, v.recursion=true;
    for each child cell of p
        for each child vertex t of v
            t.active=true;
procedure collapse_vertex(OctreeCell p, RepVertex v)
    v.active=true, v.recursion=false;
    for each child cell q of p
        for each child vertex t of v
            if t.active==true
                t.active=false;
            else
                t.recursion=false;
                t's descendants active, recursion fields ⇐ false;
```

Figure 9: Adaptive vertex refinement algorithm.

The adaptive refinement procedure has no difficulty in handling the copied vertices in Section 3.3. These vertices are treated equally as other vertices. The reason is that for each complex edge, the copied vertices have the same quadric error and position. Therefore, they have the same screen error. They are either selected into the active vertex set or vanish together. No crack will appear.

Due to the temporal coherence, in each frame, only a small portion of the active vertices changes. It is inefficient to extract the whole isosurface again. Therefore, an active triangle buffer is maintained. Prior to the isosurface extraction, each triangle in the active buffer is tested. The invalid triangles are collected for recycling. To validate the triangles, the *timestamp* field is used. Before rendering each frame, the global timestamp TS is increased by one. In the *adaptive_refinement* procedure, the *timestamp* field of each cell p is updated to TS if the following rule is satisfied:

$$(\sum_{v \subset p} \text{v.active} \oplus \text{v.active(old)})) > 0, \qquad (3)$$

where $\oplus$ is the XOR operator. This rule ensures that the *timestamp* is updated if a vertex in the cell has changed its active status. For each triangle in the active buffer, the validation function checks the maximum timestamp t_{max} of the four cells where the three vertices are created from. If t_{max} equals TS, this triangle is invalid. We don't directly use the *active* field for triangle validation since vertices may not be addressed after a CSG operation. However, the octree cells are always accessible.

4.5 Incremental Surface Extraction

The *timestamp* field can indicate two different events: CSG operations, which perform local modification on the vertex hierarchy, and view-dependent refinement of the vertex hierarchy. Therefore, we can handle them in a unified way using a truncated isosurface extraction algorithm. The full isosurface extraction is an extension of the SurfaceNets algorithm on a vertex hierarchy. The truncated algorithm has the same recursive rule as the full algorithm and saves the result into the active triangle buffer. The difference is that the truncated algorithm checks the *timestamp* field stored in each octree cell before extracting the isosurface. The criterion for performing isosurface extraction on a cell using timestamps is:

- When a cell's *timestamp* is older than TS and none of the timestamps of its 18-neighborhood cells equals TS, this cell, together with all its descendants, are skipped.

```
procedure incremental_surf_extraction(OctreeCell p, word TS)
1.  if p.timestamp < TS return;
2.  if all timestamps of p's 18-neighborhood < TS return;
3.  bool recursion=false;
4.  for each vertex v in p
5.      recursion|=v.recursion;
6.      if v.active
7.          //half of the 12 edges are used, as in SurfaceNets alg.
8.          for each edge e of the 6 edges from v's CEB
9.              for each directed sign change on e
10.                 find the 4 vertices in the 4 cells sharing e;
11.                 generate triangles;
12. if recursion or p.pRepvert==null
13.     for each child cell q in p
14.         incremental_surface_extract(q, TS);
```

Figure 10: Incremental surface extraction algorithm.

Checking the 18-neighborhood is necessary since a triangle always crosses cell boundary. By skipping cells, the incremental surface extraction procedure only takes time proportional to the count of the affected cells. Figure 10 shows our incremental surface extraction algorithm in detail. The full extraction part (steps 3-14) shares similarity with the extended dual contouring algorithm [21]. However, our algorithm works on a dynamically refined vertex octree. The vertices used for constructing triangles are usually in the intermediate levels. Based on the screen-space errors, not all the vertices in the same cell are chosen. On the contrary, their algorithm uses the vertices from the boundary leaf cells only.

In step 10 of Figure 10, we need to find the other three vertices sharing an edge **e**. Normally, we match the CEB stored in each vertex. Figure 11 shows our algorithm of finding one matched vertex for a starting vertex v. The *recursion* field is used for making decisions of traversing the current cell upward or downward.

