A Realtime GPU Subdivision Kernel

Le-Jeng Shiue*
University of Florida

Ian Jones†
University of Florida

Jörg Peters‡
University of Florida

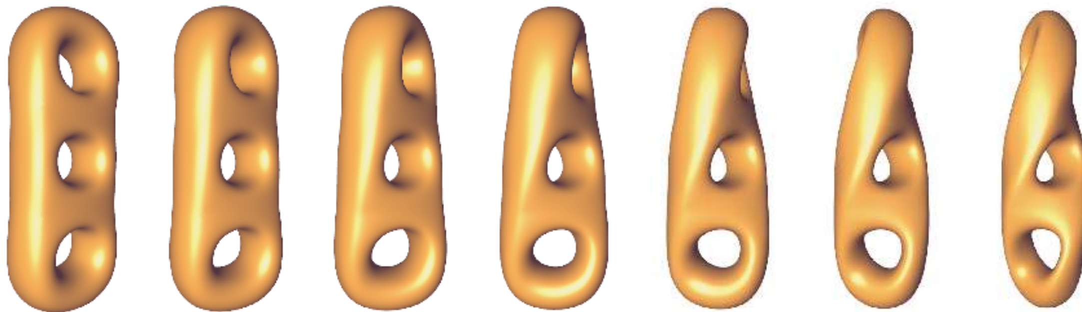

Figure 1: Catmull-Clark surfaces fully reevaluated (40 patches, depth 5) at 21 fps on an ATi 9700 mobile graphics chip.

Abstract

By organizing the control mesh of subdivision in texture memory so that irregularities occur strictly inside independently refinable fragment meshes, all major features of subdivision algorithms can be realized in the framework of highly parallel stream processing. Our implementation of Catmull-Clark subdivision as a GPU kernel in programmable graphics hardware can model features like semi-smooth creases and global boundaries; and a simplified version achieves near-realtime depth-five re-evaluation of moderate-sized subdivision meshes. The approach is easily adapted to other refinement patterns, such as Loop, Doo-Sabin or $\sqrt{3}$ and it allows for postprocessing with additional shaders.

CR Categories: I.3.4 [Computing Methodologies]: Computer Graphics—Software Support I.3.5 [Computing Methodologies]: Computer Graphics—Curve, Surface, Solid, and Object Representations I.3.8 [Computing Methodologies]: Computer Graphics—Graphics Data Structures and Data Type

Keywords: subdivision surfaces, programmable graphics hardware, shader, geometric data structure

1 Introduction

A number of important algorithms have been modified to rebalance the workload between CPU and GPU (graphics processing unit), and take advantage of parallel execution streams in programmable graphics hardwares; e.g. for particle systems [Kipfer et al. 2004; Kolb et al. 2004], collision detection [Govindaraju et al. 2003], cellular automata [Harris et al. 2003], global illumination [Purcell et al. 2003] and other numerical computations [Krüger and Westermann 2003; Bolz et al. 2003]. The algorithmic component on the GPU, called *shader*s, rely essentially on accessing regularly laid out data, typified by the 2D array, to minimize workflow branching and maximize parallelism. Irregular access typically requires interaction with the CPU. Therefore, for algorithms with irregular access such as subdivision near extraordinary points, a good approach is to precompute access patterns offline, and place the precomputed data tables into texture images for use by shaders. An algorithm along these lines was proposed in [Bolz and Schröder 2002] for Catmull-Clark subdivision. It precomputes, for a fixed grid of parameters and the most common configurations, the limit values of the Catmull-Clark generating functions to be combined at runtime. Unfortunately, tabulated evaluation limits flexibility and can increase downstream complexity: many tables are required to even partially reproduce the spectrum of semi-smooth creases [DeRose et al. 1998], and a complex procedure has to be followed to avoid numerical roundoff gaps that can arise from different evaluation order of mesh points [Bolz and Schröder].

By contrast, the proposed realtime GPU subdivision kernel generates the subdivision mesh at different depths on the GPU so that *all* evaluation work rests with GPU shaders and all major features of subdivision algorithms can be realized. The key to this approach is a locality-preserving data access that keeps all irregularities strictly inside independently refinable pieces of the mesh, called *fragment meshes*. The resulting parallel streams of workflow per fragment mesh and also per mesh node leverage the strengths of the GPU compared to the CPU. The resulting implementation performs well when measured in wall-clock frames-per-second (fps) on current GPUs as is relevant for applications such as gaming and interactive animation. The approach is not restricted to a specific subdivision scheme or hardware and can compute

- all major refinement patterns [Catmull and Clark 1978; Doo and Sabin 1978; Loop 1987; Kobbelt 2000],
- semi-smooth creases and global boundaries, and
- delivers a watertight mesh directly renderable within the GPU.

