NON-REPLICATING INDEXING FOR OUT-OF-CORE PROCESSING OF SEMI-REGULAR TRIANGULAR SURFACE MESHES

Igor Guskov

University of Michigan, Ann Arbor
guskov@eecs.umich.edu

ABSTRACT

We introduce an indexing scheme for the vertices of semi-regular meshes, based on interleaving quadtrees rooted on the edges of the base mesh. Using this indexing scheme we develop an out-of-core data structure for semi-regular mesh processing. Our approach targets applications that process vertex data in a coarse-to-fine manner performing several passes through each level of the hierarchy. We consider several measures of layout quality that provide lower bounds on the size of in-core memory buffer required for valid referencing of vertex data during an atomic step of mesh processing. The approach is tested on an in-place implementation of the Loop subdivision scheme for large control meshes.

Keywords: external data structures, semi-regular surface meshes, subdivision

1. INTRODUCTION

Mesh-based representations of surfaces are commonplace in the practice of geometric modeling and visualization. While the originally acquired surface meshes often come in irregular (unstructured) form, the efficiency requirements of further modeling and archival may require more structured representations such as semi-regular meshes. Semi-regular meshes represent geometry as regular refinements of an irregular base mesh (see Figure 1 for an example), and combine the generality of irregular meshes with the efficiency of regular ones. Recent progress in subdivision scheme analysis and remeshing approaches has made the semi-regular meshes usage more frequent. This paper addresses the issue of external data structures design for such meshes. In particular, we present an indexing scheme for the vertices of semi-regular meshes that does not create any vertex duplication for closed surface meshes. Using this indexing scheme we develop out-of-core data structures that allow efficient processing of very large semi-regular meshes.

Recently, there has been a lot of interest in OOC processing for computer graphics and visualization purposes due to the necessity of processing large datasets. The developed methods cover both regular (structured) [1] and irregular (unstructured) shape data [2] [3] [4]. However, we are not aware of any work targeted specifically at semi-regular meshes. Therefore, our goal is to develop a library that can be used to process large semi-regular surface meshes. Rather than striving for utmost generality, our approach is to specify a basic coarse-to-fine data access pattern, and specialize the design of our data structures towards that pattern.

One of the objectives of our work is to create a library that can be used instead of the usual in-core semi-regular mesh data structures in existing algorithms. In fact, we would like to minimize the changes to the currently existing code, and to substitute our OOC data structures in a way mostly transparent to the existing algorithm.

We assume that the computation conforms to the following template

```
for(k=0; k<=max_level; ++k) {
    several passes using levels 0..k
}
```

Note that a typical recursive implementation of subdivision

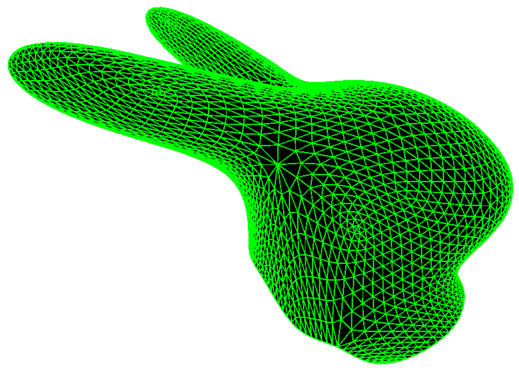


Figure 1: An example of a simple semi-regular mesh obtained by Loop subdivision. Note a number of base vertices of valence not equal to six preserved in the refined mesh. The mesh consists of a number of patches refined in a regular one-to-four fashion.

algorithms [5][6] and several existing remeshing algorithms [7][8] can be easily represented in this fashion.

