Graphics Hardware (2003)
M. Doggett, W. Heidrich, W. Mark, A. Schilling (Editors)

Mesh Mutation in Programmable Graphics Hardware

Le-Jeng Shiue,[1 †] Vineet Goel,[2 ‡] Jorg Peters,[1 §]

[1] CISE, University of Florida [2] ATI

Abstract

We show how a future graphics processor unit (GPU), enhanced with random read and write to video memory, can represent, refine and adjust complex meshes arising in modeling, simulation and animation. To leverage SIMD parallelism, a general model based on the mesh atlas is developed and a particular implementation without adjacency pointers is proposed in which primal, binary refinement of, possibly mixed, quadrilateral and triangular meshes of arbitrary topological genus, as well as their traversal is supported by user-transparent programmable graphics hardware. Adjustment, such as subdivision smoothing rules, is realized as user-programmable mesh shader routines. Attributes are generic and can be defined in the graphics application by binding them to one of several general addressing mechanisms.

1. Introduction

The algorithmic refinement and local adjustment of polyhedral meshes is a core operation of graphics modeling and of simulation for animation. Prime examples are the 'skinning' of animation characters using generalized subdivision surfaces, such as Catmull-Clark [6] and Loop subdivision [15] (see Figure 2), feature enhancement using displacement mapping (Figure 22), or computing functions on meshes (like the 'game of life' in Figure 23). The purpose of the structures proposed in this paper is to improve efficiency, flexibility and display quality resulting from these mesh operations by moving work to the GPU level.

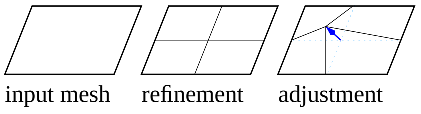

Figure 1: *Basic mesh mutation steps.*

A polyhedral mesh is a special graph. Its *nodes* carry attribute information such as position, normal, texture coordinates and its edges represent neighbor connectivity. Meshes can be visualized by displaying node positions as points

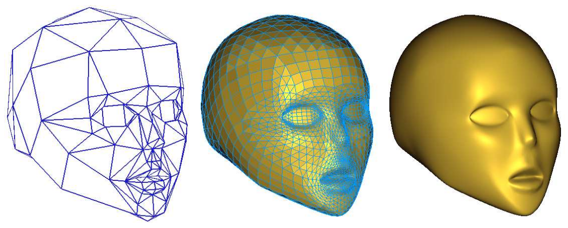

Figure 2: *from left to right: Input mesh, faceted refined input mesh and smooth rendering of a subdivision surface.*

and filling shortest edge loops with (bi)linear facets (Figure 2). *Mesh mutation* combines mesh refinement (insertion of nodes) and mesh attribute adjustment, for example modification of node position (Figure 1).

mesh	mesh mutation op
connectivity	refinement
attributes	adjustment

Mesh mutation is not user-interactive, as opposed to mesh manipulation in content-creation or Computer Aided Design packages. Transforming user intent into a high-quality graphical representation can therefore be viewed as a 3-step process: user-interaction, mesh mutation and mesh rendering. These mesh operations can be performed on the CPU or on the GPU. At present, operations and layers are associated as follows:

† sle-jeng@cise.ufl.edu
‡ vgoel@ati.com
§ jorg@cise.ufl.edu

mesh op	layer (to date)
user-interaction	CPU
mesh mutation	CPU
mesh rendering	GPU

The main reason for this distribution of work, despite the availability of vertex shaders and ample video memory in the latest hardware generation, is the need to access a mesh with *irregular connectivity*. Current hardware does not support such access. This is unfortunate since, real time mutation on graphics hardware is easily within reach: refining and adjusting a 1000-node input mesh with four attributes per node four times costs ca. 5 million ALU instructions; in the near future, SIMD hardware will be able to serve several thousand times that many vector operations per second.

One approach to redistributing work is to simplify the mesh and the mesh operations. For example, Gu et al.[10] and Losasso et al.[16] approximate the attributes and connectivity of the mesh and re-represent it as a rectangular array. This regularizes access and facilitates application of image processing techniques. Bolz et al. [5] propose to precompute (position) attributes of a number of important meshes for a fixed refinement level and load the result into video memory. This has the advantage that nodes need not visit neighbors to compute these attributes. However, simplifying the mesh representation and mesh operations comes at a cost. Complex adjustments, such as Pixar's crease rules[7] cannot all be pre-tabulated; geometry dependence at each level of refinement (in variational subdivision) cannot be pre-tabulated; there is a cost associated with storing and accessing large tables; approximation of surfaces as functions over a single domain (of a different topological genus) leads to distortions and shape artifacts.

A different approach for exploiting SIMD parallelism without sacrificing general meshes and mesh operations becomes practical as GPUs expose read and write access to random locations in video memory. The **contribution** of this paper is *a framework for mutation and rendering of complex, irregular meshes* that can be implemented as a *(mesh processor, mesh shader)* pair on such graphics hardware. The mesh shader implements the adjustment stage of mesh mutation and has localized access to neighbor nodes in the mesh. We call this framework 'Mesh in Programmable Graphics Hardware', short MiPGH. MiPGH moves mesh mutation into the GPU layer:

mesh op	layer (future)
user-interaction	CPU
mesh mutation	GPU
mesh rendering	GPU

The main advantages of redistributing the work to the GPU layer are

- decreasing the system bus traffic;
- increasing the speed through SIMD architecture.

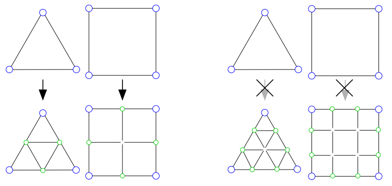

Figure 3: *(left two) Four-split corresponds to binary refinement along boundaries. (right two) Ternary refinement is not supported by the MiPGH implementation in Section 3.1.*

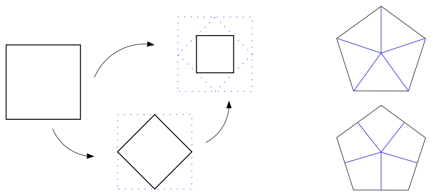

Figure 4: *(left) A complex operation can be broken into a sequence of simpler operations (here averaging). (right) Two ways to split an n-gon into quads or triangles.*

A side-effect is the user-transparent implementation of complex but basic graph access and refinement structures. This is not only convenient but raises the level of abstraction and allows the application program to concentrate on the crucial geometric adjustment operations.