As Figure 1 illustrates, our implementation of Catmull-Clark subdivision, based on the OpenGL Shading Language [John Kessenich and Rost 2004], achieves near-realtime speed on an ATi 9700 mobile graphics processor when fully reevaluating a moderately sized model. Larger models, such as in Figure 2, require adaptive refinement and adaptive reevaluation to reach this frame-rate.

*e-mail: sle-jeng@cise.ufl.edu
†e-mail: ijones@cise.ufl.edu
‡e-mail: jorg@cise.ufl.edu

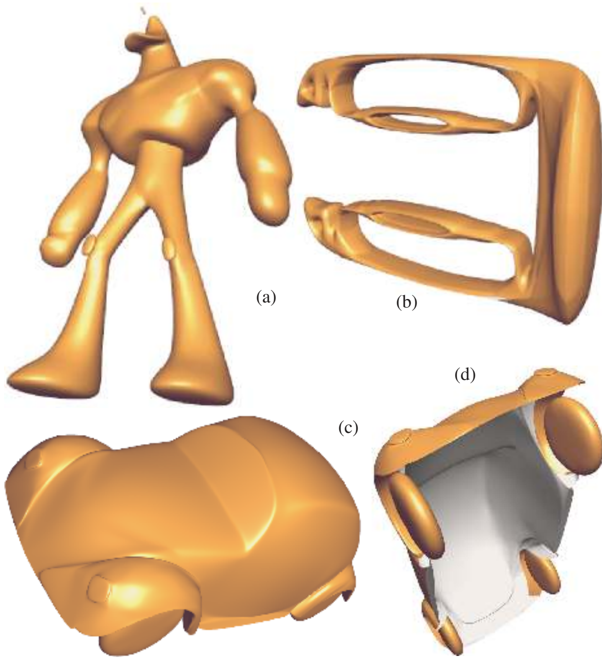

Figure 2: Subdivision surfaces generated by our GPU subdivision kernel featuring semi-smooth creases (b) and global boundaries (c,d).

2 Background: Subdivision Algorithms, Spatial Data Structures and Shaders

Subdivision algorithms create smooth surface approximations by recursively refining the connectivity and smoothing the geometry of a polyhedral input mesh, known as the *control mesh* (see e.g. [DeRose et al. 1998; Warren and Weimer 2002]). In all popular algorithms the position of a new mesh node is obtained as the weighted average of old nodes of a small submesh, whose graph is called *stencil*. The major refinement patterns are shown in Figures 3, and the stencils with the weights of Catmull-Clark subdivision are shown in Figures 4.

Subdivision implementations can be characterized by their underlying spatial data structures. *Halfedge* data structures [Weiler 1985; Guibas and Stolfi 1983; Kettner 1999]) are maximally flexible in reconfiguring general meshes and interacting with mesh processing algorithms. However, storing, maintaining and refining explicit connectivity pointers is costly. The *quad-tree*-based implementation (see e.g. [Zorin 1999] for Loop's subdivision) naturally supports adaptive refinement: each level of the tree represents one refinement level of the mesh. However, the resulting recursive non-uniform tree structure does not easily lend itself to parallel computation; and rendering may require an expensive traversal to neighbors in remote branches. *Patch-based* refinement has been advocated for fast rendering [Pulli and Segal 1996], parallel evaluation [Padrón et al. 2002], a hardware mesh framework [Shiue et al. 2003], hardware implementation [Bóo et al. 2001; Müller and Havemann 2000], cut-and-paste editing [Biermann et al. 2002], and surface conversion [Peters 2000]. Each patch represents the refined submesh corresponding to an initial facet. The patches are connected by a general, say halfedge, mesh data structure that can often be identical to the initial data structure of the control mesh. The internal connectivity is encoded in the row-column indexing of a 2D-array so that the structure is easy to implement and uses

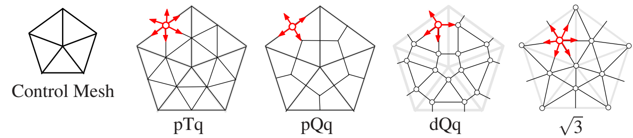

Figure 3: Four different refinement strategies of a triangle fan generally characterized by: p=primal, d=dual, T=triangulation, Q=quadrilateral, q=quad-split. The refined fan can be constructed by moving the *red* pointer-patterns along an outward spiraling path.