```
function find_matched_vertex(OctreeCell p, RepVertex v, int e)
1.  OctreeCell q ⇐ p's neighboring cell in the same level;
2.  while q== null
3.      q ⇐ q's parent cell;
4.  find a vertex t in q matching v;     ///CEB matching;
5.  while t.active != true
6.      if t.recursion == true;
7.          for each child cell r of q which contains edge e
8.              finding_vertices(r, v, e);
9.      else t = t.pParentvert;
10. return t;
```

Figure 11: Algorithm for finding a matched vertex.

The incremental surface extraction algorithm can also be used to extract surfaces from adaptively sampled volumes. In such a scenario, the *finding_vertices* function may have difficulty in finding matched vertices using the CEB information. For example, in Figure 12, the small cell has one vertex v_0, while the large cell has two vertices v_1 and v_2. Both are boundary leaf cells. Since the edge **e** lies on one face of the large cell, there is no corresponding edge in the large cell. Therefore, vertex v_0 has difficulty in finding the matched vertex. The CEB matching method fails in such cases. To find the matched vertex, we use the following rules:

1. If v_0 has another edge e' with a directed sign change and e' is on the boundary edge of the large cell, use e' to find the matched vertex in the large cell. This is equivalent to using the connected surface component for matching.
2. If no such edges, find the corresponding vertex using heuristic metrics such as minimal normal deviation or minimal distance between v_0 and other vertices.

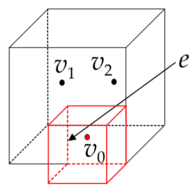


Figure 12: An example where CEB matching fails. Edge **e** is the edge that shows a directed sign change. In this configuration vertex v_0 of the small cell is unable to find the matched vertex in the large cell using its own CEB.

5 Experimental Results

We have performed tests on a 3.0GHz Pentium IV PC with 1GB memory and an on-board Intel 82865G graphics controller. The PC runs Windows XP operating system. We have used two datasets: a Lego car (Figure 1a, isovalue=120.5) and a Temple (Figure 13a, isovalue=0). Table 1 shows the statistics for the two datasets. In Figure 13a, although a large portion of the Temple is flat, the unified modeling space requires the objects to be sampled at a high resolution so that tiny details can be reconstructed correctly. In both examples, a screen-space error threshold of 2 pixels is used in LOD rendering. Due to the LOD rendering techniques, our tests runs well on the low-end graphics board.

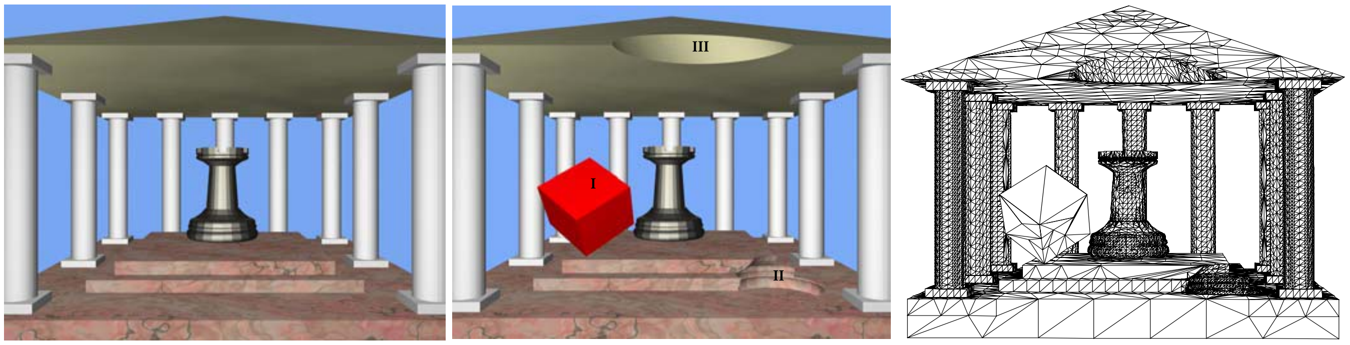


Figure 13: LOD rendering for solid modeling using the zero isosurface. (a) A view before the CSG operations. (b) Another view after three consecutive CSG operations are performed. After each CSG operation, the isopoint octree is maintained efficiently. (c) The underlying mesh rendered from a viewpoint similar to that of (b).