Refinement and node visit in MiPGH are user-transparent, and adjustment is user-programmable in the shader program. The attributes themselves are generic, i.e. can be defined in the CPU level graphics program by binding them to one of several general addressing mechanisms (see Section 4).

MiPGH op	level
attribute definition	user program
refinement & visit	mesh processor
adjustment	mesh shader

For example, the details of the splitting of every quadrilateral mesh facet into four quads (Figure 3, *left*) are user-transparent, while the specification of the subdivision stencil for Catmull-Clark (see Section 2.1) is part of a short, user-programmable shader routine. The user's graphics program might define the attribute 'CCnormal', bind it to the addressing mechanism 'node-morphic' and the rendering type 'normal' interpreted by the mesh processor.

MiPGH is conceived to work for a general mesh model. To leverage the SIMD architecture to the full extend and cover the mesh mutations of practical interest *without adjacency pointer support*, the structures proposed in Section 3.1 restrict mesh refinement and node visit to

R1 input meshes consisting of only quadrilaterals (*quad*s) and triangles;

R2 primal refinement that splits every facet into four (Figure 3);

R3 mesh shaders that access only the direct neighborhood of the node (Figure 6).

The last restriction, R3, is less severe than it appears at first sight: many complex adjustments, subdivision operations in particular, factor into a sequence of simpler operations (Figure 4, *left*); and multipass adjustment without refinement is covered by the MiPGH framework. Also, the first restriction, R1, is a common in mesh design: a single refinement step can break n-gons into triangles or quads (Figure 4, *right*). Under these conditions, MiPGH has the following properties.

- *Multi-parallel* computation: by minimizing dependencies, parallelism for node operations is fully exploited.
- *Generic attribute definition* allows for user-programmable attributes.
- *User-programmable* adjustment rules.
- *User-transparent node visit*: the mesh processor visits all nodes of the mesh.
- *Light-weight attribute indexing*: the mesh shader has a simplified view of the local submesh (Figure 24) supported by simple attribute addressing in the mesh processor.
- Support for *Level-of-detail*.
- *Unique attribute storage*: attribute addressing via indices (plus a base) avoids inconsistent floating point representations that could cause cracks in the tessellation.

Figures 2, 22 and 23 are the result of a software simulation of the anticipated architecture. Section 5 gives examples of the mesh shaders used.

2. Background and Scope

An important application of mesh mutation to date is the generation of smooth surfaces by generalized subdivision. Below, we review only those aspects of this rich subject[21] (and of 3D adjacency data structures and current programmable graphics hardware) that are relevant for understanding the use and scope of MiPGH.

2.1. Subdivision Surfaces

Subdivision algorithms can be formulated as a series of mesh mutation steps with each step consisting of refinement followed by mesh smoothing. If refinement splits each facet into four (see R2 and Figure 3, *left*), we call the refinement *binary* as opposed to, say, ternary (Figure 3, *right*). If the refinement associates a new node with every old node (and inserts nodes along edges and in the facets), the scheme is called *primal*. If every new node is defined as a weighted combination of old nodes that lie on (old) facets with a common (old) node, the scheme acts on a *1-disk* (R3). The combination of old nodes is often succinctly and graphically expressed as a *stencil* that displays a portion of the old graph

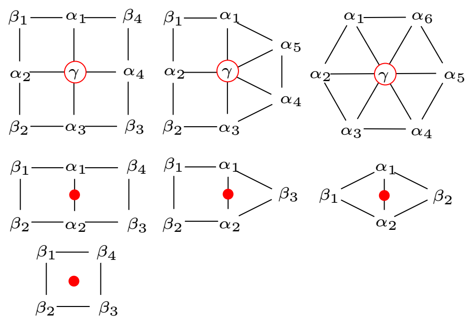
β_1 α_1 β_4 α_2 γ α_4 β_2 α_3 β_3
β_1 α_1 α_5 α_2 γ α_4 β_2 α_3
α_1 α_6 α_2 γ α_5 α_3 α_4
β_1 α_1 β_4 β_2 α_2 β_3
β_1 α_1 β_3 β_2 α_2
α_1 β_1 β_2 α_2
β_1 β_4 β_2 β_3

Figure 6: *The subdivision stencils of the generic 1-disk, binary, primal subdivision scheme.*

with the weights (that must sum up to 1) in place of the nodes; the resulting new node is indicated either as a circle, if it corresponds to an old node, or as a •, if it lies on an edge or in a facet (Figure 6). If the stencil does not change with the refinement level, the scheme is stationary. *Anisotropic* rules adjust the stencil according to tags placed by the user (e.g. Pixar's crease rules [7]) or by evaluating the attributes of the mesh neighborhood.

By providing a framework for all *primary, binary, non-stationary* (and therefore also for stationary), *anisotropic* (and therefore also isotropic) *tagged, 1-disk subdivision* schemes on *quad and triangle* meshes (Figure 6), MiPGH covers only a subset of all theoretically possible refinement and smoothing strategies; but, arguably, it covers all subdivision schemes one encounters in current modeling practice: Catmull-Clark [6] and Loop [15] subdivision, (bi)linear subdivision, 4-8 subdivision [20], 'butterfly' interpolatory subdivision [8] and even subdivision schemes that mix quads and triangles [19]. Variants of the above, i.e. alternative stencils are easily realized as user-programmable *mesh shader* code.

Hardware specific implementations and rendering of subdivision schemes can be found in [2, 3, 17].