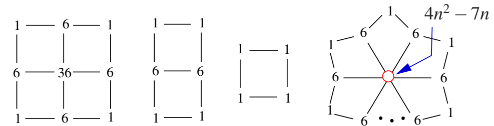

Figure 4: (*from left to right*) Catmull-Clark (pQq) subdivision stencils with weights (to be scaled to sum to 1) for a regular vertex-node, edge-node, facet-node, and an irregular vertex-node with n neighbors.

memory efficiently and contiguously. [Bunnell 2005] implements Catmull-Clark subdivision this way. On the downside, patch-based refinement has only been developed for pTq and pQq patterns, requires extra structures to access the neighbors of irregular nodes, and care has to be taken to avoid numerical roundoff gaps due to different evaluation order of shared points at patch boundaries. Table 1 compares the three (generically implemented) approaches with the proposed approach (labelled Frag-mesh). Here 'n-gon' refers to the ability to represent multi-sided mesh facets, 'locality' to the continuity of refined nodes in memory and 'slower' or 'worse' are characterizations relative to the other schemes.

	Flexibility			Efficiency		
	n-gon	Dual/$\sqrt{3}$	Adaptive	Time	Space	Locality
Halfedge	yes	yes	Yes	slower	worse	no
Quadtree	no	no	Yes	slower	worse	no
Patch-based	no	no	per patch	faster	better	yes
Frag-mesh	yes	yes	per frag	faster	better	yes

Table 1: Subdivision surface implementations compared.

Modern GPUs provide programmable parallel stream processing in the form of *vertex shaders* and *fragment shaders* [Lindholm et al. 2001]. Vertex shaders process attributes, such as positions, normals, and texture coordinates, of a single vertex without connectivity. The downstream fragment shaders process the rasterized data (i.e. attributes per pixel) and assign the resulting pixels. Fragment shaders are the key computation units for most GPU algorithms (as well as ours) because of their computation power and ability to read and write data by rendering to the framebuffer and copying to readable texture images. Strategies and techniques for computation on GPUs can be found in [Harris et al. 2004].

3 Algorithm and Implementation

This section explains the subdivision kernel for Catmull-Clark subdivision. Adjustments to other refinement patterns are discussed in Section 4. The algorithm and the work distribution of CPU and GPU are summarized in Figure 5. The control mesh is broken up into fragment meshes consisting of two layers surrounding a vertex. The fragment meshes are refined independently within the

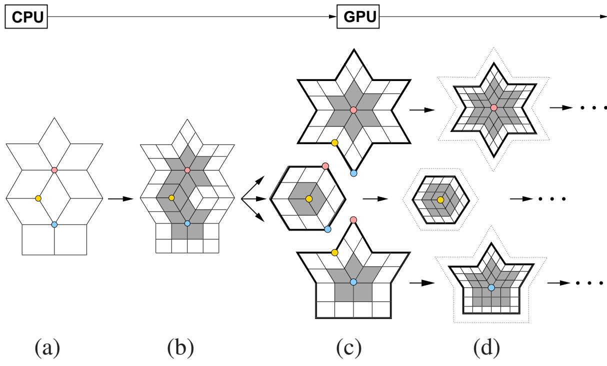

Figure 5: Workflow and data distribution. On the CPU, the control mesh is once refined and decomposed into fragment meshes that overlap. On the GPU, each fragment mesh is processed independently and the overlap shrinks towards the shared boundary.

GPU (and can be processed in parallel on multiple GPUs). The final fragment meshes have exactly matching boundaries and can be rendered directly.

Pre-computation (CPU level): At the CPU level, the control mesh is once refined, if necessary, so that irregular vertices are isolated by one layer of regular nodes. The mesh is broken up into multisided *fragment meshes*, each consisting of one possibly irregular *center node* and *two layers* of nodes surrounding it. Neighboring fragment meshes overlap in two facet mesh layers (Figure 5(b,c)), the outer of which (shaded in Figure 5(c,d)) is discarded when rendering the final fragment mesh.

Nodes in a fragment mesh are mapped into a 1D texture, called *patch-texture*, by starting with the central vertex and spiraling outward to cover two rings, the footprint of Catmull-Clark subdivision. The patch-texture contains pixels of floating-point channels, and each RGBA pixel in the patch-texture represents the (x,y,z,w) coordinates of a node. The texture coordinate is the index for accessing the node. The spiral enumeration naturally encodes the multisided fragment mesh avoiding the creation and maintenance of an explicit structure of the connectivity within a patch; only for the input mesh, which is subject to interaction and structural modification by a user program, do we use a general halfedge data structure.

