Approximate Subdivision Surface Evaluation in the Language of Linear Algebra

Michael Driscoll, et al.
Computer Science Division, UC Berkeley
{driscoll,etal}@cs.berkeley.edu

Abstract

We present an interpretation of approximate subdivision surface evaluation in the language of linear algebra. Specifically, vertices in the refined mesh can be computed by left-multiplying the vector of control vertices by a sparse matrix we call the *subdivision operator*. This interpretation is rather general: it applies to any level of subdivision, it holds for many common subdivision schemes (including Catmull-Clark and Loop), it can be extended to support hierarchical edit operations, and it subsumes sharpness and feature-adaptive schemes. Furthermore, our interpretation encourages high-performance implementations built on numerical linear algebra libraries. It is most applicable to subdivision of static control meshes undergoing deformation, i.e. animation, in which case it allows users to trade-off time-to-first-frame and framerate. We implemented our strategy as an extension to Pixar's production subdivision code and observed speedups of 2x to 14x using both multicore CPUs and GPUs.

CR Categories: I.3.5 [Computer Graphics]: Picture/Image Generation—*Display algorithms*; I.3.7 [Computer Graphics]: Computational Geometry and Object Modelling—*Curve, surface, solid, and object representation*

Keywords: subdivision surfaces, sparse matrix-vector multiplication

Links: DL PDF

1 Introduction

Subdivision surfaces are among the most popular primitives in modeling and animation applications. Coupled with the steady advance of computer performance, it has recently become feasible to evaluate subdivision surfaces in real-time, an especially important feature for animators interacting with objects. However, high-performance implementations must target multicore CPUs and GPUs to remain competitive. This places a significant burden the programmers, who must now navigate the minefield of race conditions, load imbalance, and deadlock that is parallel programming. The task is further complicated by the variety of parallel platforms available, many of which must be tuned to attain maximum performance. The so-called "portable performance" challenge is still an open problem within the research community and beyond.

One way to achieve portable performance for a collecting of application and architectures is to identify a set of common computational patterns exhibited by the applications, and implement each pattern once per architecture. Along this line, researchers at UC Berkeley have identified thirteen 'motifs' that represent common computational patterns that can benefit from parallel implementations [Asanovic et al. 2006]. Here, we cast the problem of subdivision surface evaluation in the language of the Sparse Linear Algebra motif. This interpretation allows us to draw on a vast body of knowledge and resources for manipulating sparse systems. Our interpretation leads to implementations that contain no explicit parallel code but match or exceed the performance of expertly-written code.

The contributions of this paper are:

- An interpretation of subdivision surface evaluation in the language of linear algebra.
- A novel surface evaluation algorithm built on high-performance numerical linear algebra libraries.
- Performance results showing performance comparable to existing methods on GPUs, and integral factor speedups on CPUs.
- A discussion of algorithmic extensions for handling semi-sharp creases, hierarchical edits, and feature-adaptive subdivision.

This paper is organized as follows: Section 2 presents our novel interpretation of the problem. Section 3 describes our evaluation algorithm and Section 4 analyzes our its performance on several metrics, including initialization cost, throughput (framerate), and memory requirements. Section 5 gives techniques for extending our interpretation to handle feature-adaptive subdivision, semi-sharpness schemes, and hierarchical edits. Section 6 concludes.

2 Interpretation of Surface Evaluation

In this section, we present our interpretation of subdivision surface evaluation in the language of linear algebra.

2.1 Abstract Interpretation

For purposes of exposition, we'll define a "linear" subdivision scheme as one in which the surface vertices are linear combinations of control vertices. We phrase the surface evaluation problem as follows:

> Given a vector v of n control vertices and knowledge about their connectivity, compute a vector v' of m vertices on the surface generated by l levels of a linear subdivision scheme.

Aside from linearity, we make no assumptions about the subdivision scheme.

Our algorithm is based on the observation that in a linear subdivision scheme, there must exist a transformation (i.e. matrix) that maps from the space of control vertices to the space of surface vertices. We called this transformation the *subdivision operator*. For basic subdivision, the shape and contents of the subdivision operator are uniquely defined by the subdivision scheme, the number of

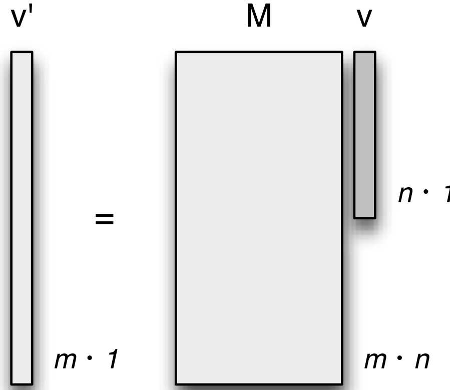


Figure 1: *Illustration of surface evaluation via matrix-vector multiplication. A vector of n control vertices v is left-multiplied by a subdivision operator M, yielding m fine vertices. M is uniquely defined by n, the subdivision scheme, and the connectivity of the control vertices.*

control vertices, and the mesh connectivity. Most importantly, the subdivision operator is independent of the geometric locations of the control vertices—it only specifies how to combine the control vertex locations (or in general, attributes) to compute surface vertex attributes.

