Multiresolution GPU Mesh Painting

Tobias Ritschel[1] Mario Botsch[2] Stefan Müller[1]

[1]Institute for Computational Visualistics, University Koblenz-Landau, Germany
[2]Computer Graphics Laboratory, ETH Zurich

Abstract
Mesh painting is a well accepted and very intuitive metaphor for adding high-resolution detail to a given 3D model: Using a brush interface, the designer simply paints fine-scale texture or geometry information onto the surface. In this paper we propose a fully GPU-accelerated mesh painting technique, which provides real-time feedback even for highly complex meshes. Our method can handle arbitrary input meshes, which are considered as base meshes for Catmull-Clark subdivision. Representing the surface by an atlas of geometry images and exploiting programmable vertex and fragment shaders allows for highly efficient LoD rendering and surface manipulation. Our painting metaphor supports real-time texturing, sculpting, smoothing, and multiresolution surface deformations.

1. Introduction

Adding high resolution geometric or color detail to a given surface is an important problem in 3D content creation. Especially in this context, where surface manipulations are more artistic rather than engineering-like, an intuitive user interface is required to support the designer's creativity. This has recently led to several commercial applications based on the well-accepted mesh painting metaphor. To modify a given high resolution mesh, the designer simply uses a brush of adjustable size to directly paint the transformations onto the model's surface. The transformations itself can be as diverse as texture or color painting, local mesh smoothing, carving, or sculpting.

Painting color or texture information onto meshes was first introduced by Hanrahan and Häberli [HH90], and later extended to painting geometric details or surface deformations [ZPKG02, LF03]. It is clear that real-time feedback is crucial for this kind of interactive mesh manipulation, but on the other hand it is also difficult to achieve for highly detailed models, which often results in inacceptable latencies.

In this paper we propose to exploit the computational power of modern graphics hardware (GPUs) not only for surface rendering, but also for painting-based surface manipulation, which eventually allows for interactive editing even of models consisting of a few millions of triangles. Recent approaches started to exploit GPUs for painting textures or colors, representing the surface either by a multiresolution atlas of charts [CH04] or by an adaptive octree [LKS*06]. However, these spatial data structures will be difficult to maintain if the surface geometry (in addition to color) is modified.

We therefore propose a surface representation based on an atlas of geometry images, similar to [SWG*03]. The regular structure of geometry images [GGH02] allows for efficient GPU processing, as was shown for smooth subdivision of a single chart geometry image in [LHSW03]. While [SWG*03] partitioned the input mesh into charts that have to be zippered later on, we rather consider the input mesh as the coarse domain for Catmull-Clark subdivision, which naturally yields a piecewise geometry image, as shown in the next section.

2. Surface Representation

The input model is an irregular polygonal mesh of arbitrary genus, which after one Catmull-Clark subdivision step consists solely of quads (Fig. 1, left). After a few, say k, additional subdivision steps each of these quads will be refined to a completely regular patch of $2^k \times 2^k$ quads, such that each patch can naturally be represented as a geometry image in a $2^k \times 2^k$ section of a GPU texture.

Since all patches share the same resolution, packing them into one global atlas texture is trivial (see Fig. 1 for $k = 1$). The different surface attributes are stored in separate textures on the GPU: While for surface geometry like vertex positions and normal vectors floating point precision is used, lower precision is sufficient for diffuse and specular color.

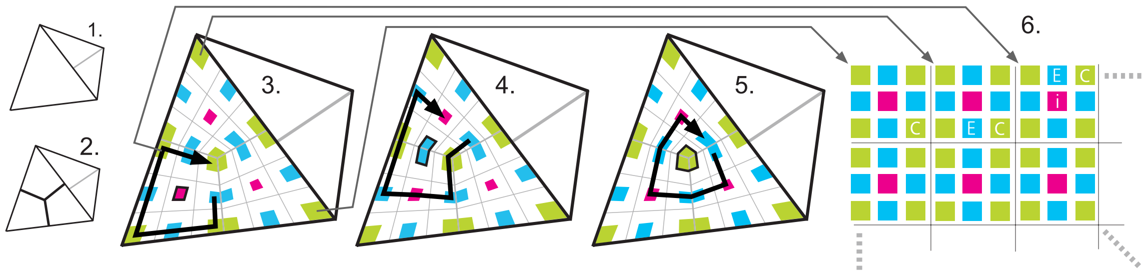

Figure 1: *Surface representation for a tetrahedron. Subdividing the base mesh (1) once yields a pure quad mesh (2). After a few additional subdivision steps the refined initial quads are mapped to quads in texture space (6). Enumerating the one-ring neighbors of interior (3), edge (4), and corner (5) vertices.*

Notice that vertices belonging to patch boundaries are multiplexed, since a copy is stored for each incident patch, which allows for easier rendering (Sec. 3) and neighborhood enumeration (Sec. 4). This representation is redundant: Interior vertices are stored only once, but vertices on an edge between two patches are doubled and vertices on patch corners are stored n times, where n is the vertex valence (Fig. 1).