Table 1: Statistics for the two datasets.

Model	Resolution	# leaf cells	# rep. vertices
Lego car	132×204×110	240K	241K
Temple	512^3	2.57M	1.62M
Temple(CSG)	512^3	2.89M	1.81M

5.1 Rendering Performance

The preprocessing time for the Lego car and Temple datasets are 2.8 sec. and 18.2 sec., respectively. Figure 14 shows the random navigation timing and triangle counts of 1000 frames for the two datasets. No backface-culling is used. For the Lego car, the system maintains a buffer of about 60K triangles. For the Temple, the buffer size is around 25K triangles. In Figure 14, The updated triangle count is the sum of invalid triangles and newly created triangles in each frame. Directly rendering the full resolution on average takes 0.28 sec. and 2.22 sec., respectively. Table 2 shows the average cost of the several stages of LOD rendering in the navigation. We observe that rendering triangles takes relative constant percentage (29% and 35%) for the two datasets. For a complex model (Lego car), there are more triangles to be updated. Therefore, the surface extraction stage takes more time. On the contrary, triangle validation stage takes more time for the Temple model.

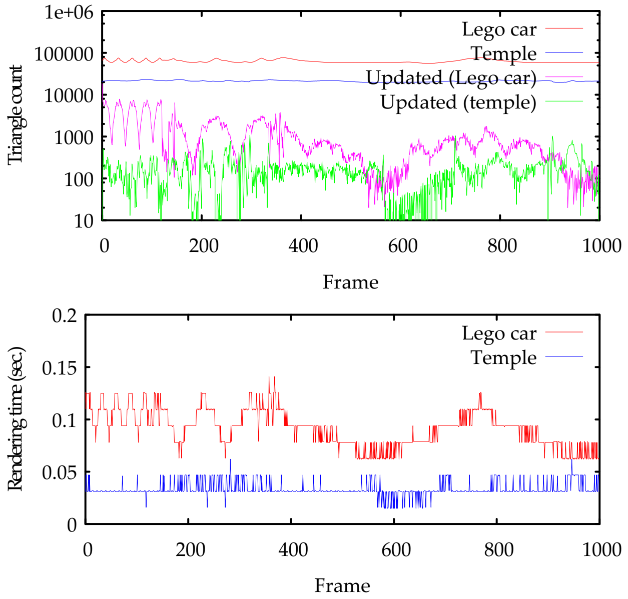


Figure 14: Triangle counts and timing results for the two datasets.

Table 2: Average percentage of cost in navigation and speedup (AR: adaptive-refinement; TV: triangle validation; SE: surface extraction; RT: rendering triangles).

Model	AR	TV	SE	RT	Time (sec.)	Speedup
Lego car	18%	17%	36%	29%	0.091	3.07
Temple	6%	42%	17%	35%	0.033	66.5

5.2 Latency

In Figure 13b, we introduce three CSG operations on the original scene. These operations are marked as I, II and III. They are non-trivial since each operation involves 57K to 71K leaf cells and 30K to 57K representative vertices. Each of these operations is computed within 1.0 second. To measure the latency of a CSG operation in our LOD rendering algorithm, we define the latency as the time between finishing a CSG operation and starting the LOD rendering procedure. It contains 3 steps: computing representative vertices, simplifying the affected octree region, and pre-selecting active vertices. Table 3 presents the time of each step. For each CSG operation, the major cost is re-building the vertex octree and the latency is within one second. The results show that our approach is efficient in rendering a dynamic vertex hierarchy.

Table 3: Latency (in sec.) between finishing a CSG operation and starting LOD rendering (CR: Computing representative vertices).