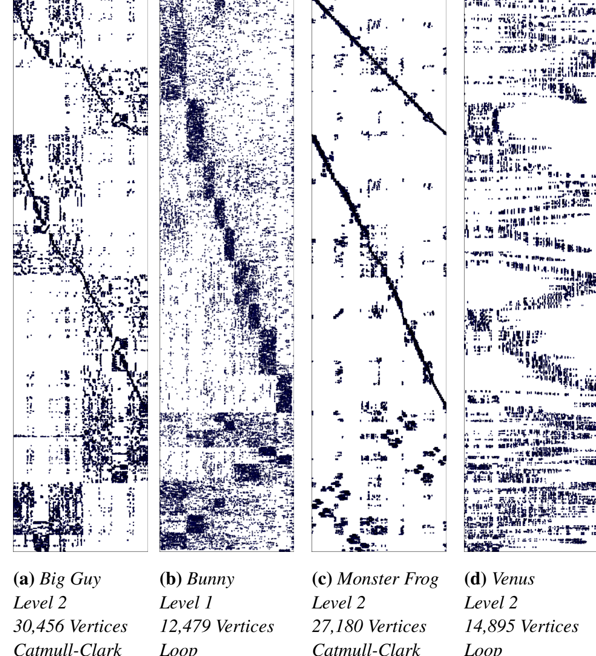
Assuming a linear subdivision scheme hardly limits the practicality of our approach. Two of the most popular schemes, Loop [Loop 2003] and Catmull-Clark [Catmull and Clark 1998], are linear. This fact is apparent upon inspection of the recursive definitions of Loop and Catmull-Clark subdivision surfaces. Vertices are each level are linear combinations of those from the previous level. By induction, the surface vertices at any level must be linear combinations of the control vertices.

Thus, any surface can be evaluated via the product of a subdivision operator M and the vector of control points v. Figure 1 illustrates this operation. M has a characteristic "tall-skinny" aspect ratio that reflects its purpose of mapping a few control points to many surface points. The exact ratio is that of surface vertices to control vertices, and therefore depends on the mesh connectivity and subdivision scheme. In the case of Catmull-Clark subdivision of a perfectly regular mesh, M is four times taller than wide.

Intuitively, each column of M corresponds to a control vertex; similarly, each row corresponds to a surface vertex. The value at $M_{i,j}$ indicates the contribution of control vertex j to surface vertex i. The values in a particular row represent the weights to use in a weighted sum of the control vertices that yields the surface vertex.

It is important to note that M is *sparse*, or mostly zero-valued. This is because a given surface vertex is computed from its corresponding neighborhood in the control mesh. The exact number of contributing vertices depends on the subdivision scheme and mesh regularity, but it is small; we saw about 6-12 non-zeroes per row in practice. The width of M (i.e. the number of control vertices) is usually much larger, so M is mostly zero-valued. This is consistent with local support: moving a distant neighbor of a particular vertex does not affect that vertex because the corresponding weight is zero.

Sparse matrices that represent real systems often have interesting or useful structure. Spy plots reveal this structure by plotting black dots over nonzero elements. Figure 2 shows spy plots for actual



(a) *Big Guy Level 2 30,456 Vertices Catmull-Clark* **(b)** *Bunny Level 1 12,479 Vertices Loop* **(c)** *Monster Frog Level 2 27,180 Vertices Catmull-Clark* **(d)** *Venus Level 2 14,895 Vertices Loop*

Figure 2: *Spy plots of subdivision operators for selected meshes. Surface vertices (or matrix rows) are sorted by type: for n control points (or n matrix columns), roughly the first n rows compute face vertices (if applicable), the next $2n$ rows compute edge vertices, and the last n rows compute vertex-vertices.*

subdivision operators of meshes to be introduced later. The diagonal patterns exhibited in the plots are indicative of nearest-neighbor stencil routines, which is exactly what these subdivision schemes are!

2.2 Constructing the Subdivision Operator

Until now we've merely posited the existence of the subdivision operator. Here, we should how it can be generated for linear subdivision schemes and arbitrary subdivision levels. We first describe how to construct it for one level of subdivision, and then how to combine single-level operators into a multi-level operator.

2.2.1 Single-Level Subdivision Operators

Single-level operators can be constructed rather straightforwardly from the recursive definition of the subdivision scheme. Complications arise in schemes that have intra-level dependencies. For example, in Catmull-Clark subdivision, edge and vertex vertices at level $i+1$ are weighted sums of vertices in level i and face vertices in level $i+1$. The authors see two ways around the dependence: 1) rewrite the definition to avoid intra-level dependencies, or 2) use a product of "intermediate operators" to capture the dependence. The former may be feasible in the case of Catmull-Clark subdivision but not in general, so we opt for the latter.

