Volume 0 (1981), Number 0 pp. 1–10 COMPUTER GRAPHICS *forum*

Local Painting and Deformation of Meshes on the GPU

H. Schäfer[†1] and B. Keinert[1] and M. Nießner[2] and M. Stamminger[1]

[1]University of Erlangen-Nuremberg
[2]Stanford University

Figure 1: *Interactive authoring using our dynamic GPU memory management approach. Left and right: simultaneous sculpting and painting on top of Catmull-Clark subdivision surfaces. Center: local surface deformations in a larger 3D scene environment. The total time to apply surface edits is well below a millisecond in all test examples.*

Abstract

We present a novel method to adaptively apply modifications to scene data stored in GPU memory. Such modifications may include interactive painting and sculpting operations in an authoring tool, or deformations resulting from collisions between scene objects detected by a physics engine. We only allocate GPU memory for the faces affected by these modifications to store fine-scale color or displacement values. This requires dynamic GPU memory management in order to assign and adaptively apply edits to individual faces at runtime. We present such a memory management technique based on a scan-operation that is efficiently parallelizable. Since our approach runs entirely on the GPU, we avoid costly CPU-GPU memory transfer and eliminate typical bandwidth limitations. This minimizes runtime overhead to under a millisecond and makes our method ideally suited to many real-time applications such as video games and interactive authoring tools. In addition, our algorithm significantly reduces storage requirements and allows for much higher-resolution content compared to traditional global texturing approaches. Our technique can be applied to various mesh representations, including Catmull-Clark subdivision surfaces, as well as standard triangle and quad meshes. In this paper, we demonstrate several scenarios for these mesh types where our algorithm enables adaptive mesh refinement, local surface deformations, and interactive on-mesh painting and sculpting.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Curve, surface, solid, and object representations I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing and texture

Keywords: real-time rendering, object modeling, GPU memory management, hardware tessellation

1. Introduction

Real-time rendering applications require storing scene data, such as geometry and material information, on the GPU. Due to bandwidth limitations and latency it is infeasible to modify complex objects on the CPU and transfer these to the GPU every frame. Instead, scene objects are typically animated by updating a few matrices that are then applied at render time by the GPU. However, it should also be possible to apply fine-scale editing operations to a scene, e.g.,

† henry.schaefer@cs.fau.de

when a user paints or sculpts objects, or through interactions between scene objects such as scratches, traces, or impacts. Performing such local changes on scene objects can be achieved by changing and updating color or displacement textures. Unfortunately, these texture updates are either global per object or restricted to a very limited number of decal textures that may be applied locally.

In this paper, we propose an alternative approach that allows performing fine-scale local changes on the appearance or geometry of virtual 3D environments. In contrast to previous approaches, we have no limitation as to where and when the modifications occur, nor as to how many modifications are applied, so long as they fit into the available GPU memory.

We consider color and surface displacements to be surface detail, which is obtained by the interactions between or with 3D objects. For instance, in authoring tools artists paint and sculpt surfaces locally by applying different brushes. Another example is video games, where mesh surfaces are modified by collisions or other interactions (e.g., bullets, footprints, scratches, etc.). The key contribution of our approach is to apply these changes locally and allocate storage only where needed. This reduces overall memory consumption, allows for higher-resolution content, and increases performance due to reduced memory I/O. In order to minimize runtime overhead, we specifically designed our approach to run entirely on the GPU, thus avoiding costly CPU-GPU memory transfer.

For efficient surface data storage, we employ the tile-based GPU texturing formats of Burley et al. [BL08] and Schäfer et al. [SPM*12], where each tile corresponds to a particular quad or triangle mesh face. As a mesh surface is modified, we allocate texture tiles in a GPU memory heap, similar to decals but within the same memory buffer. While representing and storing color data is straightforward, we use displacement offsets for local surface deformations. More specifically, we apply analytic displacements [NL13] that are rendered on top of Catmull-Clark subdivision surfaces using the GPU hardware tessellator. Furthermore, we handle triangular meshes by storing data as attributes of vertices generated by GPU tessellation patterns [SPM*12]. Note that both storage schemes support direct evaluation in a pixel shader and thus are applicable without hardware tessellation.

As use cases for our dynamically-managed GPU data structures, we implemented direct on-mesh painting and sculpting for vector- and scalar-valued surface displacements. The adaptive nature of our approach enables the local application of these modifications in large virtual environments without affecting authoring or render performance. In addition, we demonstrate an efficient employment of decals, where surfaces are modified based on object-object interaction in game-like scenarios.

2. Related Work

2.1. Mesh Data and Texturing Formats

Mesh data such as color and displacement offsets are typically stored in uv-atlas textures. Parameterizing meshes and providing consistent transitions at seams along uv-chart boundaries is a well-known issue in computer graphics. [SWG*03, GP09, SNK*14]. Hanrahan and Haberli [HH90] avoid these parameterization problems by employing vertex coloring, which restricts the amount of detail to the vertex density. Others employ spatial data structures, e.g., octree textures for storing and applying colors at the expense of additional run-time overhead introduced by traversing the tree data structure [LHN05].

Purnomo et al. [PCK04] avoid parameterization and seam problems by introducing per-face texturing based on an implicit surface-to-texture mapping. This allows them to find quadrilateral regions and align them with the 2D texture grid, thus avoiding inconsistent data access.

A conceptually more simple approach is PTex [BL08], where each face is mapped onto a single texture tile and adjacency pointers are used to perform data access along tile boundaries. PTex is particularly useful in the context of interactive authoring where data is dynamically generated on the CPU. Further development led to the adoption of tile-based texture formats for GPU applications, for instance in the context of displacement mapping [NL13, NSSL13] where texture seams result in intolerable surface cracks. Another way to avoid seam artifacts is the enforcement of boundary constraints [RNLL11] that need to be applied after each texture update.