2.2. Adjacency Data Structures

In software, we can choose between several different data structures to support mesh mutation on (orientable) polyhedral meshes. Edge based data structures [22] are commonly used and allow for maximal flexibility. Given our primary interest in structured, namely primal, binary, four-split mesh refinement, they are unnecessarily complex. A representation as a forest of quad-trees (with one tree per mesh facet) [23] is useful for structured, adaptive refinement but the neighbor access patterns appear to be too complex for efficient SIMD realization. All of the above structures require extensive manipulation of the adjacency pointers throughout the refinement increasing the complexity of the hardware design. MiPGH treats the units of structured refinement as *patches* and provides only limited but sufficient *static* adja-

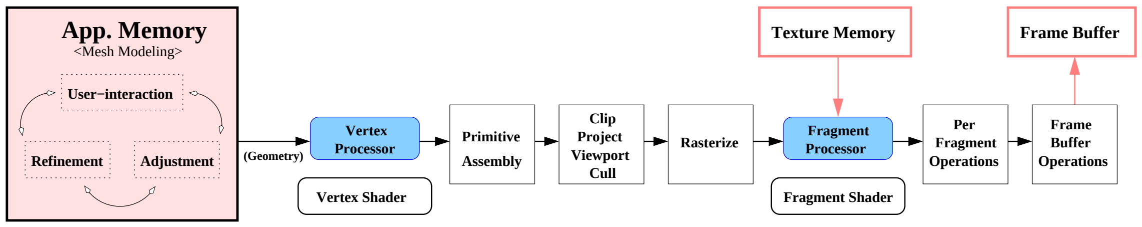

Figure 5: *Current: Mesh mutation and the graphics pipeline.*

cency traversal of the mesh based on an index-based model (Figure 9).

2.3. Programmable Graphics Hardware

Shaders are a way of exposing the SIMD processing capabilities of GPUs to an application program[14]. Shaders are programmable stages in the state machine and are concurrently executed either on the vertex processor or the fragment processor. As Figure 5 makes clear, mesh mutation by the CPU results in large data streams (each refinement quadruples the data) to be sent to the GPU via the system bus. While the primary motivation for creating the shader concept was to improve rendering quality, shaders have recently been proposed to serve as parallel processes for numerical computation [4, 13]. Therefore, exposing a **mesh shader** capability, would benefit rendering, numerical computation and mesh processing.

3. Mesh Mutation Framework

In this section, we place mesh mutation in the graphics pipeline and develop a *model* of mesh mutation on primal, binary-refined quad-triangle meshes. This model is made concrete by structures and visitation patterns that fit the SIMD paradigm of GPUs. Attribute lists and access are discussed in detail in Section 4.

3.1. Mesh Mutation in the Graphics Pipeline

Figures 7 and 8 summarize the proposed architecture. The mesh processor initializes the connectivity (see Initialization below) and the attributes from the *geometry and topology* stream. A single step of mesh mutation includes three stages: refinement, node visit and adjustment. The **refinement** can be bypassed; in that case, the mesh processor only allocates attribute memory and swaps the attribute handle(s) after target adjustment. The mesh processor **visits each node** of the mesh and collects the corresponding 1-disk submesh data. We say *visit* rather than traversal since this operation may be executed in parallel. Nodes are visited in the order: D-facets, D-edges, D-vertices. This ensures locality of the attribute access. **Adjustment** is a user-programmable unit. The user program has access to exactly a 1-disk (Figure 24) through

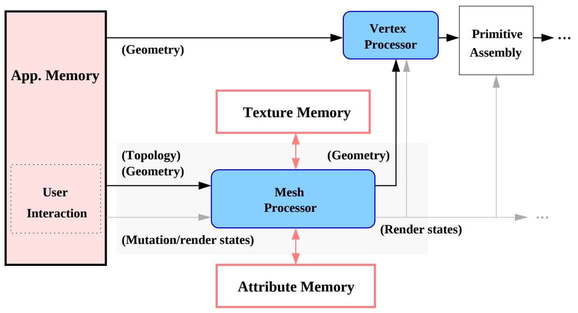

Figure 7: *Proposed: Mesh mutation and the graphics pipeline with a mesh processor (see Figure 8). Note that the notion of processor is conceptual, indicating a user-transparent unit. It can be realized in microcode rather than dedicated hardware.*

the *local* indices which enumerate the submesh. The 1-disk view hides the complexity of the traversal and access of the global mesh. After several iterations of the mesh mutation, the mesh processor *compiles connectivity and attributes* of the mesh to form primitives for the lower pipeline. We use ‘compile’ rather than ‘assemble’, since the mesh processor has to interpret the attribute type and the rendering type. We now develop on the model underlying the data access in MiPGH.

3.2. Connectivity and the Domain Atlas

The polyhedral input mesh is a discrete analogue of an oriented 2-manifold, possibly with global boundary. In this analogy, the charts that define the manifold consist of one (or more, see Initialization below) quad or triangle. Only edges of adjacent charts overlap, i.e. the integer indices pointing to their attributes are replicated in both charts that share the edge. Corner indices are replicated in n charts (Figure 9, left). All charts together form the domain atlas, short D-atlas.

At each level of refinement, the chart is made concrete as a **D-facet**. A D-facet is an array of integer indices allow lookup of the attributes associated with the node (Figure 10). At refinement level σ ($\sigma = 0$ for a single facet of the input mesh), a D-facet consists of $2^\sigma \times 2^\sigma$ **micro-facet**s, i.e.