Subdivision kernel (GPU level): The subdivision kernel scales the patch-texture and filters the pixels similar to geometry images [Gu et al. 2002; Losasso et al. 2003], but on a more complex mesh structure (and therefore without the need for initial geometric resampling and topological remeshing). A patch-texture of valence n at depth d contains

$$\sigma_{n,d} := nq(q-1)+1, \quad \text{pixels, where } q := 2^{d-1}+2.$$

The patch-texture at depth $d+1$ is computed by applying the subdivision stencils to the patch-texture at depth d as illustrated in Figure 6. Grouped by valence, fragment meshes are processed depth first: multiple refinement passes are applied to one fragment mesh up to the prescribed subdivision depth. This minimizes switches in the OpenGL context, the patch-texture, the shader, and the *lookup table*. (The lookup table is discussed in detail below. It provides the indices of the patch-texture nodes for stencil application).

To invoke the kernel, the viewport is initialized to contain $\sigma_{n,d+1} \times 1$ pixels, and a full-screen quad is drawn with the data-dimensioned viewport that has one texel per pixel. The rasterizer interpolates the node indices (texture coordinates) for each pixel. In parallel, for each pixel, the fragment shader obtains the patch-texture indices

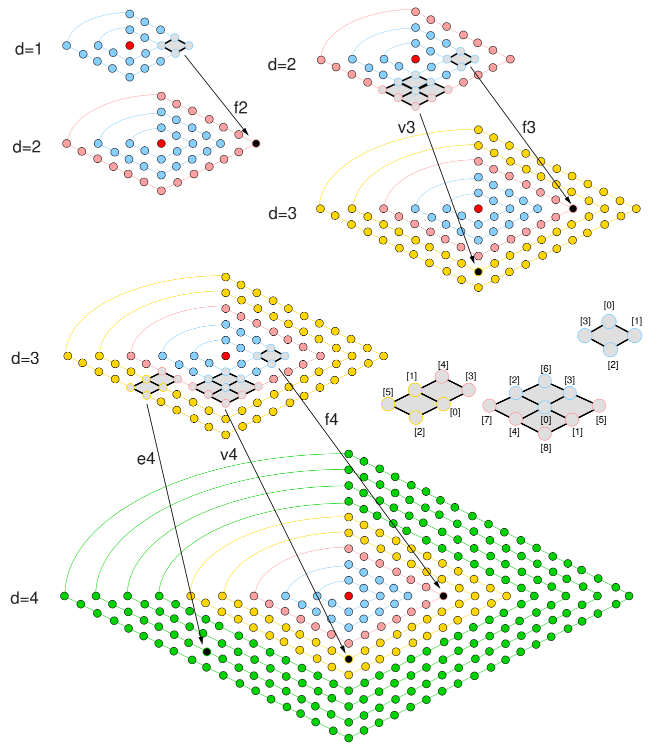

Figure 6: Catmull-Clark subdivision from depth $d=1$ to 2 (*upper left pair*), depth 2 to 3 (*upper right*) and depth 3 to 4. The initial fragment mesh consists of a central node (*red*) and two surrounding layers (*blue*). Subdivision steps $d=2,3,4$ each add $1,2,4$ orbits (*pink*, *yellow*, *green*). Colors and node labels agree with Figure 7.

of the nodes in the local stencil from a lookup table, retrieves the stencil node attributes from the input patch-texture, and computes and renders the new pixel. The intermediate result is copied from the framebuffer to the patch-texture for the next subdivision pass.

At the final depth, a shader computes the normals. Conceptually, the normals are placed into a normal array and, ignoring the outermost ring of nodes, the patch-texture vertices are placed into a vertex array. To avoid rendering via the CPU, we use the superbuffer extension [Mace 2004] binding the rendering target to the vertex and the normal array. Additional shaders can be applied as usual after this stage (Figure 10).

Fragment Shader Details: Using the OpenGL shading language, we implemented the regular Catmull-Clark-stencils as in Listing 1: all three regular stencils are computed, and the valid one is numerically masked to avoid branching. Central, irregular nodes are computed in separate shaders. To support general crease rules, we flag nodes on edges of the control mesh in the lookup table and store the crease value in the alpha channel of the pixel. (Shader length restrictions on the ATi 9700 forced us to break up the shader into three in this case, tripling the execution time since each pixel is visited three times and two computations are discarded). Global boundaries are implemented as creases: the patch-texture is padded with 0s to account for the missing sectors that are ignored during rendering.

Since overlapping nodes are always regular and $A \; op \; B = B \; op \; A$ on the GPU, i.e. pairwise operations commute, the crosswise pairing in Listing 1 for `vertPos, edgePos, facePos` results in an identical computation of corresponding nodes in adjacent fragment meshes.