Processing of semi-regular meshes requires attention both to the irregular base mesh and to the regular patch structure. In particular, the global index of a vertex in a semi-regular mesh consists of the index of the primitive (in our case a base edge) coming from the layout of the base mesh *and* the local index within a regular data structure (in our case a uniformly subdivided quadtree) associated with each of the base primitives. The following sections will deal with both of these indexing problems: first, we introduce the vertex indexing within regular patches used in our library in Section 2. Section 3 considers the issues related to the base mesh layout using notations similar to the very recent *Streaming Meshes* work of Isenburg and Lindstrom [4]. The data structures themselves are described in Section 4, followed by their application to in-place Loop subdivision in Section 5.

Our implementation operates under assumption that the base mesh is stored in internal memory, while the semi-regular information is stored externally. It should be possible to generalize our approach to bigger base meshes stored externally.

1.1 Related work

Isenburg et al. [3] classify OOC mesh processing approaches into three categories: mesh cutting, batch and online processing. Our approach to semi-regular mesh storage belongs to the online processing class, which maintains an internal memory buffer for a part of the mesh. Note that we are only concerned with the storage of the regular patches and assume that the base mesh connectivity is fixed and accessible via the usual internal memory mesh data structures. Thus, the maintenance of the proper mesh connectivity is not an issue.

Recently, the concept of *streaming meshes* was presented by Isenburg and Lindstrom [4]. Streaming meshes advocate the use of reordered mesh format that explicitly introduces and finalizes vertices. When each processing operation is able to proceed on a per-triangle basis, the measures of layout span and width from [4] provide sufficient characterization of the mesh ordering. Mesh rendering is one example of such per-triangle processing. However, not every operation is possible on a per-triangle basis. Section 3 describes several measures that are specialized to our setting.

The idea of edge-based indexing using interleaved quadtrees comes from the work on semi-regular mesh compression [9]. A different kind of interleaved quadtrees was also used in the work on terrain simplification [10].

There exist much work on improving performance of subdivision algorithms in the context of memory hierarchy, fast evaluation of surfaces, and hardware rendering [11][12] [13]. Generally, these approaches advocate to compute all the data within a single regular patch in a single pass, in order to minimize data transfer. Rather than following this path, we took a generic implementation of subdivision scheme and tried to design an external data structure that will not require rewriting of the application code. Thus, the simplicity of the original code is retained. However, the good performance is only achieved in the case when a proper (streaming) mesh layout is chosen for the base mesh.

2. SEMI-REGULAR MESH INDEXING

The vertices of a semi-regular mesh typically store position and normal information. If the finest level of the hierarchy is specified beforehand, it is convenient and more efficient to represent the vertices within each patch as a regular array. For primal semi-regular meshes such as the ones produced by Loop and Butterfly subdivision schemes [6], the patch boundaries are shared by adjacent patches and explicit duplication of assignments and synchronization of values is required, which can be inconvenient. Moreover, when the desired number of levels is small, the percentage of boundary vertices within a regular triangular patch is relatively large (see the table below).

Levels	Total number of vertices	Number of boundary verts	Boundary percentage
0	3	3	100%
1	6	6	100%
2	15	12	80%
3	45	24	53.3%
4	153	48	31.4%

The percentage of duplicated vertices can be approximately estimated as half of the boundary vertex percentage. Thus, on level four, about 15% of all the vertices are duplicates. This can be inefficient, especially if only few levels of refinement are needed over a large base mesh. In what follows, we propose an indexing scheme that does not create any duplication (for closed surface meshes without boundary). We start by introducing some notation and proceed to describe two indexing schemes: the usual face-based indexing and the non-replicating edge-based indexing.