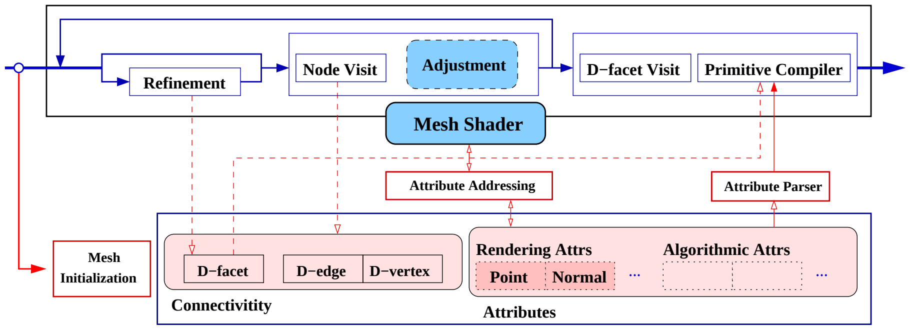

Figure 8: *Mesh processor and data access. Data arrive from the system bus and depart to the rendering pipeline.*

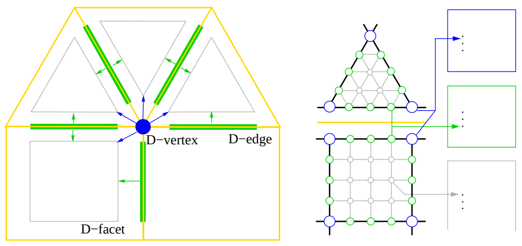

Figure 9: *(Left) The D-atlas represents the connectivity of the input mesh. Arrows indicate access directions. For example, a D-vertex can access the corners of the surrounding charts. After initialization, the D-atlas of MiPGH does not change. (Right) The nodes of a D-LoD after two refinement steps. Replicated boundary nodes point to the same attribute container for* unique *storage of attributes (including position).*

small quad or triangle facets. Each D-facet shares its boundary indices with an adjacent D-facet. The D-facet has no connectivity information to access other D-facets, D-edges or D-vertices(see Figure 16). The collection of D-facets of a given chart for all refinement levels, form the **D-LoD** of the chart. The D-LoD supports level-of-detail (at a cost of just 4/3 times the storage of the largest D-facet.) Chart edges are represented in the D-atlas as **D-edge**s. A D-edge consists of two *halfedges*. Each halfedge can access the adjacent D-facet or is null at a global boundary (Figure 11). Nodes along a D-edge are visited via the D-facet index plus the corner index and then stepping to the next node by equal increment. A (boundary or shared) chart corner is represented as a **D-vertex**. A D-vertex of valence n can access the corners of the n abutting D-facets (Figure 12). The D-atlas remains fixed during mesh mutation. Only the D-LoD grows with each refinement and the attribute lists grow correspondingly.

Initialization, localized update and adaptive refinement. The D-atlas is initialized by the geometry and connectivity information of a vertex array describing the polyhedron. Analogous to an evaluator, the mesh is mutated according to a user-defined sequence of adjustments and refinements. Parts or all of intermediate meshes can be returned to the CPU level with a system call. Such a refined mesh and refined mesh pieces in general can be (re)initialized by packing $2^\sigma \times 2^\sigma$ facets into a single D-facet at subdivision level σ. This initialization is correctly treated as part of the adaptive refinement implementation described below and can support numerous applications, like cut-and- paste [9, 1].

One strategy for generating adaptive meshes is to subsample a uniformly refined mesh. However, in large models unnecessary refinement causes substantial computation and storage overhead. In MiPGH, D-LoDs are labeled with the number of refinements (Figure13), either explicitly at the outset by an application program or algorithm evaluated inside the user-programmable mesh shader. The mesh processor will not visit the nodes or allocate memory for the attributes of a D-facet whose D-LoD-label is less than the current refinement counter. Edges separating D-LoDs that have different labels are correctly updated by maintaining a single auxiliary layer of nodes beyond the boundary (Figure 14). *Interactive* deformation or animation of mesh D-vertices is supported by resetting the tags of the affected D-LoDs. Correct rendering without cracks of the adaptively refined D-LoDs is guaranteed by a primitive compilation (Figure 8) that has full knowledge of the relative refinement and the rendering state.

4. Generic Attributes and Access

MiPGH (see Figure 15) consists of three *connectivity* lists, the D-LoD list, the D-edge list and the D-vertex list and several *attribute* lists. Since we assume read and write capability to random memory locations, the attributes need not be fixed a priori but can be *generic*. That is, the graphics program can define any attribute that uses one of the addressing modes described below, by binding it to an attribute type and an addressing mode. The addressing mode determines the

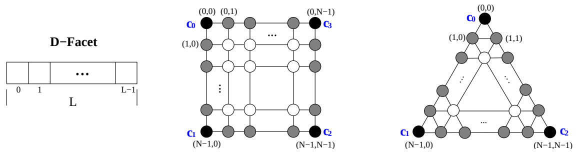

Figure 10: *A D-facet. Each D-facet is a* row major *quadrilateral or triangular array containing, for each node, the integer index of the attribute storage location. (Left) With* $N := 2^\sigma + 1$, $L := s_q = N \times N$ *for a quad and* $L := s_t = \binom{N+1}{2}$ *for a triangle (as an optimization, we can pack two adjacent triangles into a quad); (Middle and right) Black circles depict nodes that correspond to D-vertices, gray circles correspond to nodes on D-edges;* $c_0 \ldots c_3$ *are the corner indices arranged in counterclockwise (CCW) order.*

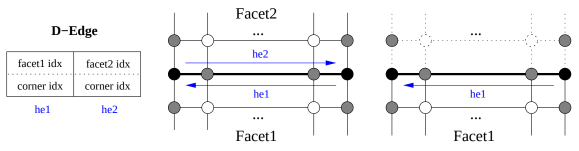

Figure 11: *A D-edge. (Left) Each D-edge consists of a pair of halfedges. A halfedge is defined by the index of the incident D-facet and the index of the pointing corner. (Middle) The first halfedge is designated as the* major *halfedge and defines the orientation of the edge. (circle shading as in Figure 10) (Right) On a global boundary, the second halfedge is* null.