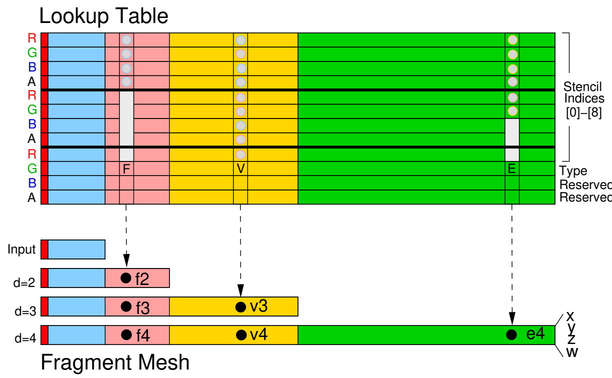

Figure 7: A stencil lookup table for depth 4 (same color coding as in Figure 6). (*top*) Three RGBA channels (12 rows) store the indices of stencil neighbors and additional tags. Three columns, corresponding to v, e and f labels in Figure 6, show in more detail the color-coded neighbor indices (and the unused channels). The four *bottom* bars illustrate the growth of the patch-texture with refinement. Each slot stores the (x, y, z, w) coordinates of a node.

In other words, the refined mesh in the GPU is water-tight and no stitching is required.

Lookup Table (Pre-computed Offline): For a given subdivision scheme, one lookup table per valence n is created once and for all and stored as a texture image in RGBA pixel format (Figure 7). The layout and number of entries of the table is the same as for the patch-texture (but the entries are fixed). Column i of this texture lists the level d patch-texture indices of all nodes contributing to a new node i at level $d+1$. Nine channels suffice for the maximum Catmull-Clark stencil. A tenth channel stores the stencil type (vertex, edge or facet, cf. Figure 6) and the remaining two channels are reserved for depth, crease and boundary flags. The first entry of each lookup table, corresponding to the central valence n node, is ignored since its stencil neighbors are known to be the first $2n+1$ nodes.

To cover more subdivision steps, the columns for new nodes are simply appended to the lookup table. That is the lookup table at depth $d-1$ consists of the lower entries of the lookup table of depth d. By filling the lookup table to the maximum subdivision depth, all lower depth indices are available. Alternatively, stencil lookups could be computed on the GPU by circulating the stencil along with the spiral, but this makes the shader program needlessly complex.

```
// The input patch-texture
uniform sampler2D InputPatch;
// The lookup table texture
uniform sampler2D LookUp;
// Texture coordinates to access lookup table
varying vec2 LookupTC;

// Subdivision stencil type
#define FACE_NODE 1
#define EDGE_NODE 2
#define VERTEX_NODE 4

// Transformation from index to texture coordinate
#define IDX2TC (1.0/2048.0)

void main(void) {
  // Collect the lookup table entry
  vec4 rgba1 =
    texture2D(LookUp, vec2(LookupTC.s, 0.0/3.0))*IDX2TC;
  vec4 rgba2 =
    texture2D(LookUp, vec2(LookupTC.s, 1.0/3.0))*IDX2TC;

  vec4 rgba3 = texture2D(LookUp,vec2(LookupTC.s,2.0/3.0));
  int type = int(rgba3.g);
  rgba3 = rgba3*IDX2TC;

  // Collect the stencil nodes in the input patch-texture
  // as in Figures 6 and 7.
  vec4 S[9];
  S[0] = texture2D(InputPatch, vec2(rgba1.r, 0));
  S[1] = texture2D(InputPatch, vec2(rgba1.g, 0));
  S[2] = texture2D(InputPatch, vec2(rgba1.b, 0));
  S[3] = texture2D(InputPatch, vec2(rgba1.a, 0));
  S[4] = texture2D(InputPatch, vec2(rgba2.r, 0));
  S[5] = texture2D(InputPatch, vec2(rgba2.g, 0));
  S[6] = texture2D(InputPatch, vec2(rgba2.b, 0));
  S[7] = texture2D(InputPatch, vec2(rgba2.a, 0));
  S[8] = texture2D(InputPatch, vec2(rgba3.r, 0));

  // Compute the position using the vertex-stencil
  vec4 vertPos = S[0]*9.0/16.0 +
                 ((S[1]+S[2]) + (S[3]+S[4]))*3.0/32.0 +
                 ((S[5]+S[7]) + (S[8]+S[6]))/64.0;
  // Compute the position using the edge-stencil
  vec4 edgePos = (S[0]+S[1])*3.0/8.0 +
                 ((S[2]+S[4]) + (S[3]+S[5]))/16.0;
  // Compute the position using the face-stencil
  vec4 facePos = ((S[0]+S[2]) + (S[1]+S[3]))/4.0;

  // Assign the valid position by numerical masking
  gl_FragColor = vertPos*float(type == VERTEX_NODE) +
                 edgePos*float(type == EDGE_NODE) +
                 facePos*float(type == FACE_NODE);
}
```

Listing 1: Fragment shader for regular Catmull-Clark stencils.