To compute the subdivision operator for a single Catmull-Clark subdivision step at level i, we first generate an intermediate operator M_f that copies the vertices at level i and simultaneously computes the face vertices at level $i+1$. Then, we generate a second operator that copies the new face vertices and computes the edge and vertex vertices at level $i+1$ in terms of the face vertices and level i vertices. The product of these operators yields a single-level, Catmull-Clark

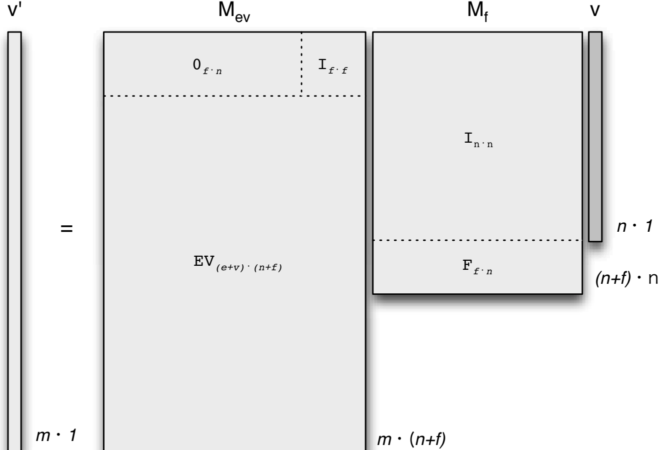


Figure 3: *The intra-level dependence of the Catmull-Clark subdivision scheme requires special handling. Here, operator M_f copies the original vertices and simultaneously computes the positions of the new face vertices. M_{ev} copies the new face vertices while computing the new edge and vertex vertices in terms of the control vertices and the new face vertices. The product $M_{ev} \cdot M_f$ represents the single-level subdivision operator and captures the intra-level dependence.*

subdivision operator, as illustrated in Figure 3. Schemes with more dependencies can be handled by additional intermediate operators.

2.2.2 Multi-Level Subdivision Operators

Multi-level subdivision operators specify how to compute vertices at an arbitrary subdivision level given a vector of control vertices. In short, they are the product of all single-level operators up to the desired level. Figure 4 shows how three subdivision operators can be collapsed into one that represents the transformation directly from the control vertices to the surface vertices.

3 Our Algorithm and Implementation

The previous section interpreted a multi-level subdivision step as the product of the control vertices and a collection of subdivision operators readily constructable from the scheme's recursive definition. Here, we describe an implementation inspired by our interpretation.

For the remainder of the paper, we choose to focus on computing subdivision surfaces for deforming meshes of static topology. Our interpretation promises to be most useful in this arena because the subdivision operator is invariant with respect to the geometric position of the mesh. The operator can be generated a priori; then, evaluating the surface at any frame reduces to applying the operator to the deformed control vertices.

It is important to note that our interpretation prescribes no evaluation order for the product of the single-level subdivision operators and the vector of control vertices. In fact, different orders can provide a trade-off between initialization cost (time to first frame) and throughput (geometry subdivided per unit time) because the multiplication of single-level operators has non-trivial cost. We see three reasonable choices to be made here:

- Combine all single-level operators via multiplication a priori, so a given frame requires a single matrix-vector multiplication between the operator and the control vertices. This option maximizes throughput at the cost of increased initialization cost.
- Combine no single-level operators, so a given frame requires one matrix-vector multiplication per operator. This option minimizes the time to the first frame, but requires more work per frame.
- Combine k single-level operators, so a given frame requires $k-1$ fewer matrix-vector multiplication per frame. k can be chosen to strike the desired balance between initialization cost and throughput.

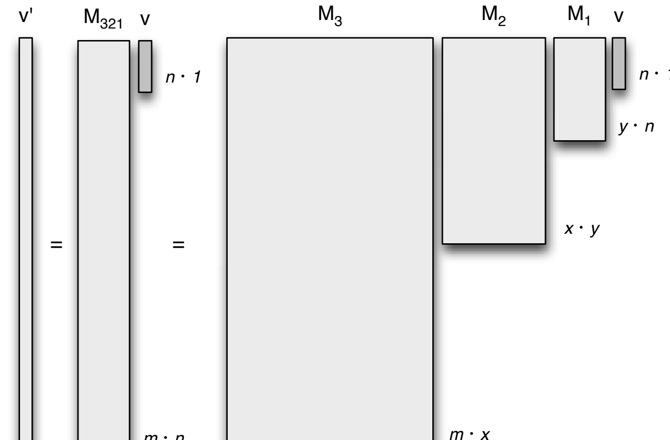


Figure 4: *Multiple subdivision steps can be captured in the product of single-level subdivision operators. Here, M_1, M_2, and M_3 combine to form M_{321}, a transformation from the control vertices directly to the surface vertices.*

Our algorithm chooses to maximize throughput. It works as follows:

1. On the first request to subdivide the control mesh, generate the complete set of single-level subdivision operators. Then, compute the multi-level operator by right-associative multiplication. Store the multi-level operator in memory.
2. Apply the in-memory subdivision operator to the control vertices, yielding the surface vertices.