Seamless texturing of triangular meshes is not supported by the previously mentioned tile-based texturing formats. Therefore, Yuksel et al. [YKH10] introduce Mesh Colors, where data is stored according to subdivision patterns for faces and edges through indirect memory access. Schäfer et al. [SPM*12] extend this idea for high-frequency surface detail and specifically target the GPU hardware tessellation unit. This approach involves assigning attribute data to vertices generated by the tessellator and supports multi-resolution content storage. We adopt this data structure for triangular meshes since it implicitly handles boundary overlap. In contrast to the original approach, we now dynamically manage vertex attribute data based on local deformation events [SKS13]. Thus, we are able to significantly reduce memory consumption by applying surface edits locally.

2.2. Local Surface Edits

When painting surfaces, direct feedback is crucial in order to guide artists during the interactive content creation process [HH90]. Recent research focuses on directly applying edits on the GPU [BKW10, LHN05] with an emphasis on multi-resolution editing [RBM06]. Multi-resolution ap-

proaches are also established in the context of surface modeling, and mostly based on subdivision surfaces [ZSS97], [KCVS98]. However, these approaches cannot handle fine-scale surface deformations since mesh levels are too coarse. Instead, triangles need to be added manually as presented by Paquette et al. [PPD01], where local subdivision is performed to account for small deformations. Since they only require inserting a few triangles per frame, their method runs at interactive rates on the CPU; however, it is not designed for parallel processing on the GPU.

2.3. Displacement Mapping and Hardware Tessellation

Displacement maps describe high-frequency surface deformations by storing either vector- or scalar-valued surface offsets in textures [Bli78, Coo84]. Displacements have been heavily used in the context of multi-resolution modeling (e.g., Guskov et al. [GVSS00] and Lee et al. [LMH00]). An overview of classical displacement mapping is provided by Szirmay-Kalos et al. [SKU08]. With the introduction of GPU hardware tessellation, displacement mapping has gained new momentum, providing dynamic tessellation of patch primitives and overcoming previous limitations to multi-resolution editing [TBB10, NCnP*09]. Therefore, smooth parametric surfaces (e.g., subdivision surfaces) are used as an underlying surface representation [LSNCn09, NLMD12, MNP08, VPBM01, BA08, SNK*14].

As previously mentioned, the challenge is to store associated displacement data efficiently and to avoid texture seam artifacts (cf. Section 2.1). While previous displacement storage approaches (e.g., Schäfer et al. [SPM*12], Nießner, Loop [NL13], Schäfer et al. [SKN*14]) allow for data consistency across boundaries, they suffer from their static memory layout. Thus, unnecessarily large textures are required in order to account for all potential surface deformations. We eliminate this limitation by dynamically allocating GPU memory only in regions when and where needed.

2.4. GPU Memory Management

Dynamic memory management is a well-researched topic in operating system engineering [JL96]. Steinberg et al. [SKKS12] introduce scattered parallel memory allocation for dynamic data. The key idea is to allocate a fixed amount of memory blocks and keep track of each block's fill rate. While this can be implemented on the GPU, it leads to internal fragmentation since data is scattered across multiple memory blocks. Instead, we avoid internal fragmentation by storing data to fit block sizes. This enables parallel allocation and deallocation, and thus greatly improves management and lookup performance.

3. Algorithm Overview

The input for our algorithm is a scene consisting of triangle or quad meshes, which we call *base meshes*. We apply painting and sculpting operations progressively to base meshes using our algorithm, as outlined in Figure 2. Note that the entire pipeline is executed on the GPU without costly scene data transfer between the CPU and GPU.

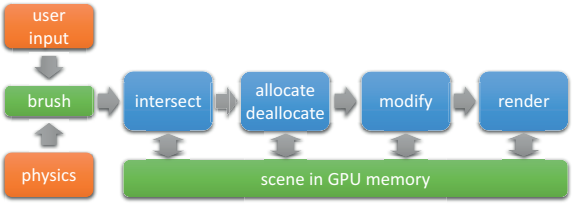

Figure 2: *Algorithm overview.*

Modifications are described using *brush objects*. The notion of a brush is inspired by authoring tools where user-controlled brushes are used to paint and deform scene objects. We define a brush as an oriented bounding box (OBB), bounding the region of influence, and a brush function describing the edits to be applied in the brush's space. The brush function can be defined as an analytic function, a texture defining colors for painting brushes, or a displacement map for sculpting brushes. We position user-controlled brushes on the surfaces of objects at the mouse cursor with the brush orientation defined by the underlying tangent frame. In order to compute the attribute values for modified base faces, we determine the affected data positions on face tiles, map these to brush space, and evaluate the brush function.

We also use brushes to deform objects at collisions, which are detected by a physics simulation. Therefore, we simply attach brushes to object parts such as the feet of animated characters. If the physics simulation detects a collision between a brush and another object, we apply deformations according to the brush function. For instance, if a foot touches soft ground, the underlying surface is automatically deformed to mimic the geometric shape of the footprint.

Once a brush has been defined, we determine all base mesh faces that are affected by the current brush in the *intersect* stage. To this end, we intersect all base faces with the OBB of the brush. Note that intersections must be computed with the displaced surface geometry, rather than the base mesh. To avoid costly intersection tests with the fully tessellated geometry, we determine a conservative bound of the base mesh's OBB considering the maximum displacement extent. We use the brush OBB and the extended base face OBB as a trivial rejection test; if there is no overlap, then all sub-faces do not intersect. Otherwise, all tessellated sub-faces need to be considered.