4 Generalization

As opposed to 2D-array patch-based schemes, spiral enumeration is not restricted to primal quadrisection. By associating different stencils, all the refinement patterns shown in Figure 3 can be implemented. For example, creating a pTq subdivision shader (Loop subdivision) is as easy as replacing the lookup table and slightly changing the scheme for generating fragment meshes on the CPU. For dQq schemes (Doo-Sabin subdivision), one refinement is performed up-front and fragment meshes consist of two layers of nodes surrounding a central facet. For $\sqrt{3}$, two steps of refinement are performed as part of the initialization and fragment meshes consist of three layers of nodes surrounding a central node. Changing to a different subdivision scheme with the same refinement pattern only requires changing the stencil weights in the shaders.

The spiral enumeration works more generally than just for fragment meshes. For example, it allows refining submeshes grouped together as in Figure 8. Collecting and accessing neighbors across group boundaries then only requires visiting nodes in reverse orientation (for details see [Shiue and Peters 2005]). Downstream techniques of modeling and rendering, developed for patch-based access, are easily adapted to spiral enumeration.

Finally, using a lookup table for the weights in addition to the stencil lookup table, spiral enumeration can be used to directly evaluate to a fixed depth of refinement; or to apply stencils that evaluate to the limit position on a fixed-size grid. The latter achieves the same effect as [Bolz and Schröder 2002] with the corresponding drawback of requiring a specialized weight table for different scenarios of semi-smooth creases (but with a simpler scheme to avoid boundary mismatches).

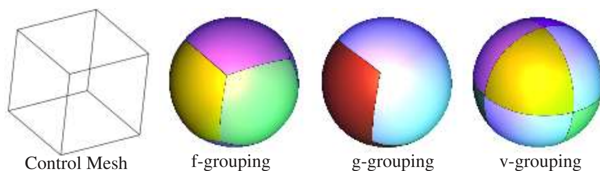

Figure 8: Groupings of the cube mesh refinable by spiral enumeration. The g-grouping contains only two submeshes.

5 Performance Analysis and Results

Lookup table textures are generated offline. On the CPU, the work to construct fragment meshes is proportional to the number of nodes in the mesh. On the GPU, the vertex processor and the rasterizer only process four vertices per fragment mesh. The work mainly rests with the fragment processor and consists of the computations and random access of the lookup table and the patch-texture. This and the pbuffer context switches are the current bottleneck. (We expect that new memory technologies will reduce the performance impact of the context switching so that performance should become proportional to the pixel throughput of the fragment shader.)

Our implementation comparisons were performed on a laptop with a 1.8GHz Pentium M CPU, 1Gb RAM and an ATi 9700 mobile GPU with 4 pixel pipelines, 24-bit floating point precision in the fragment shader, 32 bit textures with a maximum texture size of 2048×2048, 400 MHz core and 200MHz memory. On the CPU, both patch-based and spiral enumeration refinement of depth four execute twice as fast as when based on the fastest, widely available halfedge structure (CGAL [Kettner 1999]). The subdivision shader improves this speed a remarkable *20 times*, yielding 20+ frames-per-second (fps) for moderately sized models. As expected, due to the inherent per-patch and per-node parallelism, the frames-per-second (fps) measure degrades linearly with the size of the input mesh: if we render a scene with k copies of the model in Figure 1, the frame rate reduces to $21/k$ fps. Correspondingly, increasing the number of pipelines or reducing the number of patches by backface culling yields linear speed up for our implementation.

6 Conclusion

Implementing subdivision via spiral-enumerated fragment meshes preserves data locality, makes collecting neighbors simple, and is efficient even for complex control mesh configurations due to parallel stream processing both per patch and per pixel. The approach will directly benefit from extensions such as superbuffers, CPU/GPU clustering and hardware-implementation of the static lookup tables. We anticipate that size limitation of the shader will not be a factor with the next graphics hardware generation, and that the increase of parallel execution streams on a single GPU and clustering of GPUs will allow realtime animation suitable for interactive gaming with multiple large input meshes. We are currently comparing a number of adaptive update and rendering techniques noting that, since the subdivision mesh is computed at intermediate depths, it is straightforward to add displacement at different resolutions.

Acknowledgements We thank Vineet Goel, Mark Segal, Eric Xiaobin Wu and Minho Kim for their valuable comments. The work was supported by NSF Grants DMI-0400214 and CCF-0430891.