3.1 Implementation Details

Implementation vehicle. We implemented our algorithm in Pixar's OpenSubdiv framework, an open-source subdivision library that implements approximate subdivision surface evaluation for Catmull-Clark, Loop, and bilinear schemes. Pixar uses OpenSubdiv in their production pipeline, and we consider their code to be reasonably expert.

Subdivision operator representation. Subdivision operators and other sparse matrices can be represented efficiently in a variety of sparse matrix formats that only store non-zero entries and their corresponding locations in the matrix. In our implementation, we construct single-level subdivision operators in coordinate list format (COO) [Wikipedia 2012] and convert them to compressed sparse row format (CSR) [Dongarra 1995] for multiplication. For sparse matrices with nnz non-zero elements, the total memory required is $3 \cdot nnz$ with COO format and $2 \cdot nnz + \#rows$ with

CSR.

High performance libraries. One of the primary benefits of our interpretation is its direct implementability with high-performance numerical linear algebra libraries. Our implementation makes use of three routines available in popular libraries:

- Single-precision sparse matrix-vector multiplication.
- Single-precision sparse-matrix sparse-matrix multiplication.
- Conversion from COO format to CSR format.

The routines are often internally parallelized, freeing the programmer from the burden of writing parallel code, and they can be tuned for particular architectures, yielding portable performance. Packages such as Intel's Math Kernel Library [Intel], Nvidia's cuSPARSE [Nvidia], and University of Florida's CSparse [Davis] have such routines.

Logical subdivision operators The final, multi-level subdivision operator must operate on vertex elements. Our implementation used six elements per vertex, three each for the spatial location and the vertex normal. Nevertheless, the subdivision schemes take linear combinations of vertices, not vertex elements, thus providing a potential optimization: subdivision operators can be built in terms of logical vertex indices and "expanded" to handle vertex elements at the last possible moment. In a logical subdivision operator, M_{ij} denotes the contribution of control vertex j on surface vertex i; in contrast, M_{ij} in the final subdivision operator denotes the contribution of element i to element j in the vector of all vertex elements. To convert from a logical operator, the expansion step replaces each value v with a matrix vI, where I is an identity matrix whose side length matches the expansion factor (6 in our case). This step is amenable to parallelization as every element can be expanded independently.

4 Performance

This section presents a performance evaluation of our approach on modern hardware. We sought to understand the utility of our approach based on initialization cost (time to first frame), throughput (framerate), and memory requirements.

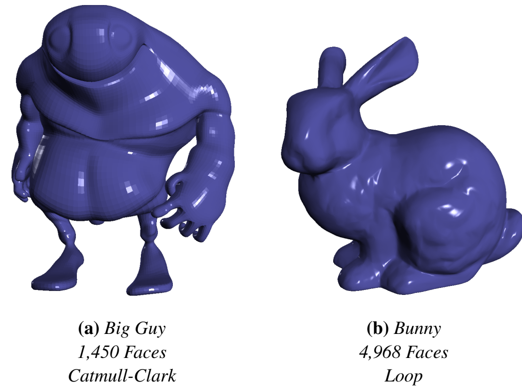
4.1 Testing Methodology

We developed two implementations using vendor-supplied numerical linear algebra libraries. The first is built on Intel's MKL and targets multicore CPUs. The second is built on Nvidia's cuSPARSE library targets GPUs. We ran our experiments on a quad-core, 2.67 GHz Intel Core i7 CPU with an attached Nvidia GeForce GTX480 GPU. Our driver code rendered frames using OpenGL, but times we report are restricted to the surface evaluation subroutine.

We evaluated our algorithm on six models, including three quad meshes and three triangle meshes. Within each category, we chose one small, fundamental shape and two larger, real-world objects. For brevity, we only present results from the Bunny and Big Guy models, shown in Figure 5; results for the remaining models are presented in Appendix A.

4.2 Throughput Performance

Our implementation sought to optimize the surface evaluation subroutine's steady-state performance, measured in geometry computed per unit time. Because our algorithm is only aware of vertices, we report performance in vertices per millisecond instead of the more traditional faces per unit time.



(a) *Big Guy* *1,450 Faces* *Catmull-Clark* **(b)** *Bunny* *4,968 Faces* *Loop*

Figure 5: *Models used in performance analysis.*

Figure 6 shows the throughput attained by our computational kernels (MKL and CuSPARSE) versus the default kernels performing table-driven subdivision. In order to compare apples and apples, we divide the kernels into two classes: CPU-backed (CPU, OpenMP, and MKL) and GPU-backed (Cuda, OpenCL, GLSL, CuSPARSE). It's clear that GPU-backed kernels achieve better absolute performance for nearly all subdivision levels test.

CPU-backed kernels. Our MKL kernel provides substantially better performance than the serial CPU kernel and the naively-parallelized OpenMP kernel. This is likely due to two effects: 1) our algorithm executes fewer flops than table-driven subdivision and 2) MKL makes reasonably good use of the cache hierarchy to maximize sustained memory bandwidth. Specifically, the MKL kernel runs 1.84x faster than the OpenMP kernel, and 7.35x faster than the serial kernel, all without requiring any user-level parallel code.

