High Performance Graphics (2014)
Jonathan Ragan-Kelley and Ingo Wald (Editors)

Real-Time Deformation of Subdivision Surfaces from Object Collisions

Henry Schäfer[1], Benjamin Keinert[1], Matthias Nießner[2], Christoph Buchenau[3], Michael Guthe[3], Marc Stamminger[1]

[1]University of Erlangen-Nuremberg [2]Stanford University [3]University of Bayreuth

Figure 1: *Example scenes showing our real-time surface deformation technique. The user-controlled car (left) and the animated character (right) cause fine-scale deformations on the terrain surfaces. While deformable surfaces are represented as displaced Catmull-Clark subdivision surfaces, we voxelize the actual surface geometry of rigid objects in order to determine their respective deformations.*

Abstract

We present a novel real-time approach for fine-scale surface deformations resulting from collisions. Deformations are represented by a high-resolution displacement function. When two objects collide, these offsets are updated directly on the GPU based on a dynamically generated binary voxelization of the overlap region. Consequently, we can handle collisions with arbitrary animated geometry. Our approach runs entirely on the GPU, avoiding costly CPU-GPU memory transfer and exploiting the GPU's computational power. Surfaces are rendered with the hardware tessellation unit, allowing for adaptively-rendered, high-frequency surface detail. Ultimately, our algorithm enables fine-scale surface deformations from geometry impact with very little computational overhead, running well below a millisecond even in complex scenes. As our results demonstrate, our approach is ideally suited to many real-time applications such as video games and authoring tools.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—

1. Introduction

In recent years, increasing attention has been devoted to dynamic scene environments. In real-time rendering, applying instant fine-detail surface deformations caused by object collisions is an open research problem (cf. Figure 1). The key issue is that physics simulation and collision detection typically run on the CPU, while surface geometry for rendering is stored on the GPU. This requires the CPU side to access mesh data from the GPU in order to compute surface deformations. In addition, physics updates involve uploading modified surface geometry back to the GPU. This results in significant runtime overhead and affects performance on current hardware architectures due to the bandwidth and latency limitations of the CPU-GPU memory bus. Hence, maintaining interactive frame rates is only feasible for moderately complex meshes.

Similar problems arise in character animation. Animating the entire mesh of a character on the CPU every frame would require costly memory transfers. Instead, the mesh is animated using skinning. While the detailed mesh of a character is kept in GPU memory, animations are provided by a small set of bone matrices which are updated every frame. Mesh updates are then directly applied on the GPU without further involving the CPU.

DOI: 10.2312/hpg.20141097

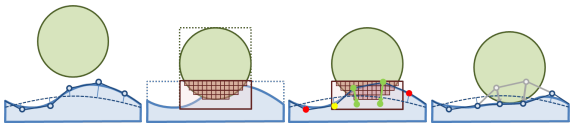

Figure 2: *Algorithm overview: deformable objects are represented as a subdivision surface with quadratic B-Spline displacements (left). For a pair of colliding objects, a voxelization of the overlap region is generated (center left). Displacement control points are pushed out of the rigid object (center right), resulting in the desired deformation (right).*

In this paper, we aim for a comparable procedure for local high-frequency deformations from collisions. Therefore, we separate out low-frequency deformations, which are computed and processed by the CPU physics simulation. Fine-detail deformations, however, are incorporated into a scalar-valued displacement function that is exclusively stored and updated on the GPU. This significantly reduces CPU-GPU memory I/O, allowing for higher frame rates, and enabling dynamic fine-scale surface deformations. To the best of our knowledge, our fine-scale deformation algorithm is the first that runs entirely on the GPU, and is thus orders of magnitude faster than all comparable CPU-based methods.

In order to represent deformable objects, we employ Catmull-Clark subdivision surfaces [CC78], which are defined by corresponding control cages called *base meshes*. Our approach also supports standard triangle meshes; however, the C^2 continuity of Catmull-Clark surfaces has significant performance advantages when updating surface displacements.

Base meshes, which may be skinned, are maintained by the CPU physics, and are coarse enough to enable real-time simulation. High-frequency surface offsets are stored in a tile-based GPU memory format where each tile corresponds to a subdivision surface base face. This enables storing and processing fine-scale deformations directly on the GPU without involving costly device-to-host memory I/O. Further, we interpret displacements as a bi-quadratic B-Spline scalar field that enables deriving surface normals analytically [NL13]. This allows for dynamic displacement updates without requiring expensive normal re-computations, which significantly differentiates us from previous approaches. In addition, we employ dynamic GPU memory management on the displacement tiles in order to minimize storage requirements [SKS13].