References

BIERMANN, H., MARTIN, I., BERNARDINI, F., AND ZORIN, D. 2002. Cut-and-paste editing of multiresolution surfaces. In *SIGGRAPH '02 Conference Proceedings*, 312–321.

BOLZ, J., AND SCHRÖDER, P. Evaluation of subdivision surfaces on programmable graphics hardware. http://www.multires.caltech.edu/pubs/GPUSubD.pdf.

BOLZ, J., AND SCHRÖDER, P. 2002. Rapid evaluation of Catmull-Clark subdivision surfaces. In *Proceedings of the Web3D 2002 Symposium*, 11–18.

BOLZ, J., FARMER, I., GRINSPUN, E., AND SCHRÖDER, P. 2003. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. In *SIGGRAPH '03 Conference Proceedings*, 917–924.

BÓO, M., AMOR, M., DOGGETT, M., HIRCHE, J., AND STRASSER, W. 2001. Hardware support for adaptive subdivision surface rendering. In *Proceedings of the Workshop on Graphics Hardware*, 33–40.

BUNNELL, M. 2005. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley, Reading, MA, ch. Adaptive Tessellation of Subdivision Surfaces with Displacement Mapping.

CATMULL, E., AND CLARK, J. 1978. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer Aided Design 10*, 350–355.

DEROSE, T., KASS, M., AND TRUONG, T. 1998. Subdivision surfaces in character animation. In *SIGGRAPH '98 Conference Proceedings*, 85–94.

DOO, D., AND SABIN, M. 1978. Behaviour of recursive division surfaces near extraordinary points. *Computer Aided Design 10* (Sept.), 356–360.

GOVINDARAJU, N. K., REDON, S., LIN, M. C., AND MANOCHA, D. 2003. Cullide: interactive collision detection between complex models in large environments using graphics hardware. In *Proceedings of the Conference on Graphics Hardware*, Eurographics Association, 25–32.

GU, X., GORTLER, S. J., AND HOPPE, H. 2002. Geometry images. In *SIGGRAPH '02 Conference Proceedings*, 335–361.

GUIBAS, L. J., AND STOLFI, J. 1983. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, 221–234.

HARRIS, M. J., BAXTER, W. V., SCHEUERMANN, T., AND LASTRA, A. 2003. Simulation of cloud dynamics on graphics hardware. In *Proceedings of the Conference on Graphics Hardware*, Eurographics Association, 92–101.

HARRIS, M., LUEBKE, D., BUCK, I., GOVINDARAJU, N., KRÜGER, J., LEFOHN, A. E., PURCELL, T. J., AND WOOLLEY, C. 2004. GPGPU: General-purpose computation on graphics hardware. *Course notes 32 of SIGGRAPH 2004*.

JOHN KESSENICH, D. B., AND ROST, R. 2004. The OpenGL shading language (version 1.10. Tech. rep., April.

KETTNER, L. 1999. Using generic programming for designing a data structure for polyhedral surfaces. *Computational Geometry 13*, 1 (May), 65–90.

KIPFER, P., SEGAL, M., AND WESTERMANN, R. 2004. Uberflow: A GPU-based particle engine. In *Eurographics Symposium Proceedings Graphics Hardware 2004*, 115–122.

KOBBELT, L. 2000. $\sqrt{3}$ subdivision. In *SIGGRAPH '00 Conference Proceedings*, 103–112.

KOLB, A., LATTA, L., AND REZK-SALAMA, C. 2004. Hardware-based simulation and collision detection for large particle systems. In *Eurographics Symposium Proceedings Graphics Hardware 2004*.

KRÜGER, J., AND WESTERMANN, R. 2003. Linear algebra operators for gpu implementation of numerical algorithms. In *SIGGRAPH '03 Conference Proceedings*, 908–916.

LINDHOLM, E., KLIGARD, M. J., AND MORETON, H. 2001. A user-programmable vertex engine. In *SIGGRAPH '01 Conference Proceedings*, 149–158.

LOOP, C. T., 1987. Smooth subdivision surfaces based on triangles. Master's Thesis, Department of Mathematics, University of Utah.

LOSASSO, F., HOPPE, H., SCHAEFER, S., AND WARREN, J. 2003. Smooth geometry images. In *Proceedings of the symposium on Geometry processing*, 138–145.

MACE, R. 2004. OpenGL ARB Superbuffers (OpenGL tutorial). Tech. rep. Game Developers Conference.

MÜLLER, K., AND HAVEMANN, S. 2000. Subdivision surface tesselation on the fly using a versatile mesh data structure. *Computer Graphics Forum 19*, 3 (Aug.).