GPU-backed kernels. Presently, the CuSPARSE kernel lags behind the hand-optimized CUDA, OpenCL, and GLSL kernels. The authors still consider this commendable, as the implementer has only a cursory knowledge of GPU programming yet still nearly matches the performance of expert code. Future work will attempt to optimize the performance of the CuSPARSE kernel.

4.3 Initialization Cost

As presented, our algorithm has a nontrivial cost associated with computing the product of the single-level subdivision operators. Here, we attempt to capture that cost. Our results here can be seen as upper bound on initialization costs for algorithms that combine some or none of the single-level operators. The MKL results reported here should be considered preliminary because the sparse-matrix sparse-matrix multiplication routine in MKL 11.0.1 is not multi-threaded [**?**].

Figure 7 shows how the initialization time for our models varies as a function of subdivision levels.

As a rule of thumb, the MKL kernel takes roughly twice as long to compute the first surface, but then computes subsequent surfaces at twice the rate. The CuSPARSE kernel takes about four times as long.

We envision that a model's subdivision operator will be generated

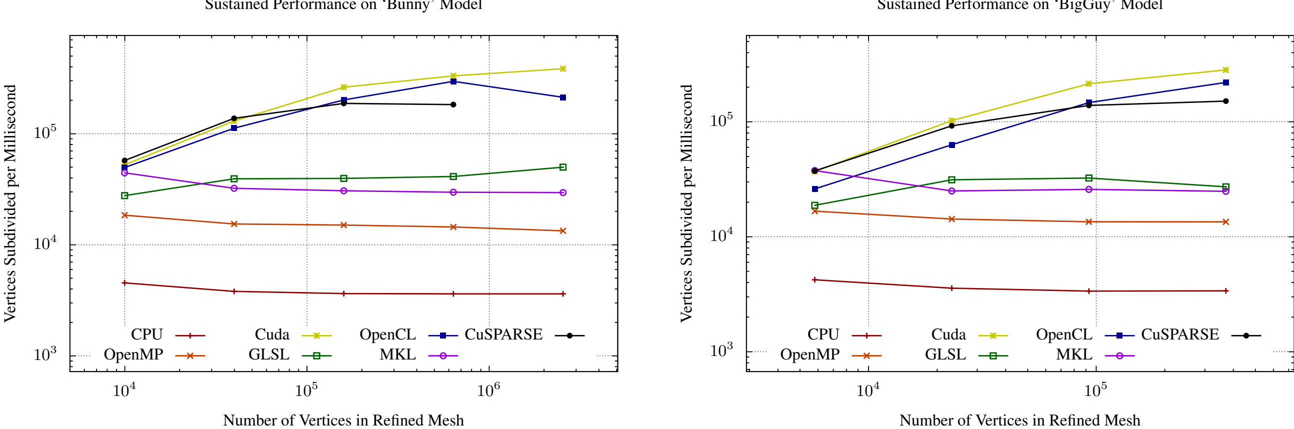


Figure 6: *Throughput as a function of subdivision level.*

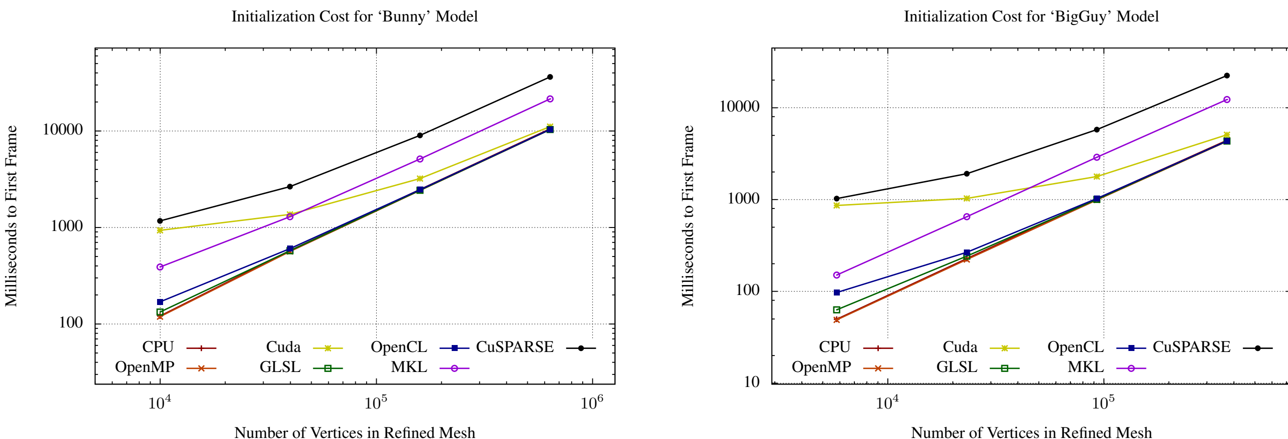


Figure 7: *Initialization time as a function of subdivision level.*

offline, perhaps between the modeling and animation stages of a production film pipeline. If this is the case, non-interactive times are tolerable and clusters can be utilized [Buluç and Gilbert 2010].