CSG op.	CR	Simplification	Selection	Total
I	0.12	0.75	0.01	0.88
II	0.09	0.54	0.01	0.64
III	0.10	0.55	0.01	0.66

5.3 Memory Consumption

The Temple dataset is challenging since an octree depth of 9 is used and the finest level cells are abundant. The modeling part almost exhausts the physical memory of the PC. The 1.81M representative vertices will form about 3.62M triangles, if fully extracted. In our experiments, the total vertices in the vertex hierarchy is about $1.3n$, where n is is the count of vertices in the finest level. The extra cost for LOD rendering takes about 78M bytes for the Temple dataset.

We also compare the memory consumption of our algorithm with the VDPM [9], SVDPM and HDS [15] algorithms in theory. Table 4 gives the space requirements of the four algorithms without considering the octree cost. We discard the ROAMing [2] algorithm since it is dedicated to terrain rendering. In VDPM, one drawback is that all its data structures scale proportionally with the size n of the fully refined mesh M^n. In SVDPM, Hoppe [10] redesigned the vertex data structure into two parts: a static part encoding the vertex hierarchy and refinement dependencies (size $88n$ bytes), and a dynamic part encoding the connectivity of the active mesh M (size $112m$ bytes), where m is the count of active triangles. The space requirement of the HDS algorithm is not reported. We estimate it in the tight octree case using the assumption of 4 triangles, 1 subtriangle and 4 children per cell. View-dependent error metrics are not included. Our algorithm gains 456%, 118% and 544% space saving over VDPM, SVDPM, and HDS, respectively.

Table 4: Memory consumption for several LOD rendering algorithms (n: # vertices in the finest mesh, m: # active triangles; g: # morphing vertices, usually $n >> m > g$).

Algorithm	Space (bytes)	Percentage
SHIC (ours)	$31n * 1.3 + 42m$	100%
VDPM	$224n$	556%
SVDPM	$88n + 112m + 52g$	218%
HDS	$228n * 1.3 + 32m$	644%

5.4 Limitations

Our framework suffers the common problems encountered by other grid based isosurface simplification algorithms. The CEB has limited encoding ability for handling too complex topology. Therefore, it has a larger lower bound of top vertices in the vertex hierarchy. Our method can't handle cases where the isosurface intersects cube faces only. Therefore, we prevent these cases from happening in CEB merging. Furthermore, the simplified mesh quality is not as good as gridless isosurfacing and optimization algorithms [16, 23], since the vertex positioning scheme in our framework hasn't considered the mesh quality issue.

6 Conclusions and Future Work

We have presented SHIC, an isopoint clustering framework for rendering large isosurfaces. Different from surface-based and point-based clustering, our framework is connectivity guided. Our contributions are: a connectivity encoding scheme using edge bitmaps, topology-preserving hierarchical and selective clustering to handle complex edges, the usage of a timestamp field to unify vertex tree modification and view-dependent vertex refinement events, and an incremental surface extraction algorithm from the isopoint hierarchy. Our experiments show that SHIC is efficient, memory-saving and suitable for rendering dynamic isosurface objects. SHIC is also easy to implement.

Currently, our framework only works for a fixed isovalue. We plan to extend our algorithms for a range of values using an isovalue spanning tree. For some isovalue ranges, the surface topology in a cell won't change, only the representative vertex positions change. We can construct the same vertex tree and fill the vertex positions dynamically. It is also possible to extend our framework for out-of-core isosurface visualization since our method integrates space subdivision and connectivity encoding.

Acknowledgments
This work is supported by NSF grant CCR0306438 and ONR grant N000140110034.

References

[1] C. S. Co, B. Heckel, H. Hagen, B. Hamann, and K. I. Joy. Hierarchical clustering for unstructured volumetric scalar fields. In *IEEE Visualization*, pages 325–332, October 2003.

[2] M. A. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich, and M. B. Mineev-Weinstein. ROAMing terrain: real-time optimally adapting meshes. In *IEEE Visualization*, pages 81–88, October 1997.