However, the after a few subdivision steps the majority of vertices are interior. For instance, at subdivision level $k = 6$ a patch consist of $4^6 = 4096$ vertices, of which only $4\sqrt{4096} = 256 = 6.25\%$ are edge vertices and only 4 are corner vertices. Although this representation imposes a small memory and runtime overhead for edge and corner vertices, it allows the perfectly regular interior vertices to be processed by the GPU in the most efficient way (see below).

This representation is similar to the one in [SGP03], but differs in two aspects. First, a direct two dimensional matrix instead of linear array is used to store interior vertices, which is the preferred access pattern for GPUs. Moreover, no indirection is required to access corner and edge vertices, which provides faster traversal during rendering.

3. Surface Rendering

In order to render a patch of $2^k \times 2^k$ vertices, the two dimensional unit quad $[0,1]^2$ is tessellated at a resolution of $2^k \times 2^k$ and sent to the GPU. For each vertex, its position within the unit quad is used as texture coordinate, and a vertex texture fetch into the corresponding geometry image yields its 3D position, normal vector, and additional surface attributes. By this the triangles within the unit quad are mapped to their corresponding 3D positions.

Rendering the whole surface consisting of n patches then only requires to draw the tessellated unit quad n times, once for each geometry image chart. Notice that adaptive level-of-detail (LoD) rendering can be implemented simply by rendering a quad of coarser resolution $2^l \times 2^l$, $l < k$. This selects a subset of vertices, but still uses the high resolution normals and thus guarantees smooth shading.

4. Surface Manipulation

In order to modify a certain surface attribute (like vertex colors or positions) a function that transforms this attribute has to be called for each vertex. In general, these transformation functions require access to the one-ring neighbors of the current vertex, for instance to perform filtering of colors or geometric positions.

In our context, transforming a geometry image $\mathcal{S}$ to $\mathcal{S}'$ means that each texel of the geometry image (i.e., each vertex of the mesh) is transformed by a fragment shader, which reads its inputs from the texture $\mathcal{S}$ and writes its result to the texture $\mathcal{S}'$. However, while for the regular interior vertices this is similar to simple image filtering, the local surface neighborhoods are not regular for edge or corner vertices.

Storing and enumerating the one-ring neighborhoods therefore differs for the three kinds of vertices (Fig. 1). Since interior, edge, and corner vertices are treated differently, they also have to be sent to the GPU in different batches.

Interior vertices are stored only once and have a regular neighborhood. Their 4 neighbors can be accessed easily by offsetting their own texture coordinate by one texel in each direction. Since the interior vertices of a $2^k \times 2^k$ patch are regularly arranged as a quad in texture space, they can be sent to a fragment shader by rasterizing an image space quad of size $(2^k-2) \times (2^k-2)$, each pixel of which then corresponds to a texel in the geometry image.

Edge vertices also have 4 neighbors, but those are split over the two patches sharing that edge. In order to update the vertices belonging to a certain patch boundary, a horizontal or vertical line of length $2^k - 2$ is rasterized. During this update the texture space location of the opposite patch is passed to the GPU as a constant shader variable, which then allows to access the other half of the neighborhood.

Corner vertices are stored for every patch sharing this corner, the number of which is the valence v_i of the corner vertex i. To enumerate its neighbors a small look-up texture is used, which stores the neighborhood information of corner i in its ith row. Each row holds the valence v_i and the neighbors' texture coordinates $t_1, \ldots, t_{v_i}$. A fragment program updating the corner i first fetches v_i and then uses v_i dependent texture reads to enumerate the neighbors. Since the relative number of corner vertices is small, this additional overhead is negligible. In order to update all corner vertices a set of points (equipped with proper texture coordinates) is rendered.

This representation successfully puts the memory and computation overhead on the small percentage of edge and corner vertices, whereas the large regular parts can be processed by the GPU at maximum efficiency.

5. Surface Painting

A painting transformation requires several sub-tasks: finding the vertex under the mouse pointer, computing the brush's influence weight for all vertices, applying a (weighted) transformation, and updating the mesh. Each of these steps has to be implemented as efficiently as possible, i.e., on the GPU, which is described in the following.