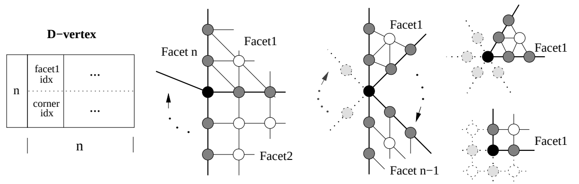

Figure 12: *A D-vertex. (Left) A D-vertex of valence* n *is stored as the integer* n *followed by* n *index pairs each representing, in CW order, the index and the corner index of D-facets. (Right) A D-vertex in the interior, on the global boundary and at a global corner of the input mesh.*

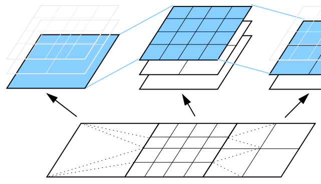

Figure 13: *Adaptive mesh as selection of D-facets at different D-LoD-levels.*

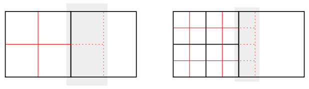

Figure 14: *Adaptive refinement uses a single auxiliary layer along boundaries.*

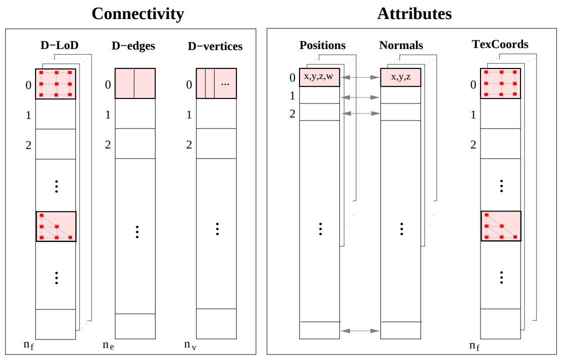

Figure 15: *MiPGH has three connectivity lists (D-facet, D-edge and D-vertex) and several attribute lists. Here three possible (rendering) attributes are shown – more or fewer can be defined in the graphics routine and bound to the shader. The input mesh has* n_v *vertices,* n_e *edges and* n_f *facets.*

memory allocation for the attribute list. We distinguish two attribute addressing modes: indexed and hashed. Indexed means that the attribute addresses are computed from a base address plus an offset index listed in the D-facet. Hashed means, it is computed based on the regular correspondence of the attributes to those indices. Hashed addressing comes in many flavors.

For example, all attributes associated one-to-one with nodes (node-morphic) are indexed (Figure 17). Along D-facet edges, the unique index guarantees unique storage of attributes. Since node-morphic data is stored in the order: corners, edges, interior nodes, we can compute the parent D-facet, D-edge and D-vertex from any point index of the refined mesh.

By contrast, there may be two different texture coordinates associated with a single node on an D-edge, one for each D-facet. Since a D-facet has as many nodes as texture coordinates and they are arranged in the same order (Figure 18), the hashing function for such chart-morphic data is simply the texture coordinate base plus the node offset within the D-facet.

Some mesh algorithms need to tag the n edges emanating from a node. Rather than storing such data with the node, and provision for an arbitrary number n of neighbors, we associate the attributes with the micro-facet, since a micro-facet always has either 3 or 4 corners (Figure 19). By associating each micro-facet corner with two values, one for the arriving halfedge and one for the departing halfedge (in CCW

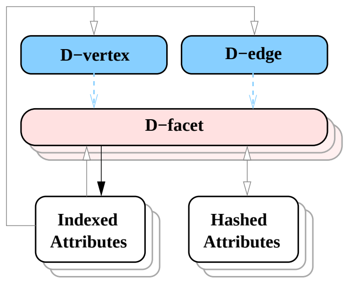

Figure 16: *Data access by shader and processor in MiPGH. Dashed arrows (from D-vertex and D-edge to D-facet) represent access encoded in the D-atlas, the solid black arrow represents index-based addressing of attributes and empty arrows indicate hash-based addressing.*

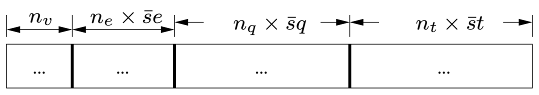

Figure 17: *A node-morphic attribute list. Here $N := 2^\sigma + 1$ for σ refinements, $\bar{s}_e = N - 2$, the number of nodes on a D-edge, $\bar{s}_q = (\bar{s}_e)^2$ the nodes inside a quad D-facet and $\bar{s}_t = \binom{\bar{s}_e}{2}$ inside a tri D-facet.*

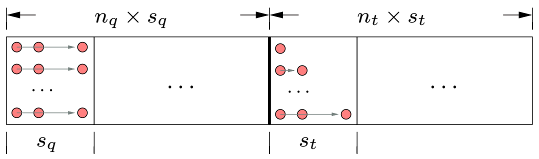

Figure 18: *A chart-morphic attribute list. Here $s_q = N^2$, the number of nodes of a quad D-facet and $s_t = \binom{N+1}{2}$ of a tri D-facet.*

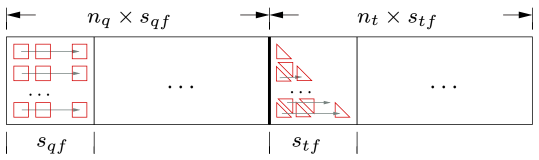

Figure 19: *A micro-facet-morphic attribute list. Here $s_{qf} = s_{tf} = (N-1)^2$ is the number of the micro-facets of the D-facet.*

order), each micro-facet has either eight or six values associated and each edge has four associated values. Implementing Pixar's 'semismooth crease' rules [7] then corresponds to the special case where two pairs of the four edge values are equal. Figure 16 summarizes the attributes addressing and the adjacency access patterns.

Memory Access. Each type of attribute data is allocated sequentially in a contiguous array by giving the base address and the size. Threading eliminates the latency caused by the indirect (base+offset) memory access. Also, the attributes of

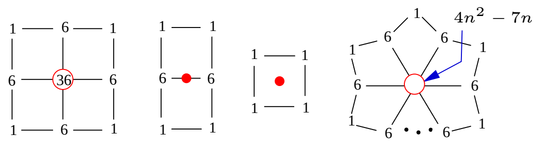

Figure 20: *Unnormalized Catmull-Clark subdivision stencils; n is the valence.*

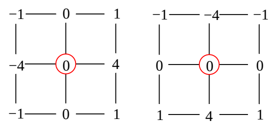

Figure 21: *The CC derivative stencils.*

each D-facet are clustered as illustrated in Figures 17, 18 and 19. From the users point of view, the attributes are accessed *randomly* via a local index into the 1-disk. This transparency can be achieved since, in a mesh shader, the base address is available as an attribute handle and the offset is an index maintained by the mesh processor.