PADRÓN, E. J., AMOR, M., BÓO, M., AND DOALLO, R. 2002. Efficient parallel implementations for surface subdivision. In *Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization*, 113–121.

PETERS, J. 2000. Patching Catmull-Clark meshes. In *SIGGRAPH '00 Conference Proceedings*, 255–258.

PULLI, K., AND SEGAL, M. 1996. Fast rendering of subdivision surfaces. In *Proceedings of the EUROGRAPHICS Workshop on Rendering Techniques*, 61–70.

PURCELL, T. J., DONNER, C., CAMMARANO, M., JENSEN, H. W., AND HANRAHAN, P. 2003. Photon mapping on programmable graphics hardware. In *Proceedings of the Symposium on Graphics Hardware*, Eurographics Association, 41–50.

SHIUE, L.-J., AND PETERS, J. 2005. A pattern-based data structure for manipulating meshes with regular regions. In *Graphics Interface*. to appear.

SHIUE, L.-J., GOEL, V., AND PETERS, J. 2003. Mesh mutation in programmable graphics hardware. In *Proceedings of the Conference on Graphics Hardware*, 15–24.

WARREN, J., AND WEIMER, H. 2002. *Subdivision Methods for Geometric Design*. Morgan Kaufmann Publishers.

WEILER, K. 1985. Edge-based data structures for solid modeling in curved-surface environments. *IEEE Computer Graphics and Applications 5*, 1 (Jan.), 21–40.

ZORIN, D. 1999. Implementing subdivision and multiresolution meshes. *Chapter 6 of Course notes 37 of SIGGRAPH 99*.

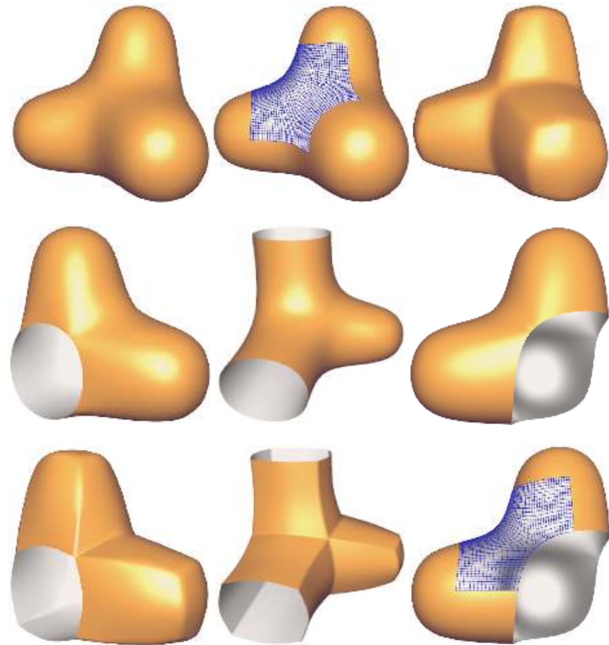

Figure 9: Global boundaries, semi-smooth creases, and the parameter lines of two fragment meshes (one in the interior, one on the boundary) generated in realtime by our shader implementation.

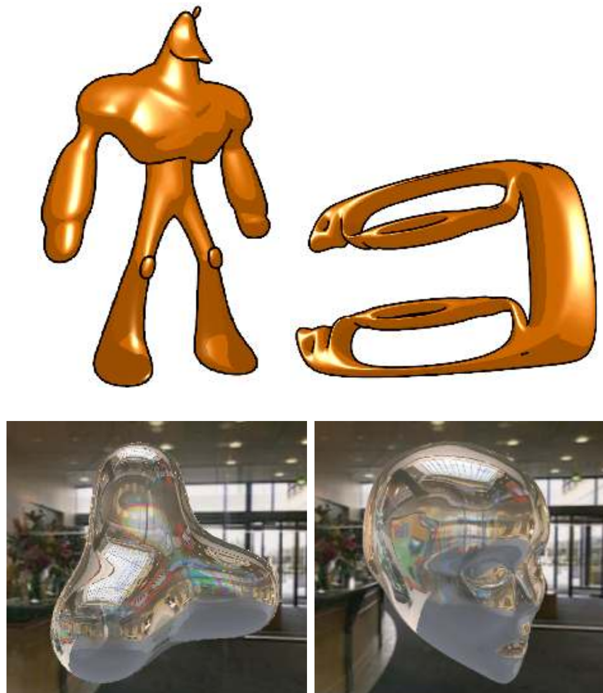

Figure 10: Application of a cartoon-shader (*top*) and a glass-shader (*bottom*) following the subdivision kernel.