[3] M. Garland and P. S. Heckbert. Surface simplification using quadric error metrics. In *SIGGRAPH Proceedings*, pages 209–216, August 1997.

[4] T. Gerstner and R. Pajarola. Topology preserving and controlled topology simplifying multiresolution isosurface extraction. In *IEEE Visualization*, pages 259–266, October 2000.

[5] B. Gregorski, M. Duchaineau, P. Lindstrom, and V. Pascucci. Interactive view-dependent rendering of large isosurfaces. In *IEEE Visualization*, pages 475–482, October 2002.

[6] T. He, L. Hong, A. Kaufman, A. Varshney, and S. Wang. Voxel based object simplification. In *IEEE Visualization*, pages 296–303, October 1995.

[7] M. Hopf and T. Ertl. Hierarchical splatting of scattered data. In *IEEE Visualization*, pages 433–440, October 2003.

[8] H. Hoppe. Progressive meshes. In *SIGGRAPH Proceedings*, pages 99–108, August 1996.

[9] H. Hoppe. View-dependent refinement of progressive meshes. In *SIGGRAPH Proceedings*, pages 189–198, August 1997.

[10] H. Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *IEEE Visualization*, pages 35–42, October 1998.

[11] T. Ju, F. Losasso, S. Schaefer, and J. Warren. Dual contouring of Hermite data. In *SIGGRAPH Proceedings*, pages 339–346, July 2002.

[12] L. P. Kobbelt, M. Botsch, U. Schwanecke, and H. Seidel. Feature-sensitive surface extraction from volume data. In *SIGGRAPH Proceedings*, pages 57–66, August 2001.

[13] P. Lindstrom and V. Pascucci. Visualization of large terrains made easy. In *IEEE Visualization*, pages 363–370, October 2001.

[14] W. E. Lorensen and H. E. Cline. Marching Cubes: A high resolution 3D surface construction algorithm. In *SIGGRAPH Proceedings*, pages 163–169, July 1987.

[15] D. Luebke and C. Erikson. View-dependent simplification of arbitrary polygonal environments. In *SIGGRAPH Proceedings*, pages 199–208, August 1997.

[16] Y. Ohtake and A. Belyaev. Dual/primal mesh optimization for polygonized implicit surfaces. In *ACM Solid Modeling*, pages 171–178, June 2002.

[17] V. Pascucci and C. L. Bajaj. Time critical isosurface refinement and smoothing. In *Volume Visualization and Graphics Symposium*, pages 22–42, October 2000.

[18] M. Pauly, M. Gross, and L. Kobbelt. Efficient simplification of point-sampled surfaces. In *IEEE Visualization*, pages 163–170, October 2002.

[19] R. N. Perry and S. F. Frisken. Kizamu: A system for sculpting digital characters. In *SIGGRAPH Proceedings*, pages 47–56, August 2001.

[20] R. Shekhar, E. Fayyad, R. Yagel, and J. Cornhill. Octree-based decimation of marching cubes surfaces. In *IEEE Visualization*, pages 335–342, October 1996.

[21] G. Varadhan, S. Krishnan, Y. J. Kim, and D. Manocha. Feature-sensitive subdivision and isosurface reconstruction. In *IEEE Visualization*, pages 99–106, October 2003.

[22] R. Westermann, L. Kobbelt, and T. Ertl. Real-time exploration of regular volume data by adaptive reconstruction of isosurfaces. *The Visual Computer*, 15(2):100–111, 1999.

[23] Z. J. Wood, P. Schröder, D. Breen, and M. Desbrun. Semi-regular mesh extraction from volumes. In *IEEE Visualization*, pages 275–282, October 2000.

[24] J. C. Xia and A. Varshney. Dynamic view-dependent simplifcation for polygonal models. In *IEEE Visualization*, pages 327–334, October 1996.

[25] N. Zhang, W. Hong, and A. Kaufman. Dual contouring with topology-preserving simplification using enhanced cell representation. In *IEEE Visualization*, 2004.