An overview of our algorithm is shown in Figure 2. For simplicity, we first consider the collision of a rigid object with a deformable one. The deformable object is represented as a displaced subdivision surface, whereas the rigid one can be an arbitrary subdivision surface or triangle mesh (Figure 2 left). For all pairs of objects with overlapping bounding boxes, we voxelize the rigid object within the overlap region using an extended variant of the real-time binary voxelization approach by Schwarz [Sch12] (Figure 2 center left). From the obtained voxelization, we determine the displacements of deformable objects to match the shape at the collision impact (Figure 2 center right and right). In the case that both objects are deformable, we form two collision pairs, with each deformable acting as rigid collider for the other deformable and only applying a fraction of the computed deformations in the first pass.

Our approach can handle collisions with arbitrary, dynamic, and animated objects. Since voxelizations are computed for every collision pair from scratch, the fine-scale deformations always match the corresponding mesh geometry. All of these steps are executed efficiently on the GPU, without transferring displacement information between host and device, resulting in an immediate response and minimal computational overhead.

In contrast to the fine-detail collision handling, the global physics simulation runs on the CPU, using the coarser base mesh. The base mesh is considered to be deformable and fully animated depending on the parameters of the physics simulation. Modified base meshes are uploaded to the GPU each frame in order to guarantee instant visual feedback based on physics updates.

Note that the computed fine-scale deformations are only used to achieve a visually richer rendering; they are *not* part of the physics simulation, which is solely executed on the CPU. We are not aiming for a complete physics simulation on the GPU, but for visually compelling fine-scale deformation effects.

Our key contribution is a real-time technique for fine-detail surface deformations on top of a global, potentially low-resolution, physics simulation. We show how to compute collision impacts with almost any kind of renderable object using voxelizations of the overlap regions, and how to update fine-scale deformations stored in displacement maps. Our approach combines methods for fast voxelization, efficient patch culling, ray casting, and hardware tessellation such that no costly GPU-CPU memory transfers are necessary. Even in complex scenes, we achieve instant response and visually appealing effects, as can be seen in Figure 1 and the accompanying video. Note that in all these examples the computational overhead to apply deformations is below a millisecond.

2. Previous Work

Real-time deformable surfaces: Deforming complex meshes based on accurate simulation of physics is computationally very expensive [Mil07]. In order to reduce the degrees of freedom, Galoppo et al. [GOM*06] run their simulation on scalar surface offsets, similar to Wrotek et al. [WRM05] who simulate and represent deformations in

bump maps. In our work, we omit accurate physics computation in favor of fine-scale detail, but adopt the idea of a dynamic deformation texture that can be accessed by the GPU for surface updates and subsequent rendering.

More recently, an online surface deformation approach has been proposed by Yusov [Yus12]. It makes use of the tessellation unit (as we do), but is restricted to height-field terrains only. Nykl et al. [NMC13] presented a method for simulating deformations on height fields using ray casting on box shapes. Another recent interactive sculpting and deformation method [SKS13] combines surface updates with a dynamic GPU memory management strategy. However, their surface deformations are limited to user-controlled (i.e., mouse input) sculpting stencils. In contrast, we incorporate the actual surface geometry, which is dynamically determined at runtime. This makes automatic object-object interaction (e.g., collision-based) feasible, and integration into existing physics systems easy.

Surface rendering with hardware tessellation: Hardware tessellation is ideal for rendering smooth surfaces while providing flexible level-of-detail control. In particular, subdivision surfaces (e.g., Catmull-Clark [CC78]) have been proven to be well suited to this platform [NLMD12, Nie13, SNK*14]. In order to incorporate high-frequency surface detail, displacements can be used [SKU08, TBB10]. When rendering displacements, great care must be taken to avoid cracks at patch boundaries and texture seams. Therefore, Burley and Lacewell [BL08] propose Ptex, an offline approach that stores adjacency pointers to enforce consistency. Mesh colors [YKH10] index all texture data and provide consistent texture access. Schäfer et al. [SPM*12] use a similar idea to generate a multi-resolution scene representation that is efficiently rendered using hardware tessellation. A Ptex-like GPU variant for (dynamic) displacement mapping has been proposed by Nießner and Loop [NL13]. The approach is based on subdivision surfaces and provides analytic surface normals for rendering. Our surface representation is based on this approach since it enables dynamic surface updates without expensive normal re-computation. In addition, we employ a dynamic GPU memory management strategy [SKS13, SKNS14] on the face textures to minimize the memory footprint for surface offsets storage.