For all (newly) affected base mesh faces, memory is allocated to store new color or deformation information. Memory may also be deallocated in case modifications are removed or the available memory is exhausted. The *allocation* and *deallocation* step requires global memory handling, which we will detail in Section 4. In particular, this step heavily depends on the underlying mesh data structure. In this paper, we explain and present results for two different mesh data structures. First, we look at quad meshes, subdivided using Catmull-Clark subdivision and a tile-based texture format similar to PTex [BL08]. Second, we consider triangle meshes using the multi-resolution mesh attributes approach by Schäfer et al. [SPM*12].

Once the memory has been assigned, we evaluate the brush and *modify* all affected faces (i.e., both newly and previously allocated faces). Finally, the scene is *rendered*. Therefore, sculpted displacements and painted colors are directly evaluated in corresponding domain and pixel shaders [SPM*12, NL13] using our dynamic data structures. Surface deformations are applied using GPU hardware tessellation, enabling adaptive level-of-detail rendering. Additionally, we perform feature-adaptive Catmull-Clark subdivision [NLMD12, Nie13] on the quad meshes.

4. Dynamic GPU Memory Management

Dynamically adding and removing attributes requires managing memory chunks which store attribute data. In this section, we describe the general concept of our memory management method, which will be refined in the following sections for varying applications and data structures.

We assume scene environments consisting of meshes with triangle and quad faces. For some of these faces, we want to assign memory, possibly at a high resolution, in order to store additional attribute values for face modifications. In our examples, these attributes are displacement values or colors, generated by user input or by a physical simulation. For rendering Catmull-Clark subdivision surfaces and displacement mapping, faces are subdivided at render time using hardware tessellation, and the attributes of the modified faces are applied in the tessellation shaders.

While the application is running, the set of modified faces and attribute values changes dynamically. Some faces obtain new attributes (e.g., when a brush is applied for the first time), some faces have attributes modified (e.g., when a brush is applied to a previously edited face), and some faces have attributes removed (e.g., if an eraser brush is applied).

Since we consider scenes with large face counts and high attribute block resolutions, we typically cannot pre-allocate an attribute block for all faces in a scene. Instead, we pre-allocate a fixed amount of attribute blocks, the *attribute heap*, which we use to dynamically assign blocks to faces. Each face stores a pointer, which references a heap block if it has assigned attributes.

Next, we introduce a parallel GPU memory management mechanism in order to allocate new attribute blocks and to deallocate blocks if they are no longer used. To keep track of unused blocks in the attribute buffer, we maintain a list of free blocks, where n_{free} is the number of available elements (see Fig. 3). For efficient allocation, deallocation, and modification, we apply the operations depicted in Fig. 4. First, a simple *classification kernel*, running on all faces, determines whether an attribute block should be allocated for a face, whether the attributes should be modified, or whether the attribute block for the face can be deallocated.

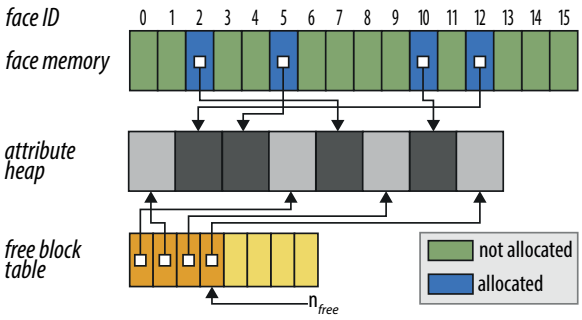

Figure 3: *Memory management: the attribute heap contains memory blocks of equal size that maintain high-resolution face attributes. Modified faces (dark blue) possess a pointer to one of these blocks. References to free blocks are stored in the free block table.*

Second, a multi-compaction operation is applied, which outputs three lists: faces that require allocation, faces that require modification, and faces that can be deallocated. Such a multi-compaction can be performed very efficiently using a parallel scan operator [Ble90, SHGO11, HSO07].

Once these lists are generated, we efficiently process the allocations, modifications, and deallocations using three GPU kernels, one for each list:

- For the allocation, each thread i simply assigns the attribute block at position $n_{free} - i$ from the free block table to face `allocateSet[i]`. After this, n_{free} is globally decreased by the number of allocations.
- The modification kernel applies brush edits to the attributes of face `modifySet[i]`.
- The deallocation kernel writes the attribute block pointer of face `deallocateSet[i]` back to the free block table at position $n_{free} + i + 1$, and sets the attribute block pointer to `null`. Then, n_{free} is globally increased by the number of deallocations.

With this approach, all synchronization has been moved to the compaction operation, which is insignificant with regard to computation time. All steps are executed in parallel and no further synchronization is required.

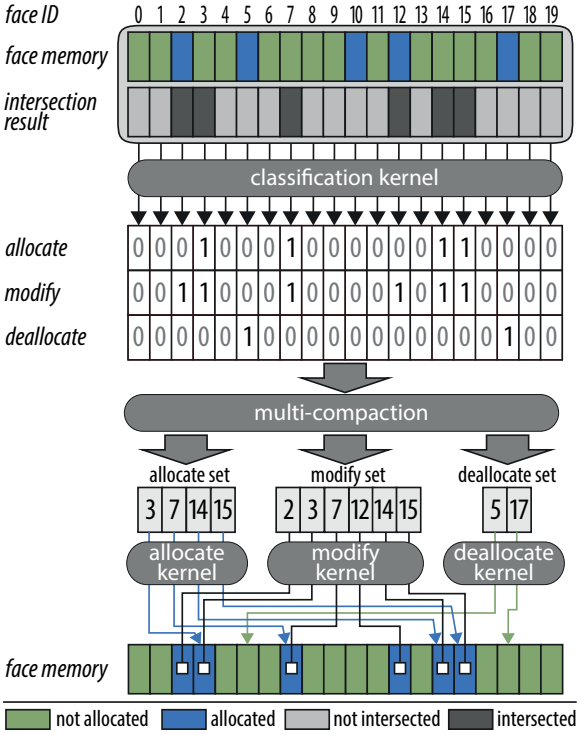

Figure 4: *Memory management: first, for each face, we compute whether attributes need to be allocated, modified, or deallocated (classification kernel). Second, resulting decision arrays are transformed into index-arrays containing all affected faces (multi-compaction). Finally, for each of these arrays, kernels are executed that perform allocation, modification, and deallocation without requiring further synchronization.*