Attributes compilation and rendering. Attributes are either *algorithmic* or *bound to a rendering type* such as position, normal or node color. Algorithmic attributes support computations in the mesh shader or can be channeled to the vertex shader for computations. For rendering attributes, based on the type, the attribute parser and primitive compiler (Figure 8) encodes the rendering state for the stream of the primitives, for example the color of each facet. Vertex shaders or pixel shaders can also serve as rendering types. Partial meshes can thereby be associated with different shaders, say to apply a different shader to the eye region in Figure 2 than to the mouth.

5. Examples of Mesh Mutation

This section presents two examples of user-programmable mesh shaders using MiPGH. The pseudocode bases on the mechanism of the OpenGL shader language [12] and the extensions of the Appendix. Due to space constraints, we do not show code for creating the program object or shader linking. The first example illustrates Catmull-Clark subdivision with MiPGH. Figure 20 shows the subdivision stencil.

Catmull-Clark (CC) Subdivision

```
void main(void) {
// Position is the pre-defined variable of the position
// of the central point in the 1-disk.
  if (MeshType != MT_EOV) {
    const float fvmask[4] =
      {1/4.0, 1/4.0, 1/4.0, 1/4.0};
    const float evmask[6] =
      {6/16.0, 6/16.0, 1/16.0, 1/16.0, 1/16.0, 1/16.0};
    const float vvmask[9] =
      {36/64.0,
       6/64.0, 6/64.0, 6/64.0, 6/64.0,
       1/64.0, 1/64.0, 1/64.0, 1/64.0 };
    vec4 mesh[9];
    if (MeshType == MT_FV) {// Facet-vertex
      readAttribute4(IPositionHandle,
                     IVertexIndices,
                     mesh, 4);
      Position = inner_product(mesh, fvmask, 4);
    } else if (MeshType == MT_EV) {// Edge-vertex
      readAttribute4(IPositionHandle,
                     IVertexIndices,
                     mesh, 6);
      Position = inner_product(mesh, evmask, 6);
    } else if (MeshType == MT_VV) {// Vertex-vertex
      readAttribute4(IPositionHandle,
                     IVertexIndices,
                     mesh, 9);
      Position = inner_product(mesh, mask, 9);
    }
  } else {// extraordinary vertex-vertex
    const int mash_size = 1+2*MaxValence;
    vec4 mesh[mesh_size];
    readAttribute4(IPositionHandle,
                   IVertexIndices,
                   mesh, mesh_size);
    float cw = 4*Valence*Valence - 7*Valence;
    float w = cw + 6*Valence + Valence;;
    float mask[mesh_size];
    mask[0] = cw/w;
    for (int i = 1; i <= max_valence ; ++i) {
      mask[i] = 6/w;
      mask[i+max_valence] = 1/w;
    }
    Position = inner_product(mesh, mask, mesh_size);
  }
}
```

Our second example computes the normal of Catmull-Clark limit surface, for regular nodes, as the cross product of the tangents generated by applying the stencils of Figure 21 (see [11] for the general case). Then the shader displaces the mesh node in the normal direction by an amount specified by a texture. The refinement step is bypassed in this example. The result is shown in Figure 22.

Normal Displacement

```
void main(void) {
// Normal is the pre-defined variable of the normal
// of the central point in the 1-disk.
  vec4 mesh[9];
  readAttribute4(IPositionHandle,
                 IVertexIndices,
                 mesh, 9);
  const float t1_mask[9] = { 0,  0, -4, 0, 4,
                             -1, -1, 1, 1};
  const float t2_mask[9] = { 0, -4, 0, 4,  0,
                             -1, 1, 1, -1};
  vec3 t1 = inner_product(mesh, t1_mask, 9);
  vec3 t2 = inner_product(mesh, t2_mask, 9);
  Normal = normalize(cross(t1, t2));

  vec2 coord = readAttribute2(ITexCoordHandle,
                              IVertexIndicex[0],
                              IFacetIndicators[0],
                              ILevel,
                              AT_CHART_MORPHIC);
  vec4 color = texture2D(tex_sampler, coord);
  Position = mesh[0] + color.r * Normal;
}
```

6. Conclusion

The mesh processor framework proposed in this paper strikes a balance between the SIMD characteristics of the GPU and fully general mesh access and refinement. Although our prime example is subdivision, the mesh processor is more general than just an implementation of subdivision surfaces (see Figures 22 and 23). The framework is anchored in the well-understood and accepted theory of manifolds and the concepts of atlas and chart. In particular, the notion of edge and facet complement the primary graphics programming concept of vertex. Ultimately, we envision that these notions will be mapped onto abstract, but SIMD-aware interface definitions accepted by the community so that *refinement and node visit*, too, become user-programmable.

References

1. Henning Biermann, Ioana Martin, Fausto Bernardini, and Denis Zorin. Cut-and-paste editing of multiresolution surfaces. In John Hughes, editor, *SIGGRAPH 2002 Conference Proceedings*, pages 312–321, 2002.
2. Stephan Bischoff, Leif P. Kobbelt, and Hans-Peter Seidel. Towards hardware implementation of loop subdivision. In *Proceedings 2000 SIGGRAPH/EUROGRAPHICS workshop on on Graphics hardware*, pages 41–50. ACM Press, 2000.
3. M. Bo, M. Amor, M. Doggett, J. Hirche, and W. Strasser. Hardware support for adaptive subdivision surface rendering. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on on Graphics hardware*, pages 33–40. ACM Press, 2001.
4. Jeffrey Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse matrix solvers on the gpu: Conjugate gradients and multigrid. In Jessica Hodgins, editor, *Siggraph 2003, Computer Graphics Proceedings*, pages xx–xx, 2003.
5. Jeffrey Bolz and Peter Schröder. Rapid evaluation of catmull-clark subdivision surfaces. In *Proceedings of the Web3D 2002 Symposium (WEB3D-02)*, pages 11–18, New York, February 24–28 2002. ACM Press.
6. E. Catmull and J. Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer Aided Design*, 10:350–355, 1978.
7. Tony DeRose, Michael Kass, and Tien Truong. Subdivision surfaces in character animation. In Michael Cohen, editor, *Siggraph 1998, Computer Graphics Proceedings*, pages 85–94, 1998.
8. Nira Dyn, David Levine, and John A. Gregory. A butterfly subdivision scheme for surface interpolation with tension control. *ACM Transactions on Graphics (TOG)*, 9(2):160–169, 1990.
9. C. Gonzalez and J. Peters. Localized hierarchy surface splines. In S.N. Spencer J. Rossignac, editor, *ACM Symposium on Interactive 3D Graphics*, 1999.
10. Xianfeng Gu, Steven J. Gortler, and Hugues Hoppe.