2.1 Notation

We consider a base manifold mesh $\mathcal{M} = (\mathcal{V}, \mathcal{E}, \mathcal{F})$ where $\mathcal{V}$ is the set of base vertices, $\mathcal{E}$ is the set of base edges, and $\mathcal{F}$ is the set of triangular base faces. The connectivity of a primal semi-regular mesh is built by applying one-to-four splits recursively as in Loop subdivision scheme [5]. We shall denote as $\mathcal{V}_L$ the set of all the vertices of the mesh at level L, so that $\mathcal{V}_0 = \mathcal{V}$. A natural way of indexing the set $\mathcal{V}_L$ is to specify a base triangle $f \in \mathcal{F}$ and an index tuple $i = (i_1, i_2)$ within a regular triangular slab of level L. We have $i_1 \geq 0$, $i_2 \geq 0$, and $i_1 + i_2 \leq n(L) = 2^L$. We assume that for a base face $f = (v_a, v_b, v_c)$ the base vertex v_a has indices $(0, 0)$, the base vertex v_b has indices $(2^L, 0)$, and the base vertex v_c has indices $(0, 2^L)$. A vertex lying on a non-boundary edge of the base mesh belongs to two triangular patches, and has two face-indices corresponding to it.

This face-based indexing is very simple and makes it very easy to find neighboring vertices by simple increments and decrements of integer indices. Hence, many implementations of subdivision and remeshing algorithms naturally use this face-based indexing.

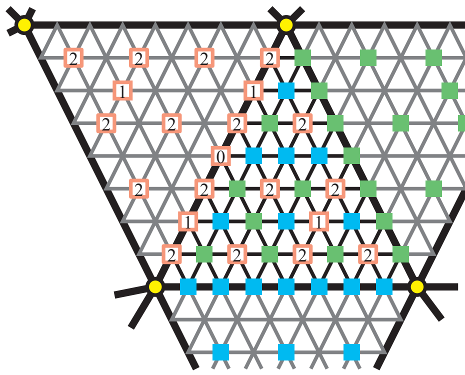


Figure 2: Edge-based indexing: all the newly appeared vertices within the center patch after three levels of refinement are covered by three interleaving vertex quadtrees associated to the edges of the base mesh. Vertices of the pink quadtree are labeled with indices corresponding to their quadtree level.

2.2 Non-replicating edge-based indexing

An alternative way of covering all the vertices in a semi-regular hierarchy is suggested by the location of wavelet coefficients used for compression of semi-regular meshes [9]. Each "band" of wavelet coefficients makes up a quad-tree rooted at the middle of a base edge. It is easy to see that each non-base vertex in a semi-regular hierarchy belongs to one of these quadtrees. For a closed mesh, there is one-to-one correspondence, and no index space is wasted. For meshes with boundaries, the quadtrees corresponding to boundary edges stick out, and waste half of their index space. In this paper, we do not consider the issue of boundary quadtrees. Figure 2 shows an example of covering a patch by three interleaved quadtrees. One of the quadtrees is also marked with the level indices. Note that this indexing scheme does not replicate any vertex, and that the base vertices have to be treated separately.

One disadvantage of the edge-based indexing is the non-intuitiveness of the neighbor indexing – each vertex is surrounded by vertices from several other quadtrees. Our purpose is to create an out-of-core representation for semi-regular meshes that internally uses edge-based non-replicating indexing but can be substituted instead of in-core face-based data structures. Thus, we need a transformation from a face-based index to an edge-based index. The next subsection will introduce the specifics of such index mapping. The description of our out-of-core data structure management will follow in Section 4.

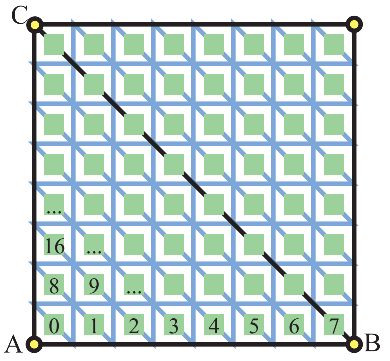


Figure 3: Vertex indexing of the third level associated with the edge BC. The vertices are in one to one correspondence with diagonally oriented edges of the previous level.