4.4 Memory Requirements

Our algorithm requires nontrivial space for storage of the multi-level subdivision operator. Figure 8 shows that our memory requirement scales linearly with the surface resolution. The slope of the curves reveals that we use about 32 bytes per surface vertex element (or $6 \cdot 32 = 192$ bytes per vertex), including the weights themselves and the CSR metadata. Our biggest models required up to a gigabyte of storage at fine resolutions, which we consider reasonable given modern memory capacities.

5 Extensions

5.1 Feature-Adaptive Subdivision

Feature-adaptive subdivision [Niessner et al. 2012] exploits the fact that Catmull-Clark subdivision of regular quad meshes yields bicubic b-spline surfaces. Therefore, feature-adaptive schemes merely subdivide near irregular vertices and leave tessellation of regular regions to the rendering hardware. Our interpretation is still useful for calculating irregular vertex positions, and our algorithm may shine here because the total number of vertices is smaller than in the global subdivision problem.

5.2 Semi-sharpness Schemes

Our technique effortlessly supports the semi-sharpness scheme implemented in OpenSubdiv, based on [DeRose et al. 1998]. The OpenSubdiv mesh representation provides our kernels vertex weights that reflect sharpness tags, so the weights can be immediate placed into single-level subdivision operators. The multiplication of single-level operators 'mixes' the sharpness-informed weights appropriately, yielding the correct multi-level operator.

5.3 Hierarchical Edits

Hierarchical edits require some modification of our scheme. We distinguish between two types of edit operations in OpenSubdiv, add and set.

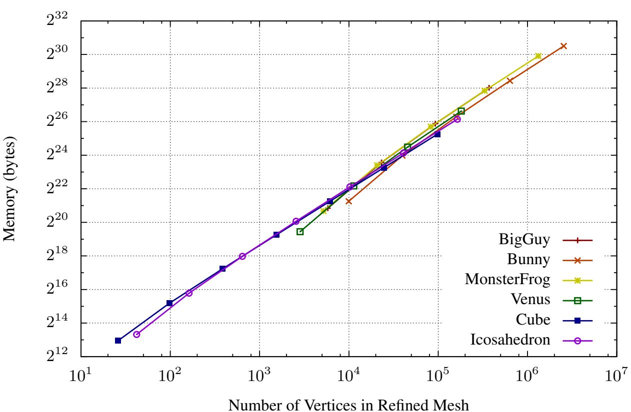


Figure 8: *Memory Usage for Selected Models*

Additive edits Modelers can arbitrarily add a vector to a vertex at any level of subdivision. Fortunately, addition is a linear operation, so we can use the distributive property to compute the addition's effect at the finest level of subdivision; then each frame requires an additional vector-vector add to reflect the edits. Figure 9 illustrates an additive edit at the middle of three subdivision steps.

Additive edits potentially complicate the use of logical subdivision operators (as described in 3.1) if the edits are in terms of vertex elements, not logical vertices. Logical subdivision operators can be expanded early to accommodate such edits.

Nonlinear edits Modelers can also set a vertex element to an arbitrary value at any level of subdivision. Such an operator is inherently nonlinear, so we cannot in general find a representation of the edit at the finest level of subdivision. We suggest in this case building two multi-level subdivision operators for 'either side' of the edit operation. The first operator can be applied to map the control vertices to the intermediate surface, the edits can be applied, and the second operator can map the intermediate vertices to the intended surface vertices.

6 Conclusion

We have presented an interpretation of subdivision surface evaluation in the language of linear algebra. By casting the problem as a matrix-vector multiplication problem, we've enabled the use of high-performance linear algebra libraries that yield substantial speedups over expert production code on multicore CPUs and comparable performance on GPUs. Our evaluation strategy supports many features expected in modern subdivision libraries, including hierarchical edits and semi-sharp creases, and allows users to trade-off initialization cost and frame rate. We expect our techniques to be especially applicable to feature-adaptive subdivision, where problem sizes are smaller but performance is still critical.

Acknowledgments

The authors wish to thank Pixar for open-sourcing their subdivision code. We are also grateful to Evangelos Georganas, Penporn Koanantakool, David Sheffield, and Aydin Buluc for their advice on various aspects of the implementation.

References

ASANOVIC, K., BODIK, R., CATANZARO, B. C., GEBIS, J. J., HUSBANDS, P., KEUTZER, K., PATTERSON, D. A., PLISHKER, W. L., SHALF, J., WILLIAMS, S. W., AND YELICK, K. A. 2006. The landscape of parallel computing research: A view from berkeley. Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec.

BULUÇ, A., AND GILBERT, J. R. 2010. Highly parallel sparse matrix-matrix multiplication. Tech. Rep. UCSB-CS-2010-10, UCSB CS Department, June.

CATMULL, E., AND CLARK, J. 1998. Seminal graphics. ACM, New York, NY, USA, ch. Recursively generated B-spline surfaces on arbitrary topological meshes, 183–188.

DAVIS, T. Cxsparse: an extended version of csparse.

DEROSE, T., KASS, M., AND TRUONG, T. 1998. Subdivision surfaces in character animation. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, SIGGRAPH '98, 85–94.