In order to handle specific data structures, we need to further extend our scan-based memory management scheme. For example, to store tessellation attributes, we want to distinguish between attributes per vertex, per edge, and per face, and we typically want to store attribute blocks of different resolutions. We introduce these extensions in Section 5 and 6, along with painting and sculpting applications.

5. Dynamic Mesh Modifications on Quadrilateral Faces

In this section, we consider dynamic modifications of meshes composed of quadrilateral faces. We dynamically assign texture tiles to faces which store local detail. In our case, this is color and/or displacement data. Each face stores a memory pointer which indicates the allocation of texture tiles. The strength of our method is the incremental tile allocation through surface painting and deformation at runtime.

Each allocated tile corresponds to one specific face, similar to the PTex format [BL08]. We manage tiles for color, scalar, and vector displacement data independently. Our default for surface offsets is the analytic displacement format [NL13], since it implicitly conveys surface normal information. We also support traditional displacements; however, normal re-computation after surface edits causes additional computational overhead.

In contrast to the original PTex format, we additionally store a one-texel boundary overlap for each tile. While this causes some memory overhead due to redundant data storage, it is beneficial from a rendering standpoint since most filter operations can be performed without adjacency access. Further, storing a mip-map hierarchy per tile (see Figure 5) allows for efficient tri-linear interpolation. More elaborate data access such as anisotropic filtering is also feasible, but requires reads from multiple tiles.

In order to obtain data from neighboring tiles, we store the mesh connectivity in a static GPU buffer. Since each tile knows its corresponding face index, mesh connectivity and tile adjacency pointers are equivalent. While this involves one level of indirection, it abstracts from dynamic tile allocation.

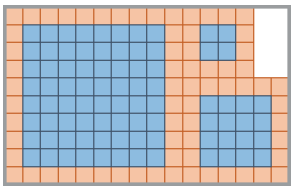

Figure 5: *Layout of a tile: each tile keeps the original square attribute grid (left) and down-filtered versions (right). To ease and accelerate rendering, each mip-map level additionally stores a one-ring boundary overlap (red) which contains copies of values of adjacent tiles.*

5.1. Allocation and Deallocation

For allocating and deallocating per-face texture tiles, we employ the memory management described in Section 4. Since color and displacements have different memory footprints and resolutions, we use two different heaps with their own free memory tables. Note that displacements are either vectors or scalars depending on modeling flexibility and storage constraints.

5.2. Painting and Deformation

Next, we process all faces that are flagged as *modify* based on our brush input. Note that these include all faces designated for *allocation*. Based on the brush type, we apply respective edits to the local tile. Since obtained face IDs map to allocated tiles, we can directly write edits to GPU memory without further synchronization. For paint brushes, we use single color input with different falloff variations and textures (see Section 3). For each affected tile texel, we map

its position to brush space and evaluate the brush function. Sculpting is performed similarly, while taking advantage of analytic displacements [NL13] which do not require surface normal re-computation.

5.3. Overlap Update

After tile data has been written, we need to update the boundary overlap of modified tiles. Therefore, we employ the pre-computed mesh connectivity data for accessing adjacency information. First, we run an edge kernel that copies overlap along edges (we use one thread per texel). Next, we compute consistent data for texels at extraordinary points; i.e., we equalize corner texel values by averaging the corresponding corner values. Finally, we update the corner overlap of all patches. Since we only update the overlap of tiles flagged *modify*, runtime overhead is marginal.

5.4. Subdivision Surface Rendering

For visualization in our demos, we use the *OpenSubdiv* framework, which incorporates subdivision surface rendering using GPU hardware tessellation [NLMD12] and supports PTex input data [BL08]. During rendering, tile data is applied to faces with attached textures. For displacements, surface offsets (either scalar- or vector-valued) are evaluated as analytic displacements [NL13], avoiding the requirement for normal maps. Note that tiles are stored in texture arrays and maintain a one-texel boundary overlap. This enables hardware filtering for color data except for anisotropic filtering where we manually access adjacent tiles.

6. Dynamic Modification of Triangle Meshes

In this section, we describe handling local modifications on triangle meshes. We employ the data structure proposed by Schäfer et al. [SPM*12], which is based on an implicit parameterization for triangle meshes. Attributes are stored at pre-defined domain locations corresponding to the tessellation patterns of the hardware tessellator. Since data access is indexed and shared along edges, data consistency is implicitly enforced.

Instead of a single tile per face as in the previous section, we now require storage for base mesh vertices as well as vertices on tessellated edges and faces. The corresponding *attribute heap* and memory layout (see Figure 6) contain distinct blocks for vertices, tessellated edges and interior vertices of tessellated triangles. We dynamically manage data only for edges and faces, but not for vertices of the base triangle, since the storage requirement is not significant enough to justify the computational overhead.