2.3 Transforming face-based index into edge-based form

Given a vertex of the semi-regular mesh located on the level L with the index pair (i_1, i_2) within a face $f = (v_a, v_b, v_c)$, we would like to find its indices in the edge-based quadtree representation. This can be done in a sequence of steps:

- Find the level on which the vertex first appeared. This can be done by performing a binary logarithm-like operation on the index pair (i_1, i_2) to find the position of the first non-zero bit in either of the indices i_1 or i_2. This would give the difference between the current level L and the level l where the vertex got introduced. We define the corresponding index pair on level l as (k_1, k_2) where $k_d = i_d 2^{l-L}$ for $d = 1, 2$.
- By construction, at least one of the indices k_d has non-zero least significant bit. Define b_d to be the least significant bit of k_d for $d = 1, 2$. It is clear that $b_1 \vee b_2 = 1$.

- Based on the pair of bits (b_1, b_2) we can find the edge of the face whose quadtree contains the vertex. The edge (v_b, v_c) corresponds to the pair $(1, 1)$ as shown in Figure 3. The edge (v_a, v_b) corresponds to $(1, 0)$, (v_c, v_a) to $(0, 1)$.
- For the edge (v_b, v_c) the vertex indices within the square grid of the level l are given as $((k_1-1)/2, (k_2-1)/2)$. For the other two quadtrees a proper rotation of the face can be performed to find the indices.

The above sequence of operations produces the base edge, the quadtree level, and the tuple of indices with the square grid on that quadtree level. Thus, the edge-based vertex index is fully specified for any non-base vertex from $\mathcal{V}_L$. We shall now turn to more precise characterization of base mesh layout.

3. STREAMING MESH CONSIDERATIONS

Streaming meshes work of Isenburg and Lindstrom [4] introduced several measures characterizing mesh layouts that are relevant for applications performing per-triangle processing of irregular meshes such as rendering. In their original form, these measures do not characterize base mesh layouts in terms of edge indexing and wider stencils that are used in our approach. In this section we introduce mesh layout characterizations that will be immediately useful for our application.

As an example, consider irregular mesh smoothing. It is usually implemented in a way that sequentially processes every mesh vertex, and for each vertex, the one-ring of neighboring vertices is collected and averaged. Hence, the differences of the vertex indices within one-ring should be of more interest than the differences within a single triangle. Generally, many mesh processing algorithms are local, and proceed in a number of passes each of which sequentially modifies mesh vertices based on their neighborhoods. We use the word *stencil* for the set of neighbors used in a particular local operation. Consider a collection Ω of such stencils. Each stencil $\omega \in \Omega$ is a subset of the vertex set ($\omega \subset \mathcal{V}$). For instance, Ω could be the set of all the vertex triples corresponding to mesh faces $\Omega = \mathcal{F}$, this stencil collection is considered in [4].

Let i be an indexing of vertex set $i : \mathcal{V} \rightarrow \mathbf{N}$. We define the span of a stencil collection Ω with respect to the indexing i as follows:

$$span[i](\Omega) \overset{def}{=} \max\{|i(u) - i(v)| : u \in \omega, v \in \omega, \omega \in \Omega\}.$$

Given an algorithm we can consider the corresponding collection of stencils and find its span σ with respect to some particular mesh indexing. We can expect that the internal buffer size of at least σ vertices is required for the algorithm to proceed in a streaming fashion.

The above considerations will hold if the application stores its data with vertices. In our setting, the semi-regular patch data will be saved with base edges, hence the corresponding spans and indexing needs to be redefined for edge stencils. Let j be an indexing of the edge set of a mesh $j : \mathcal{E} \rightarrow \mathbf{N}$, and let A be a collection of edge stencils (each edge stencil α is a subset of $\mathcal{E}$). Then, the span of A with respect to j can be defined in the same way as above. For applications that perform per-triangle processing, the edge span of the *face* collection is important, that is, $\sigma_f = span[j](A_f)$, where each stencil $a \in A_f$ consists of three edges forming a triangle.