DONGARRA, J. 1995. Compressed row storage.

INTEL. Intel math kernel library (intel mkl) 11.0.

LOOP, C. T. 2003. *Smooth subdivision surfaces based on triangles*. Master's thesis, University of Utah, Salt Lake City, Utah, USA.

NIESSNER, M., LOOP, C., MEYER, M., AND DEROSE, T. 2012. Feature-adaptive gpu rendering of catmull-clark subdivision surfaces. *ACM Trans. Graph. 31*, 1 (Feb.), 6:1–6:11.

NVIDIA. cusparse.

WIKIPEDIA, 2012. Sparse matrix, Sept.

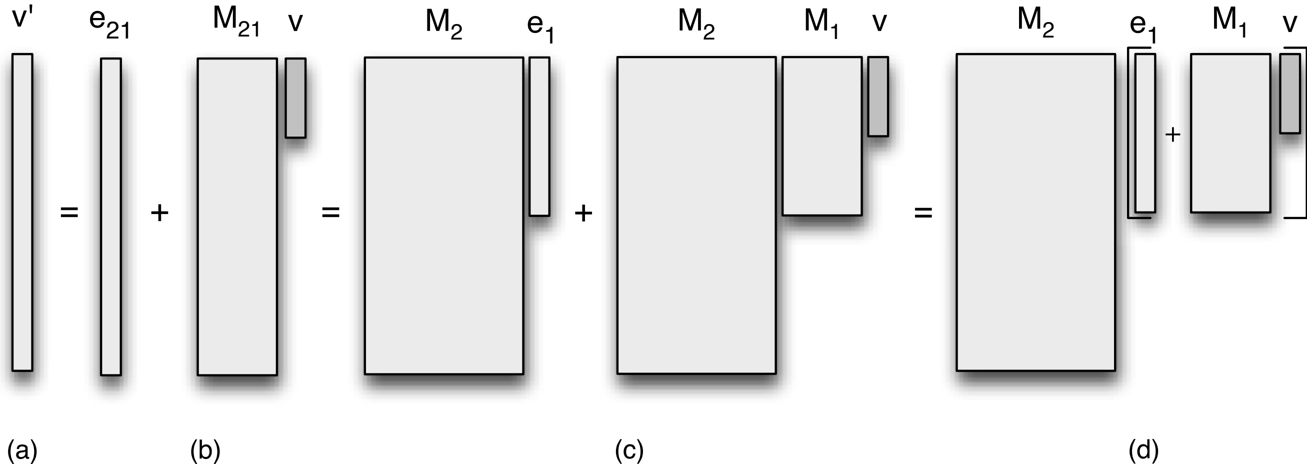


Figure 9: *Linear edits at any subdivision level can be propagated to the finest level a priori and applied in a vector-vector addition step at runtime. Here, (d) shows edit vector* e_1 *applied at level 1. Using the distributive property in (c), the expression can be rewritten to separate the edit vector from the control vertices entirely. If the edits are invariant over time, they can be evaluated to form a vector* e_{21} *which influences the solution in a final vector-vector addition step (b) at runtime.*

A Additional Performance Results

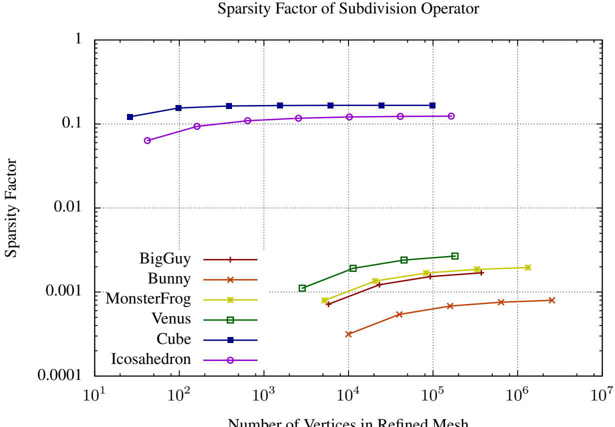
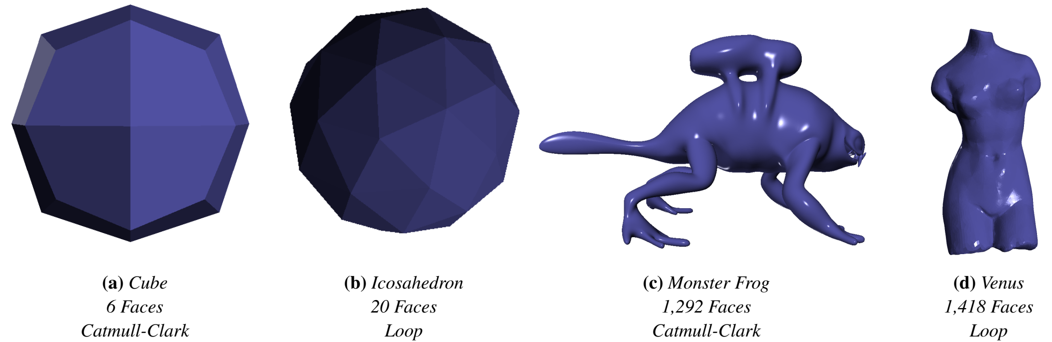