1. Finding the location of the brush, i.e., the vertex under the mouse cursor, is difficult since the mesh is continuously deforming. We therefore render the mesh into a second render buffer, but instead of lighting each vertex, we encode its texture coordinates in color channels. The pixel color at the mouse position then identifies the closest vertex, i.e., the center of the brush.
2. The influence region of the brush is defined by computing a weighting factor for each mesh vertex from its distance to the brush center, using a linear or Gaussian transfer function. Textures can be used to further modulate this weighting. Using the general framework described in Sec. 4 these computations are performed by a fragment shader.
3. Another fragment shader detects patches that lie completely outside the influence region. Those patches are discarded from the following transformation, which effectively avoids unnecessary computations, in particular for small brush sizes (see Table 1).
4. The selected transformation is applied to each vertex, weighted by the brush's influence. The set of transformations we implemented contains color painting, smoothing of colors or positions, sculpting, extrusion, or multiresolution deformation (Sec. 6).
5. The normal vectors at each vertex are re-computed as the normals of the limit surface of the Catmull-Clark subdivision process.
6. Optional: For multiresolution deformations the detail reconstruction is computed (Sec. 6).

Notice that all of these steps are implemented on the GPU, which guarantees high performance during interactive mesh painting.

6. Multiresolution Deformations

The painting metaphor allows to intuitively deform the surface geometry, for instance by extrusion, sculpting, or dragging transformations. However, whenever the surface is deformed on a coarser subdivision level, one has to make sure that all its fine-scale geometric details on the higher subdivision levels are preserved and transformed in an intuitive manner.

This functionality is provided by so-called *multiresolution* or *multi-scale* deformations. Since we are dealing with subdivision surfaces, we follow the basic ideas of [ZSS97]. The multiresolution framework has to provide operators for subsampling, subdivision, and detail encoding.

Subsampling maps the surface to a lower level in the subdivision hierarchy. Subsampling from level k to level l, $l < k$, denoted by $\downarrow_l^k(\mathcal{S})$, is implemented by recursive Gaussian filtering. In $k-l$ rendering passes the patch charts of size $2^k \times 2^k$ are successively reduced to charts of size $2^l \times 2^l$. Each pass renders a patch of size $2m \times 2m$ to a patch of size $m \times m$ and hence corresponds to a standard reduction operator [BGH*04].

Subdivison from level l to level k, $k > l$, denoted by $\uparrow_l^k(\mathcal{S})$, implements the Catmull-Clark subdivision masks in a fragment program. This time, $k-l$ passes are used to expand patch charts of size $2^l \times 2^l$ to charts of size $2^k \times 2^k$.

Detail encoding. For a multiresolution deformation at subdivision level $l < k$, a *base* surface is generated in a preprocess by subsampling the highest level: $\mathcal{S}_b = \downarrow_l^k(\mathcal{S})$. The difference between $\mathcal{S}$ and the smooth upsampled base surface $\uparrow_l^k(\mathcal{S}_b)$, i.e., the high frequencies, are encoded as displacement vectors in local coordinate frames consisting of two orthogonal tangent vectors and the surface normal, similar to [ZSS97].

Whenever the user deforms the base surface $\mathcal{S}_b$ to $\mathcal{S}_b'$, the high frequency details are added back onto the subdivided deformed base surface $\uparrow_l^k(\mathcal{S}_b')$, which finally yields the deformed high resolution surface as $\mathcal{S}'$. Since all computations are performed entirely on the GPU, multiresolution deformations of complex models can be done at interactive rates.

7. Results

Table 1 gives some performance statistics of our GPU-based painting framework. At a medium brush size it is possible to manipulate and render even a complex surface of about 2.3M vertices (Fig. 2) at 8.3 fps on a NVIDIA GeForce 7800. The times include all the painting steps described in Sec. 5 and high quality rendering with a subsurface lighting effect and tone mapping. Subsurface lighting is approximated by smoothing illumination in texture space. More examples and demonstrations of different painting tools can be found in the accompanying video.