Mesh indexing that has low span values can be processed efficiently in a streaming fashion as described in [4]. As mentioned in [4], finding optimal mesh layout is an NP-hard problem, and several approximate algorithms are used in practice. Possibly, the simplest mesh indexing is given by arranging the mesh faces to be sorted along the longest spatial axis. Once the faces are sorted, we index the edges in a compatible way, by introducing them in the order they are used in the faces, similar to the compaction operation from [4]. We use this spatial sorting strategy for all the results reported in this paper.

We shall now proceed to describe our implementation of OOC data structures.

4. SEMI-REGULAR OOC MESH LIBRARY

4.1 Memory mapping

Our goal is to enable out-of-core processing of large semi-regular meshes with minimal changes to the existing algorithms. Taking into account the streaming considerations of the previous section, we assume that the algorithms using our library will adhere to the following outline:

```
for(k=0; k<=max_level; ++k) {
    activate level k
    several passes using levels 0..k
}
```

Each pass in the above processing needs to proceed in the order optimized for good streaming performance. A basic operation in each pass consists of a sequential processing of stencils. For instance, in a subdivision algorithm, for each face of the base mesh, we loop through all the vertices of the current level that belong to the corresponding patch of the surface, and for each such vertex we perform the following operation:

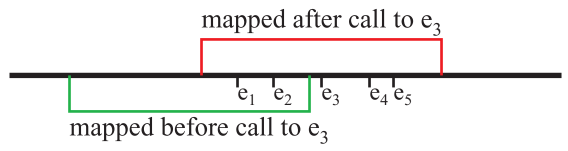
```
patch->at(level,i,j) =
   Compute(patch->at(level-1, i1, j1),
           patch->at(level-1, i2, j2),
           ...)
```

Our library overloads the `at(l,i,j)` function that returns a reference to the vertex data. Thus, a call to `at()` function may have a side effect of performing management of the OOC data structures. In particular, a portion of internally stored data can be written to the disk and another portion of externally stored data can be loaded. We would still like to retain the original description of the algorithm without modifications. In particular, all the pointers to vertex data passed to the `Compute()` function above should be valid. This is achieved as follows: the internal data are maintained in a circular memory buffer that is big enough to cover twice the span of stencils of the `Compute()` function with respect to particular base edge indexing.

Our `at()` function has the following form:

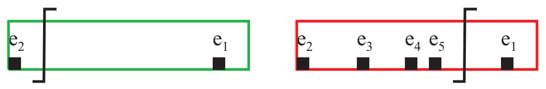
```
function at(level, i, j) {
   if(not mapped(this->edge_index))
      map(this->edge_index)
   ...
   return vertex data;
}
```

The `map` function reconfigures the active buffer mapping in such a way that the currently accessed patch is in the middle of the memory-mapped segment. In order to overcome index range restrictions of 32-bit operating systems, we do not use their memory mapping functionality, rather performing explicit file reading and writing. Each mapping operation operates on two possibly overlapping index space segments, and the use of circular buffer allows to keep all the data in the overlapping region untouched. Since all the pointers within the same stencil are guaranteed to be inside the newly mapped region, all the previously found pointers to data remain valid as illustrated in the figure below:



The patches corresponding to two edges e_1 and e_2 were memory mapped in the old (green) buffer segment that did not include e_3. Upon the call to e_3 data, the new (red) segment is mapped to the memory that contains all the edges $e_1, \ldots, e_5$ that participate in the currently processed stencil.

The following illustration shows the state of the circular buffer before and after re-buffering caused by the access to e_3 – we see that after the access all the five elements of the stencil are mapped to the memory buffer, and that the previously mapped e_1 and e_2 retain their memory location:



Note that the data access operation will remain correct even if the stencils are re-indexed (as long as the edge indexing stays the same). However, a bad ordering of the stencils may affect the performance by excessive loading and unloading of file segments.