Figure 11: *Sparsity factor of selected meshes at various refinement levels. A factor of 1 indicates a completely dense operator, and a factor of 0 means a completely sparse operators.*



(a) *Cube*
6 Faces
Catmull-Clark

(b) *Icosahedron*
20 Faces
Loop

(c) *Monster Frog*
1,292 Faces
Catmull-Clark

(d) *Venus*
1,418 Faces
Loop

Figure 10: *Additional models used in performance analysis.*

Sustained Performance on 'Venus' Model

Sustained Performance on 'MonsterFrog' Model

Sustained Performance on 'Cube' Model

Sustained Performance on 'Icosahedron' Model

Vertices Subdivided per Millisecond

Number of Vertices in Refined Mesh

CPU, OpenMP, Cuda, GLSL, OpenCL, MKL, CuSPARSE

Figure 12: *Performance results for additional models.*

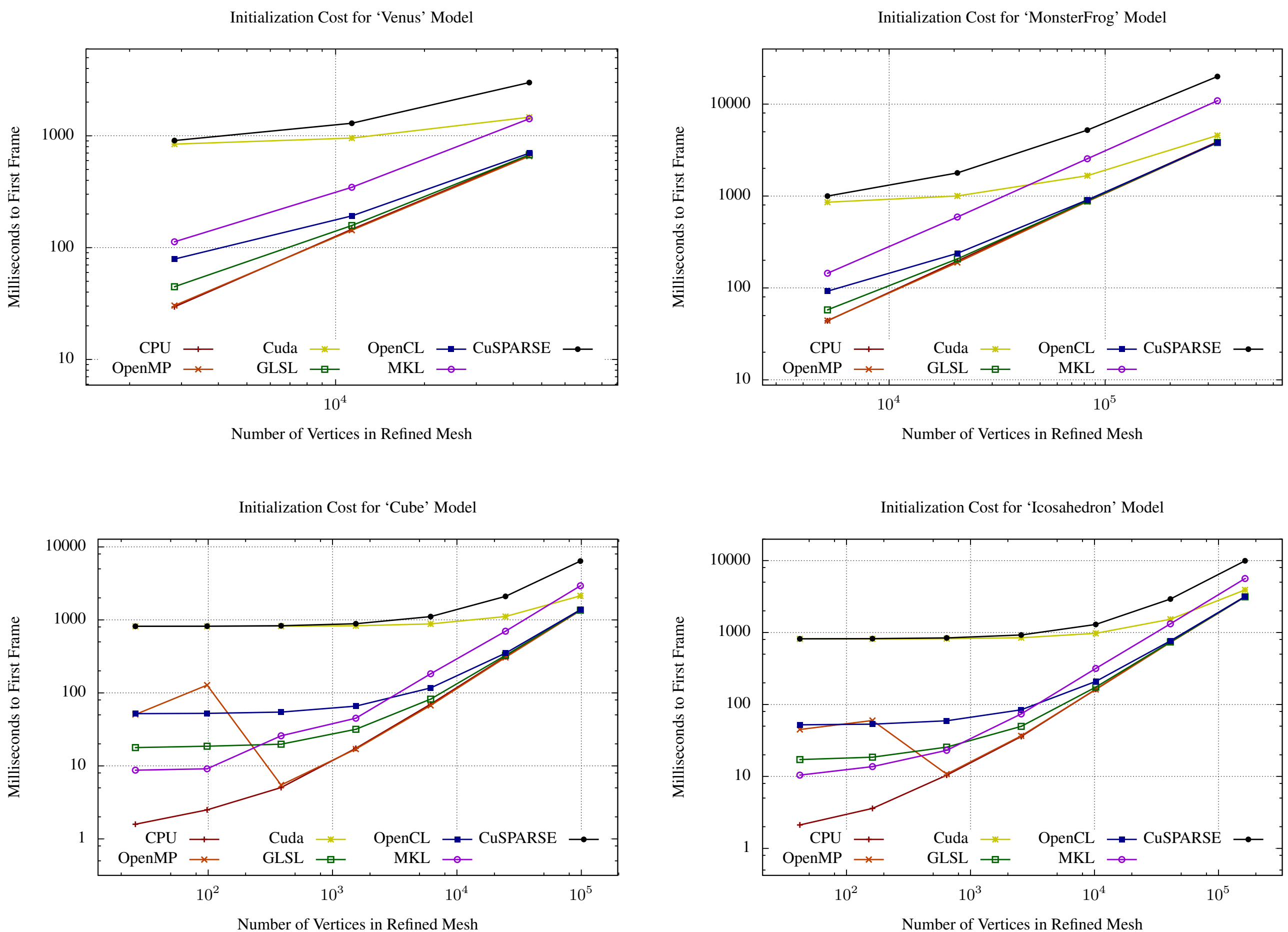


Figure 13: *Initialization time for additional models.*