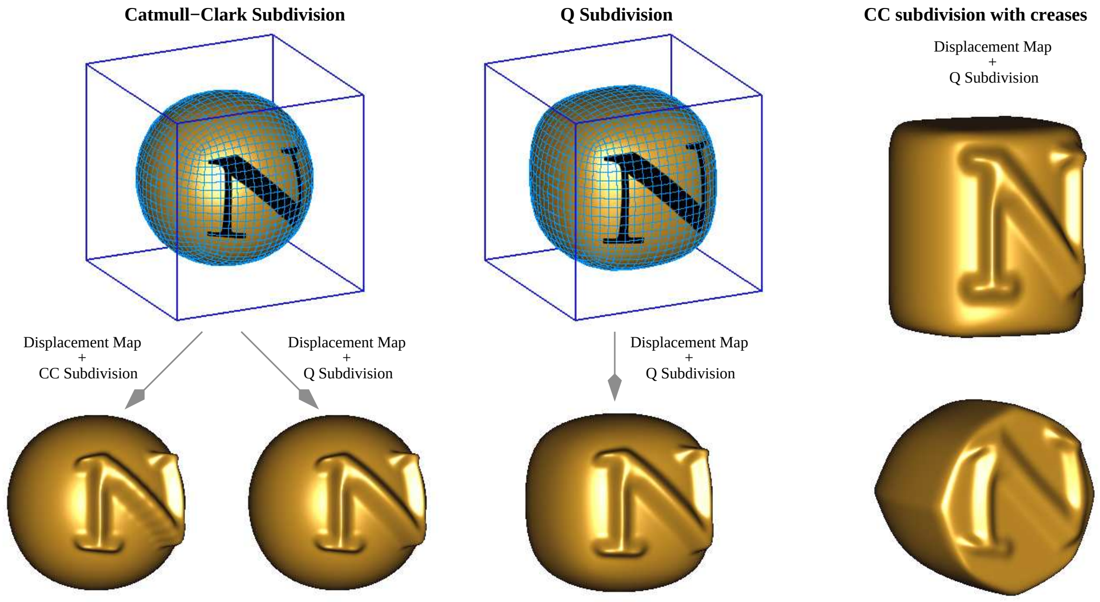
Catmull−Clark Subdivision
Q Subdivision
CC subdivision with creases
Displacement Map + Q Subdivision
Displacement Map + CC Subdivision
Displacement Map + Q Subdivision
Displacement Map + Q Subdivision

Figure 22: *Different shaders (CC-subdivision, Q-subdivision, CC-subdivision with creases, normal_displacement) applied at different levels.*

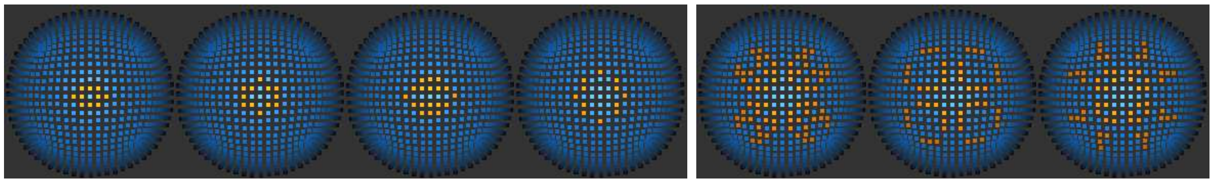

Figure 23: *Snapshots from a game of life simulation on a CC-subdivided mesh. The tripple on the right shows a period-3 'pulsar'. The 'life' simulation corresponds to repeated adjustment without refinement.*

Geometry images. In John Hughes, editor, *SIGGRAPH 2002 Conference Proceedings*, Annual Conference Series, pages 335–361. ACM Press/ACM SIGGRAPH, 2002.

11. Mark Halstead, Michael Kass, and Tony DeRose. Efficient, fair interpolation using catmull-clark surfaces. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 35–44. ACM Press, 1993.
12. D. Baldwin J. Kessenich and R. Rost. Opengl shading language specification ver1.05. Technical report, February 2003.
13. Jens Krüger and Rüdiger Westermann. A framework for numerical simulation techniques on graphics hardware. In Jessica Hodgins, editor, *Siggraph 2003, Computer Graphics Proceedings*, pages xx–xx, 2003.
14. Erik Lindholm, Mark J. Kligard, and Henry Moreton. A user-programmable vertex engine. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 149–158. ACM Press, 2001.
15. Charles T. Loop. Smooth subdivision surfaces based on triangles, 1987. Master's Thesis, Department of Mathematics, University of Utah.
16. F. Losasso, H. Hoppe, S. Schaefer, and J. Warren. Smooth geometry images. In H. Hoppe L. Kobbelt, P. Schröder, editor, *Eurographics Symposium on Geometry Processing*, 2003.
17. Kari Pulli and Mark Segal. Fast rendering of subdivision surfaces. In *Proceedings of the eurographics workshop on Rendering techniques '96*, pages 61–70. Springer-Verlag, 1996.
18. R. Rost. Opengl 2.0 overview ver1.2 (whitepaper). Technical report, February 2002.