4.2 Per-level storage of patch information

The basic pattern of usage presented at the beginning of the preceding section implies that the data are processed in a coarse-to-fine manner. In particular, this means that during algorithm passes on coarser levels the finer level data are not required. Hence, we separate the quadtree levels of each edge into different files; that is, each particular level gets mapped to an individual file, and within each file all the regular quadrilateral grids for all the edges are stored sequentially. Thus, if each vertex need S bytes of storage, the overall size of the file at level L is given by $S\,|\mathcal{E}|\,4^{L-1}$. Note that the level L of the semi-regular mesh corresponds to the level $L-1$ of the quadtree (in particular, the base mesh at level 0 does not have any out-of-core storage).

The total size of all the memory buffers is specified as a parameter during the initialization: the size of the memory buffer for each level is computed so that the range of edge indices covered by each buffer is approximately the same: hence each consecutive level has the buffer four times bigger than the buffer of the previous coarser level.

5. AN APPLICATION: LOOP SUBDIVISION

5.1 In-place Loop subdivision implementation

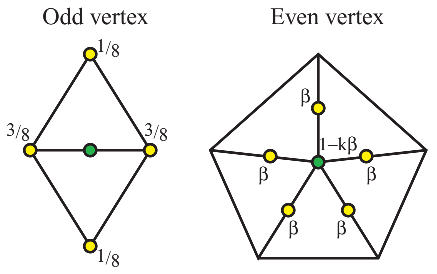
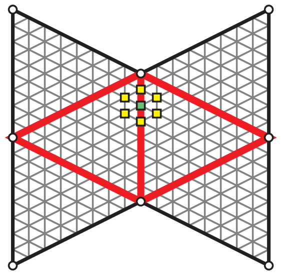


Figure 4: The in-place Loop subdivision scheme stencils. The coefficient β in the even vertex computation is given by $\beta = \frac{8}{5}\beta'$, where β' is the corresponding coefficient from the usual (not in-place) implementation of Loop subdivision scheme [6].

As an example application of our OOC semi-regular mesh library we used the in-place implementation of the Loop subdivision scheme [5] [6]. The scheme proceeds in two passes: the first pass computes odd vertices, and the second pass updates the even vertices using the odd vertices computed in the previous pass. Figure 4 shows the stencils used in the computation. The in-place implementation has the advantage of not requiring additional memory for the storage of the coarser level, and is similar to the dual-primal implementations of quadrilateral subdivision schemes [14]. The normals are computed in the same way as for the usual Loop scheme. There exist faster ways of computing subdivided values including precomputed tables and per-patch computation specifically useful for faster rendering. We chose an existing in-place implementation of the Loop scheme as a typical example of recursive multiresolution algorithm proceeding in a coarse to fine fashion. The only modification to the existing in-core algorithm was to explicitly activate the levels of the hierarchy before the operation of each level.

5.2 Stencil span

In order to obtain the lower bound on the size of the memory buffer necessary for the correct operation of Loop subdivision, we need to consider all the stencils used in the computations. For instance, consider an even vertex update shown below:



The set of yellow and green vertices belong to the regular quadtrees associated with five base edges shown in bold red. Thus, if the even vertex update is computed in a single expression and all the data from the corresponding patches is required to be in-core at the same time, we need to compute the corresponding edge indexing span for the collection of stencils of this form. That is, for every base edge e, define the stencil $\alpha(e)$ to be the set of five base edges consisting of e and the other four edges that form two triangular base faces adjacent to e. Consider the collection A of all such stencils: $A \overset{def}{=} \{\alpha(e) : e \in \mathcal{E}\}$. Then, as discussed in the previous section, the memory buffer should be of such a size as to be able to contain more than $2\sigma_A$ edge quadtrees, where $\sigma_A = span[i_e](A)$, and $i_e : \mathcal{E} \to \mathbf{N}$ is the edge indexing used for a particular mesh model.

Given the edge span σ_f of the face collection A_f introduced above we can find an upper bound on σ_A; in fact it is easy to see that

$$\sigma_A \leq 2\sigma_f.$$

However, our experiments show that this estimate is too conservative in the case of layouts built by spatial sorting; and by computing σ_A directly for a given mesh layout, we get values lower than $2\sigma_f$ (see the following section for examples). For improved performance, the layout optimization methods described in [15] may be applied directly to stencil collection A.