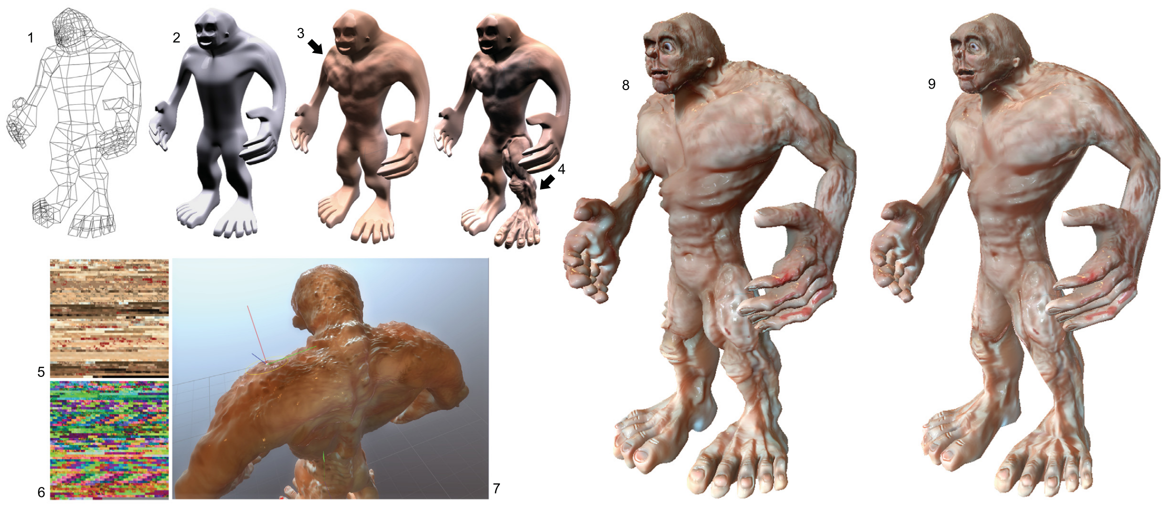

Figure 2: *A mesh with 2000 facets (1) subdivided to level 7 (2). Organic deformation (3, 4) is added by painting. The resulting geometry image for this mesh (diffuse in 5 and normals in 6). Color and geometry channels can be painted together with immediate high quality feedback (7). The final result (8). Lower tessellations can be used for LoD rendering (9).*

Faces	1%	10%	50%	LoD	Full
100k	30 ms	30 ms	30 ms	16 ms	16 ms
600k	60 ms	80 ms	100 ms	50 ms	50 ms
2.3M	120 ms	220 ms	390 ms	75 ms	230 ms

Table 1: *Overall performance for different brush sizes (1%, 10% and 50% of surface area), at different mesh resolutions, including LoD rendering at* 800×600 *pixels. Rendering alone: with LoD and without (Full).*

The maximum surface complexity is limited by the available GPU memory. For 256 MB GPU memory the 2.3M character is the highest resolution possible. Another drawback of our system is under-sampling under extreme deformations. This could be avoided by surface remeshing [LF03], but this does not seem to be suitable for a GPU implementation.

8. Conclusion

In this paper we proposed a surface representation based on piecewise geometry images, which allows for a fully GPU-based implementation of a mesh painting framework. Our technique handles arbitrary input meshes, achieves high update rates even for complex meshes, and generally scales with the rapidly increasing GPU (instead of CPU) performance.

References

[BGH*04] BUCK I., GOVINDARAJU N., HARRIS M., KRÜGER J., LEFOHN A. E., LUEBKE D., PURCELL T. J., WOOLLEY C.: GPGPU: General-purpose computation on graphics hardware. In *ACM SIGGRAPH course notes* (2004).

[CH04] CARR N. A., HART J. C.: Painting detail. In *Proc. of ACM SIGGRAPH* (2004), pp. 845–852.

[GGH02] GU X., GORTLER S. J., HOPPE H.: Geometry images. In *Proc. of ACM SIGGRAPH* (2002), pp. 355–361.

[HH90] HANRAHAN P., HAEBERLI P.: Direct WYSIWYG painting and texturing on 3D shapes. In *Proc. of ACM SIGGRAPH* (1990), pp. 215–223.

[LF03] LAWRENCE J., FUNKHOUSER T. A.: A painting interface for interactive surface deformations. In *Proc. of Pacific Graphics* (2003).

[LHSW03] LOSASSO F., HOPPE H., SCHAEFER S., WARREN J. D.: Smooth geometry images. In *Proc. of Symp. on Geometry Processing* (2003), pp. 138–145.

[LKS*06] LEFOHN A., KNISS J. M., STRZODKA R., SENGUPTA S., OWENS J. D.: Glift: Generic, Efficient, Random-Access GPU Data Structures. *ACM Trans. on Graphics 25*, 1 (2006).

[SGP03] SHIUE L., GOEL V., PETERS J.: Mesh mutation in programmable graphics hardware. In *Proc. of Graphics Hardware* (2003), pp. 15–24.

[SWG*03] SANDER P. V., WOOD Z. J., GORTLER S. J., SNYDER J., HOPPE H.: Multi-chart geometry images. In *Proc. of Symp. on Geometry Processing* (2003), pp. 146–155.

[ZPKG02] ZWICKER M., PAULY M., KNOLL O., GROSS M.: Pointshop 3D: An interactive system for point-based surface editing. In *Proc. of ACM SIGGRAPH* (2002), pp. 322–329.

[ZSS97] ZORIN D., SCHRÖDER P., SWELDENS W.: Interactive multiresolution mesh editing. In *Proc. of ACM SIGGRAPH 97* (1997), pp. 259–268.