For now, we assume that if a triangle is subdivided, only a single user-defined tessellation factor T, e.g., $T = 16$, is used. The number of new vertices after tessellation is defined by $\#V_{edge} = T - 1$ for edges and $\#V_{face} = \lfloor \frac{3T}{2}(\frac{T}{2} - 1) + 1 \rfloor$

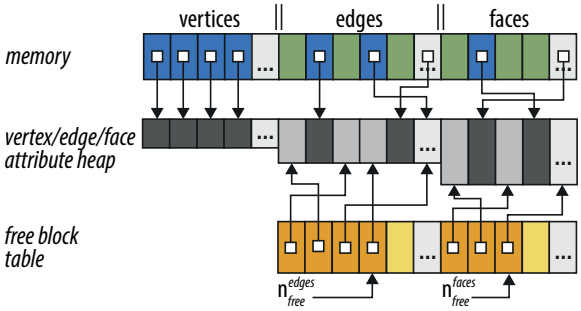

Figure 6: *Data layout for dynamic memory management, adapted for the storage of tessellation attributes. The attribute heap contains memory blocks for vertices, edges, and faces, all of different size. Attributes for vertices are statically assigned, whereas we dynamically allocate memory for edges and faces. Two free memory tables keep track of unused attribute blocks.*

within a face, respectively. The attribute heap contains three types of attribute blocks: first, one attribute for each base mesh vertex, then a number of blocks of size V_{edge} for edges, and lastly, a number of blocks of size V_{face} for faces. Expanding upon Euler's formula for closed meshes, we allocate twice as many edge blocks as face blocks. Figure 6 shows the described memory layout.

6.1. Allocation and Deallocation

To handle allocation and deallocation, we apply our memory management scheme described in Section 4. However, we now need to allocate and deallocate data for the affected edges and faces. Thus, we perform the brush intersection computation for all edges and faces, and have the multi-scan output separate processing flag arrays (*allocSet*, *modifySet*, and *deallocSet*) for both entities. We then allocate and deallocate memory blocks for edges and faces separately.

6.2. Modification

Next, we must set the attribute blocks of the edges and faces from the modify list according to the current brush. We execute GPU kernels for all these edge and face vertices to compute the new attribute values. In both kernel types, we iterate over all vertices that result from tessellation, compute their attribute value from the current brush, and store the value in the attribute block. The iteration over all vertices along an edge is simple. For vertices in the interior of a base face, we have to invert the indexing scheme. That is, we reconstruct the barycentric coordinate within the face of the base mesh directly from the index. An efficient method for computing this inversion can be found in [SPM*12].

6.3. Normal Computation

Shading displaced objects typically requires a normal map. When sculpting a mesh, displacements are changed, and normals must be re-computed based on the new displacement data. In the following, we show an efficient method to compute displaced surface normals on-the-fly.

A smooth normal at a vertex is typically computed by averaging the geometric normals of the neighboring faces [Max99], which involves traversing the one-ring of a vertex. There are eight different one-ring configurations for vertices that are generated by hardware tessellation from triangular base patches (see Fig. 7). In order to compute normals for edges on base patch boundaries, we fetch displacement data from both adjacent faces. On patch corner points that are affected by deformations, adjacency information is stored to perform the one-ring traversal. We compute area-weighted normals by summing up unnormalized face normals. As we only re-compute normals for modified faces, we store resulting normals in a separate buffer whose elements correspond to displacement values. Normal re-computation is implemented in a compute shader and handles all affected vertices. Note that computed normals converge to the geometric per-face normals of the base patch if displacement values are zero. To prevent flat-shading artifacts, we interpolate between smooth base patch normals and newly computed normals based on the displacement length.

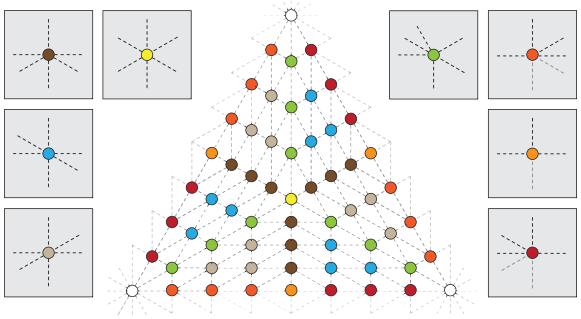

Figure 7: *Tessellation pattern showing the eight topological cases for a one-ring traversal on a tessellated triangle face.*

6.4. Triangle Mesh Rendering

When rendering scenes, we process displacements using hardware tessellation and access tile colors in a pixel shader. In contrast to Section 5, the base mesh is not a subdivision surface. Hence, the tessellator linearly interpolates triangular patches and applies assigned displacement attributes. In the hull shader, we determine whether a patch has allocated displacement data. If so, we specify tessellation factors for edges and face interiors; otherwise, the base triangle is rendered without further tessellation. We also obtain the start index of the memory block of the current triangle in the hull shader. Finally, the local memory index within a block is determined in the domain shader based on the barycentric coordinates (cf. [SPM*12]). This allows us to access and process vertex attribute data.

7. Extensions

7.1. Multi-Resolution Handling

When sculpting an object, typically a rough shape is sculpted first, which is then progressively refined. For this purpose, most sculpting applications support deformations at different detail levels. We enable this kind of multi-resolution editing by allowing for different memory block sizes. More specifically, we support two additional tile resolutions, $T/2$ and $T/4$, where T is the resolution of the original face (quad or triangle). We also store free memory tables to maintain the additional memory blocks, which are shown in Fig. 8 for triangular tiles.

When an editing operation is applied at a selected level, three states are possible: 1) The element was not previously edited. In this case, we allocate memory and edit the element. 2) The object has been previously edited at the selected level and memory has already been allocated. In this case, we can directly apply edits. 3) The element has been edited previously, but at a different level. In this case, we first allocate memory at the specified level and then copy attribute values to all other levels in order to enforce consistency between levels.