19. Jos Stam and Charles Loop. Quad/triangle subdivision. *Computer Graphics Forum*, 22(1):1–7, 2003.
20. Luiz Velho and Denis Zorin. 4-8 subdivision. *Computer Aided Geometric Design*, 18(5):397–427, June 2001.
21. Joe Warren and Henrik Weimer. *Subdivision Methods for Geometric Design*. Morgan Kaufmann Publishers, New York, 2002.
22. Kevin Weiler. Edge-based data structures for solid modeling in curved-surface environments. *IEEE Computer Graphics and Applications*, 5(1):21–40, January 1985.
23. D. Zorin. Implementing subdivision and multiresolution meshes. *Chapter 6 of Course notes 37 of SIGGRAPH 99*, 1999.

Appendix A: Mesh Shader Variables and Functions based on the OpenGL Shading Language

While a vertex shader [12, 18] operates on an isolated vertex, a mesh shader operates a 1-disk submesh via built-in variables (in a pre-defined arrangement) and built-in functions. The mesh shader receives two groups of attribute handles and mesh variables from the mesh processor: a 1-disk source submesh and a 1-disk target submesh. The submesh nodes are explicitly indexed as shown in Figure 24. The submesh of a node with n neighbors has the same index pattern, listing, in CCW order, first the direct neighbors and then the diagonal neighbors. The rendering attributes are bound to rendering states during initialization of the shader object.

Variable	**C**omment
Attributes Handles (user-defined)	
uniform attr1D [I/O]RTypeHandle	[I/O]: source or target mesh RType can be Position, Normal ..etc
uniform attr1D [I/O]AttributeHandle[0/1/2/..]	algorithmic attributes [0/1/2/..]
Mesh Variables (built-in)	
uniform int [I/O]VertexIndices[]	
uniform ivec4 [I/O]FacetIndicators[]	[idx,corner idx,4/3,D-facet idx]
uniform int [I/O]Level	refinement level
uniform int MeshType	of the source mesh
uniform int Valence	of the target mesh
Output Vertex (built-in)	
varying vec4 Position	of the target center vertex
varying vec4 Normal	of the target center vertex

The mesh shader addresses the *generic* attributes with a set of built-in functions. These functions are classified by their addressing mode, indexed or hashed as explained in Section 4. of this paper. For indexed attributes, the prototype addressing functions are

```
vec readAttribute(attr1D attr_handle, int idx)
vec writeAttribute(attr1D attr_handle, int idx, vec attr)
```

The full function names are to indicate the output size. For

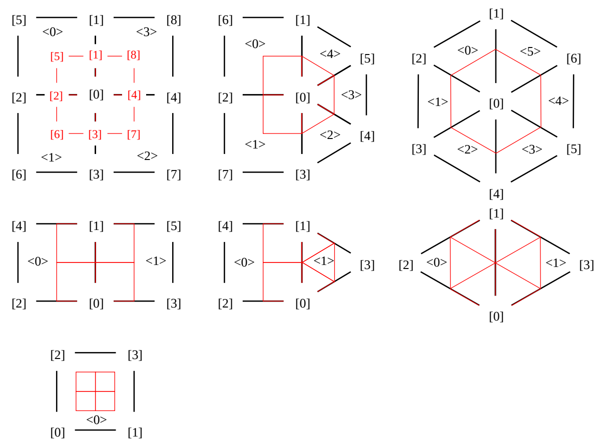

Figure 24: *A mesh shader's view of the enumeration of (regular) 1-disks: [ ]s indicate the the nodes and <>s the facets. Black indicates the input 1-disk and red the output. If the refinement step is bypassed, only the vertex-vertex 1-disk is passed to the shader.*

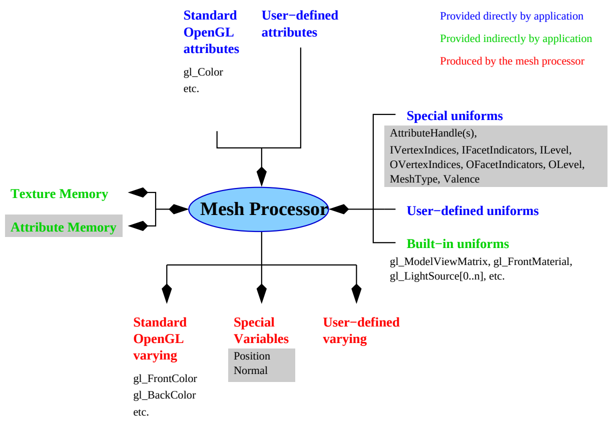
Standard OpenGL attributes
gl_Color etc.
User−defined attributes
Provided directly by application
Provided indirectly by application
Produced by the mesh processor
Special uniforms
AttributeHandle(s), IVertexIndices, IFacetIndicators, ILevel, OVertexIndices, OFacetIndicators, OLevel, MeshType, Valence
Texture Memory
Attribute Memory
Mesh Processor
User−defined uniforms
Built−in uniforms
gl_ModelViewMatrix, gl_FrontMaterial, gl_LightSource[0..n], etc.
Standard OpenGL varying
gl_FrontColor gl_BackColor etc.
Special Variables
Position Normal
User−defined varying

Figure 25: *The mesh shader as an extension of the vertex shader in the OpenGL shader specification.*

example, the function for reading position=(x,y,z,w) is vec4 readAttribute4(). For hashed attributes, the hashing paramenters and the hash mode, for example chart-morphic, is passed to the mesh processor. The prototype hash functions are as follows.

```
vec readAttribute(attr1D attr_handle, int idx, ivec4 indicator, int mode)
vec writeAttribute(attr1D attr_handle, int idx, ivec4 indicator, int mode, vec attr)
```

To speed up access, array access functions are also provided. For example, the following functions apply to indexed attributes.

```
void readAttribute(attr1D attr_handle, int[] indices, vec[] attrs, int size)
void writeAttribute(attr1D attr_handle, int[] indices, vec[] attrs, int size)
```