The update of the coordinates of an extraordinary vertex can result in even wider stencil collection that would include the subsets of edges adjacent to each base vertex. In our implementation, this update is not computed in a single step, hence we do not consider the edge span of this stencil collection.

5.3 Examples

We evaluated the performance of our approach by running our in-place Loop subdivision implementation on three large models: a molecular surface (*Deo*), a scanned David statue (*David*), and the happy Buddha model (*Buddha*). The following table lists the characteristics of these base meshes. The stencil and face edge spans are given for the spatially sorted models.

Model	# Vertices	# Edges	σ_f	σ_A
Deo	26,016	78,084	4,864	4,983
David	274,831	824,517	17,764	24,857
Buddha	545,714	1,637,334	38,765	56,176

Figure 5: Subdivided molecular and Buddha models.

Figure 6 shows the diagram visualizing the stencil span for the Buddha model before and after spatial sort. Each stencil takes one horizontal line and the edges used in that stencil are depicted along each line at a location given by their index. The horizontal width of the black region corresponds to the edge span of the stencil collection. It easy to see that spatial order improves the streaming characteristics of the indexing.

Figure 7 shows the comparison of the diagrams for the collection of faces A_f and the collection of Loop stencils A. While the latter seem to be wider, the numbers from the above table show that the maximal spans of these two collections are similar.

The table below shows the timing results for the subdivision performance of our algorithm for the three models: *Deo* and

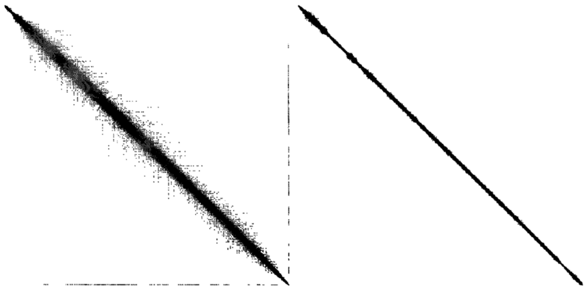


Figure 6: Buddha model span diagram of stencil collection $\{\alpha(e) : e \in \mathcal{E}\}$ before (left) and after (right) spatial sort. Horizontal axis corresponds to edge index, vertical axis corresponds to the stencil index.

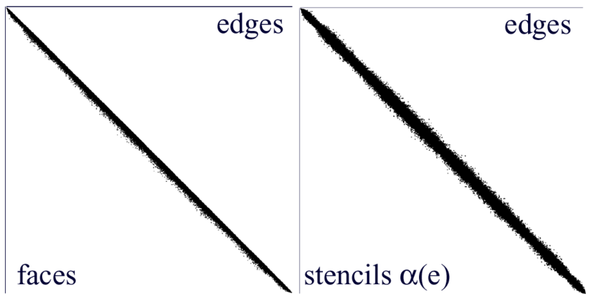


Figure 7: Molecule model span diagram. Left: the face collection, right: Loop even-stencil collection.

David models subdivided to level 5, and *Buddha* model subdivided to level 4. We have not modified the original recursive implementation of the Loop subdivision; although various optimizations are available in the computer graphics literature [11][12] [13] . In particular, our generic recursive implementation performs several passes over the mesh for the subdivision of a single level and the normal computation. The results are reported for a 3GHz Pentium 4 PC with 1GB of RAM. The memory used in each run of the algorithm is also reported – this quantity was adjusted by specifying the size of the memory buffer available for internal storage of patches. The size of the circular buffer allocated for each level was varied as described in Section 4.2. The sizes of the external data structures on disk are also included in the table.