7.2. Removal of Edits

Our memory management system also supports dynamic deallocation of assigned data. For instance, an eraser brush can remove previous edits. Modifications may also decay over time. In these cases, we smoothly fade out modifications using linear blending, and deallocate memory blocks at some point. To ensure temporal causality, we store timestamps of edit events at corresponding data locations (texel or vertex attributes). If modifications are applied to faces that already contain edits, the timestamps are updated accordingly. Thus, only the most recent edit and timestamp needs to be considered.

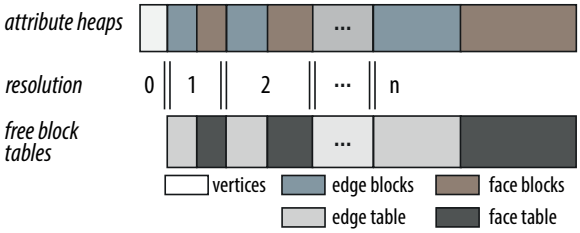

Figure 8: *Multi-resolution handling: for each resolution, we allocate different edge and face blocks (attribute heap), and maintain a separate free memory table.*

Figure 9: *From left to right: mesh without fine detail (no memory allocated); mesh after surface painting with a textured brush; visualization of memory allocation status (red = allocated tiles, green = free tiles).*

8. Results

We implemented our algorithm using DirectX 11 running under Windows 8. All our GPGPU kernels use Direct Compute, and we employ the Direct3D 11 graphics API for rendering. Accordingly, all our GPU code is written in HLSL and runs on an NVIDIA GTX Titan. Times are provided in milliseconds and account for all runtime overhead except UI rendering. Our default render resolution is 1920x1080. In the following, we provide results for tile-based editing of quadrilateral meshes as well as triangle meshes which use the tessellation pattern storage scheme. Additionally, we show measurements for brush intersection and GPU memory management.

8.1. Tile-based Editing

In our first example, we apply dynamic editing on the *Frog* and *Dragon* subdivision surface models (both quads only) with 1292 and 2782 base faces, respectively. We apply our dynamic memory management scheme on top of these models rendered using feature adaptive Catmull-Clark subdivision and show corresponding results in Fig. 9. Green regions account for faces without allocated color and displacement data; red regions correspond to allocated tiles. A dynamic memory management example for simultaneous painting and sculpting is shown in Fig. 1 left and right. Render times are 1.07 ms for the *Frog* and 0.89 ms for the *Dragon* model. Fig. 10 shows the results of an interactive painting and sculpting session on triangular meshes. An example of vector displacements is depicted in Fig. 11. In the middle of Fig. 1, we show a large scene environment, on which we locally apply displacements. The render time for the larger *Streets of Asia* scene environment, which consists of 800K triangles, is about 6 ms.

Note that our memory management scheme does not involve any overhead to access tile data. Compared to uv-atlas textures, which require loading uv coordinates for face vertices (i.e., 24 byte for triangles, 32 byte for quads), our tile-based data structures require fewer global reads to access tile data (16 byte per triangle, 6 byte per quad patch). Texture atlas formats are also unsuitable for dynamic memory management since dynamically bin-packing texture charts to a texture atlas requires a costly global optimization step.

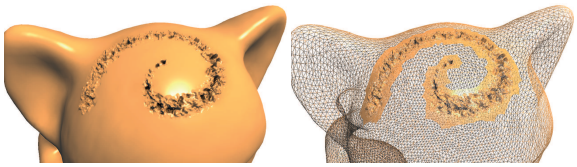

Figure 10: *Example of dynamic memory allocation: scalar displacement values are stored in regions affected by a brush as shown in the wireframe overlay.*

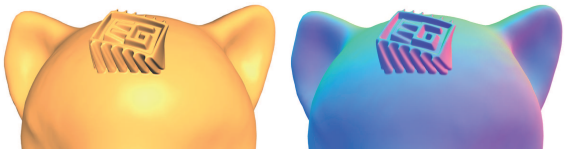

Figure 11: *Vector displacement edits using our adaptive GPU memory management. The image on the right illustrates displaced surface normals.*

8.2. Intersection and GPU Memory Management

When an edit operation is applied to a mesh, we first intersect the mesh with the brush OBB. Then, we compact the result buffer in order to reduce work on subsequent algorithm stages (cf. Section 4). Table 1 depicts the overall processing times for these two stages for the test models shown in Fig. 12. Additionally, we subdivide the dragon model one and two times in advance to to show the scalability of our method for high patch count subdivision surface models. As expected, performance scales linearly with respect to the input mesh size since intersections and compaction are performed at the face level. While intersection is more expensive for quadrilateral meshes by a constant factor

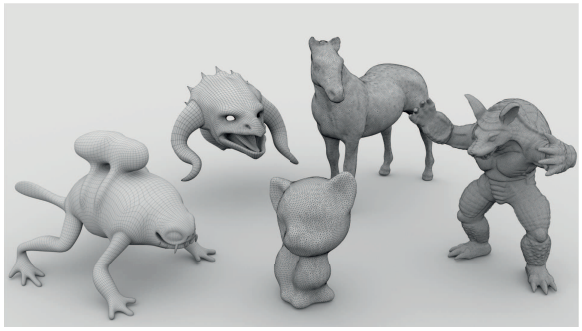

Figure 12: *We perform measurements on the shown set of test models ranging from 36k to 500k input elements (patches for subdivision models, edges and faces for triangle models). From left to right: frog and dragon subdivision surface models; kitten, horse and armadillo triangle models.*

(more reads for control points per patch are involved), scan behaves similarly since it scales with the number of input elements. Once deformation events are identified, we trigger GPU memory management and update the data of affected faces; i.e., we process edits caused by painting or deformation.

mesh	input elements	intersect (in ms)	scan (in ms)	manage (in ms)
Dragon*	36 k	0.25	0.025	0.006
Dragon SubD 1*	76 k	0.78	0.028	0.006
Dragon SubD 2*	208 k	2.31	0.049	0.006
Frog*	42 k	0.27	0.028	0.006
Kitten	65 k	0.017	0.038	0.006
Horse	298 k	0.058	0.122	0.006
Armadillo	507 k	0.068	0.203	0.006

Table 1: *Performance of brush intersection and multi-scan kernel on a set of test models shown in Fig. 12. For quadrilateral meshes (marked with *) input elements correspond to the base mesh face count; for triangular meshes, both faces and edges are considered input elements.*

Typically, less than 100 faces are affected by a brush at the same time in our test scenarios. Overall, the GPU memory management time was approximately 0.006 ms and the corresponding time for processing edits was in the range of 0.05 ms.

Finally, dynamic GPU memory management enables the on-the-fly allocation of high-resolution face textures where and when needed. In contrast to global parametrization and texturing approaches, only a subset of the scene faces occupy GPU memory. Thus, painting and sculpting with very high detail, as shown in Fig. 13, is easily feasible, even on hardware with a small memory budget.

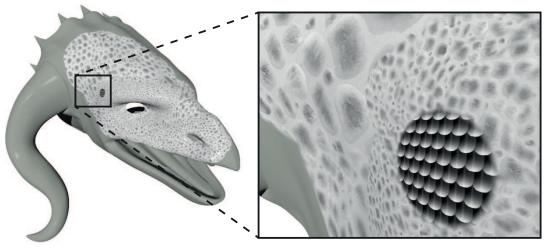

Figure 13: *Example of dynamic allocation and painting on the dragon model (left), allowing for local very high-resolution tile textures as shown in the close-up (right).*

9. Conclusion

We presented a system for performing local edits on large scenes, including both triangle meshes and quadrilateral Catmull-Clark subdivision surfaces. By using tailored, parallel memory management, local edits are applied within a few microseconds. This high performance makes our method applicable even under the highly strict time constraints of video games, and allows interactive sculpting and painting applications to edit very detailed meshes with immediate response.

One limitation is that base faces can only be tessellated up to a maximum tessellation factor. While this is irrelevant for most scenarios, the resolution may be insufficient for large base faces. Therefore, we typically subdivide large input faces until we consider their size to be small enough. Another limitation, particularly for triangle meshes, is that we are forced to use the hardware tessellation patterns. This occasionally causes problems for poorly shaped input triangles, where larger tessellation factors are necessary to avoid rendering artifacts. In such cases, we recommend scene preprocessing to improve the mesh quality; i.e., subdividing large input triangles and avoiding non-uniform mesh tessellations. However, we see significant potential for improvement in this area. In particular, creating better adaptive tessellation patterns that are able to handle arbitrarily shaped input triangles is a promising future research direction.

Acknowledgements

The authors would like to thank the Blender Foundation for the Dragon model and the animated character shown in the video. The Steets of Asia scene was modelled by Stonemason. The Kitten, Horse and Armadillo models are provided by AIM@SHAPE and the Stanford 3D scanning repository. This work was partly supported by the Research Training Group 1773 "Heterogeneous Image Systems", funded by the German Research Foundation (DFG). We would also like to thank Angela Dai for proofreading.

References

[BA08] BOUBEKEUR T., ALEXA M.: Phong Tessellation. *ACM Trans. Graph. 27*, 5 (2008), 141:1–141:5. 3

[BKW10] BÜRGER K., KRÜGER J., WESTERMANN R.: Sample-based Surface Coloring. *TVCG 16*, 5 (2010), 763–776. 2

[BL08] BURLEY B., LACEWELL D.: Ptex: Per-Face Texture Mapping for Production Rendering. *Proc. EGSR'08 27*, 4 (2008), 1155–1164. 2, 4, 5, 6

[Ble90] BLELLOCH G. E.: *Prefix Sums and Their Applications*. Tech. Rep. CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, 1990. 4

[Bli78] BLINN J. F.: Simulation of wrinkled surfaces. In *ACM SIGGRAPH Computer Graphics* (1978), vol. 12, ACM, pp. 286–292. 3

[Coo84] COOK R. L.: Shade Trees. *Computer Graphics (SIGGRAPH'84 Proceedings) 18*, 3 (1984), 223–231. 3

[GP09] GONZÁLEZ F., PATOW G.: Continuity Mapping for Multi-Chart Textures. *ACM Trans. Graph. 28* (2009), 109:1–109:8. 2

[GVSS00] GUSKOV I., VIDIMČE K., SWELDENS W., SCHRÖDER P.: Normal Meshes. In *SIGGRAPH Proceedings* (2000), pp. 95–102. 3

[HH90] HANRAHAN P., HAEBERLI P.: Direct WYSIWYG painting and texturing on 3D shapes. *ACM SIGGRAPH Computer Graphics 24*, 4 (1990), 215–223. 2

[HSO07] HARRIS M., SENGUPTA S., OWENS J. D.: Parallel prefix sum (scan) with CUDA. *GPU gems 3*, 39 (2007), 851–876. 4

[JL96] JONES R., LINS R. D.: Garbage collection: algorithms for automatic dynamic memory management. 3

[KCVS98] KOBBELT L., CAMPAGNA S., VORSATZ J., SEIDEL H.-P.: Interactive multi-resolution modeling on arbitrary meshes. In *SIGGRAPH Proceedings* (1998), SIGGRAPH '98, pp. 105–114. 3