Model (# levels / verts on finest level)	Memory used	Time elapsed (m:s)	CPU time (m:s)	Size on disk
Deo (5 / 26M)	180MB 640MB	8:27 5:26	2:00 2:00	1GB 1GB
David (5 / 281M)	913MB	115:24	21:20	10GB
Buddha (4 / 139M)	530MB	71:41	11:39	5GB

6. CONCLUSIONS AND FUTURE WORK

We have introduced an out-of-core semi-regular mesh data structures and evaluated its performance for an approximating subdivision scheme. Minimal modification of the existing subdivision code was required to be able to use our external memory data structures.

We would like to apply our library in a remeshing procedure that would produce an approximation of a fine original mesh. We expect that while the semi-regular representation used here can be applied without modification, the processing of the original mesh would require the approach possibly similar to processing sequences work of Isenburg et al. [3].

Our implementation works for uniform meshes. Complex real-world models often require adaptive refinement in areas of high curvature. We note that the proposed edge-based indexing of the regular patches should work well for a block-based refinement of semi-regular meshes similar to the work on efficient representation of adaptively sampled distance fields [16].

Acknowledgments This work was supported in part by NSF (CCR-0133554). Models are courtesy of the Scripps Research Institute and the Digital Michelangelo project at Stanford University [17].

References

[1] Pascucci V., Frank R.J. "Global Static Indexing for Real-time Exploration of Very Large Regular Grids." *Super Computing 2001*. 2001

[2] Cignoni P., C.Montani, C.Rocchini, R.Scopigno. "External memory management and simplification of huge meshes." *IEEE TVCG*, 2003

[3] Isenburg M., Lindstrom P., Gumhold S., Snoeyink J. "Large Mesh Simplification using Processing Sequences." *IEEE Visualization 2003*. 2003

[4] Isenburg M., Lindstrom P. "Streaming meshes." *submitted*. 2004

[5] Loop C.T. *Smooth subdivision surfaces based on triangles*. Master's thesis, Department of Mathematics, University of Utah, 1987

[6] Zorin D., Schröder P., editors. *Subdivision for Modeling and Animation*. Course Notes. ACM SIGGRAPH, 1999

[7] Guskov I., Vidimče K., Sweldens W., Schröder P. "Normal Meshes." *Proceedings of SIGGRAPH*, pp. 95–102, 2000

[8] Lee A.W.F., Sweldens W., Schröder P., Cowsar L., Dobkin D. "MAPS: Multiresolution Adaptive Parameterization of Surfaces." *Proceedings of SIGGRAPH*, pp. 95–104, 1998

[9] Khodakovsky A., Schröder P., Sweldens W. “Progressive Geometry Compression.” *Proceedings of SIGGRAPH*, pp. 271–278, 2000

[10] Lindstrom P., Pascucci V. “Terrain Simplification Simplified: A General Framework for View-Dependent Out-of-Core Visualization.” *IEEE TVCG*, vol. 8, no. 3, 239–254, 2002

[11] Pulli K., Segal M. “Fast rendering of subdivision surfaces.” *Proceedings of the eurographics workshop on Rendering techniques ’96*, pp. 61–70. 1996

[12] Bischoff S., Kobbelt L.P., Seidel H.P. “Towards hardware implementation of loop subdivision.” *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pp. 41–50. 2000

[13] Bolz J., Schröder P. “Rapid Evaluation of Catmull-Clark Subdivision Surfaces.” Submitted

[14] Zorin D., Schröder P. “A Unified Framework for Primal/Dual Quadrilateral Subdivision Schemes.” *CAGD*, vol. 18, no. 5, 429–454, 2001

[15] Diaz J., Petit J., Serna M. “A survey of graph layout problems.” *ACM Comput. Surv.*, vol. 34, no. 3, 313–356, 2002

[16] Perry R.N., Frisken S.F. “Kizamu: a system for sculpting digital characters.” *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pp. 47–56. ACM Press, 2001

[17] Levoy M. “The Digital Michelangelo Project.” *Proceedings of the 2nd International Conference on 3D Digital Imaging and Modeling*. Ottawa, October 1999