[LHN05] LEFEBVRE S., HORNUS S., NEYRET F.: *GPU Gems 2*. Addison-Wesley, 2005, ch. Octree Textures on the GPU. 2

[LMH00] LEE A., MORETON H., HOPPE H.: Displaced subdivision surfaces. In *SIGGRAPH Proceedings* (2000), pp. 85–94. 3

[LSNCn09] LOOP C., SCHAEFER S., NI T., CASTAÑO I.: Approximating Subdivision Surfaces with Gregory Patches for Hardware Tessellation. *ACM Trans. Graph. 28* (2009), 151:1–151:9. 3

[Max99] MAX N.: Weights for computing vertex normals from facet normals. *Journal of Graphics Tools 4*, 2 (Mar. 1999), 1âĂŞ6. 7

[MNP08] MYLES A., NI T., PETERS J.: Fast parallel construction of smooth surfaces from meshes with tri/quad/pent facets. In *Computer Graphics Forum* (2008), vol. 27, Wiley Online Library, pp. 1365–1372. 3

[NCnP*09] NI T., CASTAÑO I., PETERS J., MITCHELL J., SCHNEIDER P., VERMA V.: Efficient Substitutes for Subdivision Surfaces. In *ACM SIGGRAPH 2009 Courses* (2009), pp. 13:1–13:107. 3

[Nie13] NIESSNER M.: *Rendering Subdivision Surfaces using Hardware Tessellation*. Dissertation. Dr. Hut, 2013. 4

[NL13] NIESSNER M., LOOP C.: Analytic Displacement Mapping using Hardware Tessellation. *ACM Transactions on Graphics (TOG) 32*, 3 (2013), 26. 2, 3, 4, 5, 6

[NLMD12] NIESSNER M., LOOP C., MEYER M., DEROSE T.: Feature-adaptive GPU rendering of Catmull-Clark subdivision surfaces. *ACM Trans. Graph. 31*, 1 (2012), 6. 3, 4, 6

[NSSL13] NIESSNER M., SIEGL C., SCHÄFER H., LOOP C.: Real-time Collision Detection for Dynamic Hardware Tessellated Objects. *Proc. EG'13* (2013). 2

[PCK04] PURNOMO B., COHEN J. D., KUMAR S.: Seamless Texture Atlases. In *Proc. SGP'04* (2004), ACM, pp. 65–74. 2

[PPD01] PAQUETTE E., POULIN P., DRETTAKIS G.: Surface aging by impacts. In *Proc. Graphics interface '01* (2001), GRIN'01, pp. 175–182. 3

[RBM06] RITSCHEL T., BOTSCH M., MÜLLER S.: Multiresolution GPU Mesh Painting. *Eurographics 2006, Wien* (2006). 2

[RNLL11] RAY N., NIVOLIERS V., LEFEBVRE S., LEVY B.: Invisible Seams. *Proc. EGSR'10 29*, 4 (2011), 1489–1496. 2

[SHGO11] SENGUPTA S., HARRIS M., GARLAND M., OWENS J. D.: Efficient Parallel Scan Algorithms for many-core GPUs. In *Scientific Computing with Multicore and Accelerators* (Jan. 2011), pp. 413–442. 4

[SKKS12] STEINBERGER M., KENZEL M., KAINZ B., SCHMALSTIEG D.: ScatterAlloc: Massively Parallel Dynamic Memory Allocation for the GPU. In *Proc. InPar'12)* (2012). 3

[SKN*14] SCHÄFER H., KEINERT B., NIESSNER M., BUCHENAU C., GUTHE M., STAMMINGER M.: Real-Time Deformation of Subdivision Surfaces from Object Collisions. In *Proceedings of the 6th High-Performance Graphics Conference* (2014), ACM, pp. –. 3

[SKS13] SCHÄFER H., KEINERT B., STAMMINGER M.: Real-time Local Displacement using Dynamic GPU Memory Management. In *Proceedings of the 5th High-Performance Graphics Conference* (2013), ACM, pp. 63–72. 2

[SKU08] SZIRMAY-KALOS L., UMENHOFFER T.: Displacement Mapping on the GPU - State of the Art. *Computer Graphics Forum 27*, 1 (2008). 3

[SNK*14] SCHÄFER H., NIESSNER M., KEINERT B., STAMMINGER M., LOOP C.: State of the Art Report on Real-time Rendering with Hardware Tessellation. In *Proceedings of EG'14* (2014), Eurographics Association. 2, 3

[SPM*12] SCHÄFER H., PRUS M., MEYER Q., SÜSSMUTH J., STAMMINGER M.: Multiresolution Attributes for tessellated Meshes. In *Proc. I3D'12* (2012), pp. 175–182. 2, 3, 4, 6, 7

[SWG*03] SANDER P. V., WOOD Z., GORTLER S. J., SNYDER J., HOPPE H.: Multi-Chart Geometry Images. In *Proc. SGP'03* (2003), pp. 146–155. 2

[TBB10] TATARCHUK N., BARCZAK J., BILODEAU B.: Programming for Real-Time Tessellation on GPU. *AMD whitepaper 5* (2010). 3

[VPBM01] VLACHOS A., PETERS J., BOYD C., MITCHELL J. L.: Curved PN Triangles. In *Proc. I3D'01* (2001), pp. 159–166. 3

[YKH10] YUKSEL C., KEYSER J., HOUSE D. H.: Mesh Colors. *ACM Trans. Graph. 29*, 2 (2010), 15:1–15:11. 2

[ZSS97] ZORIN D., SCHRÖDER P., SWELDENS W.: Interactive Multiresolution Mesh Editing. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (1997), ACM Press/Addison-Wesley Publishing Co., pp. 259–268. 3