Real-time voxelization: In order to identify fine-scale surface collisions, we voxelize object parts in overlapping regions which are given by the intersections of respective object bounding boxes. There are various real-time techniques for generating a suitable real-time voxelization on modern GPUs [DCB*04, ED06, ED08, SS10]. Our algorithm relies on a practical solid binary voxelization [Sch12], which can be efficiently determined in a single pass on the GPU. In order to identify potential deforming patch candidates in the overlapping region of two object bounding boxes, we cull patches against the region based on their spatial extent. In addition, we need to account for potential displacements by bounding the Catmull-Clark patch normals. For instance, this can be achieved by calculating the cone of normals as proposed by Shirmun and Abi-Ezzi [SAE93]. While this provides for accurate patch normal bounds, it is computationally expensive. Therefore, we approximate the cone of normals [SM88] similar to Munkberg et al. [MHTAM10] and Nießner et al. [NL12]. This is significantly faster than the accurate variant and provides similar quality bounds. Patches can be also culled by computing the parametric tangent plane [LNE11], i.e., determining whether a patch is front or back-facing. However, this is unsuitable for our needs, since this approach only applies to non-displaced surfaces.

3. Deformable Surface Representation

Our system stores fine-scale deformations as displacements applied to a coarse base mesh. To achieve high-quality displacements, a high resolution of the displacement values and smooth interpolation are necessary, both within patches and over patch boundaries. We thus use Catmull-Clark [CC78] subdivision surfaces, which provide smooth, high-quality surface detail. These surfaces can be rendered efficiently using the GPU hardware tessellation unit introduced with DX11 [Mic09]. More specifically, we render the true Catmull-Clark limit surface by employing feature adaptive subdivision [NLMD12]. We interpret displacements as control points of an analytic displacement function [NL13]. This is a scalar-valued, bi-quadratic B-Spline with (dual) Doo-Sabin [Doo78] connectivity and special treatment at extraordinary vertices. In principle, we could use any displacement mapping approach such as [SKU08, SPM*12]. The reason for our design choice is that analytic displacements (which are C^1 everywhere) provide continuous normals that can be evaluated on-the-fly. This allows for efficient displacement updates without costly normal re-computations (cf. Section 6). It also facilitates faster rendering than traditional displacement variants with normal maps.

The original analytic displacement mapping approach suggests storing the scalar-valued B-Spline coefficients in a tile-based texture format. That is, every patch of the Catmull-Clark base mesh corresponds to a rectangular tile domain with a fixed set of texels. In addition, every tile contains a one-texel overlap to enable evaluation and filtering at tile boundaries. We also pre-compute edge and corner overlap adjacency pointers, which are later used to enforce global displacement consistency after texture updates (see Section 6). To keep memory requirements low, we only allocate memory for patches that have been deformed, using a dynamic GPU memory management scheme [SKS13]. Note that analytic displacements efficiently handle lookup at vertices with irregular connectivity, thus enforcing data consistency (for details see [NL13]).

4. Physics Simulation

For the physics simulation, we employ the bullet physics library [C*06] which runs on the CPU. We use the control cage (or a modified triangular approximation) of the base surface as a physics representation. This handles the physics interactions between objects including collision detection and response, as well as rigid and possibly non-rigid surface deformations at the base surface level. Resulting kinematic updates are then sent to the GPU where the base mesh control points are updated by a compute kernel. While this runs efficiently in real-time, it lacks high-frequency deformation features due to the relatively low-detail base mesh representation. Thus, we additionally modify displacement offsets directly on the GPU to generate fine-scale surface deformations from collisions, as described in the next section.

These fine-scale surface deformations are only computed to generate visually convincing results, but are not part of the physics simulation. Thus, surface deformations do not generate friction and are not volume-preserving.

5. Deformation Detection

In order to determine fine-scale deformations, we compute overlap regions from bounding volumes of colliding scene objects. Geometry inside an overlap region is voxelized, and the resulting voxelization is later used to compute the deformation.

Overlap Regions We compute oriented bounding boxes (OBBs) for all scene objects using a principal component analysis (PCA) on all control points of the Catmull-Clark subdivision surface, which hold convex hull property. In addition, we incorporate a safety margin based on the maximum displacement extent. For rigid meshes or keyframe animations, we can pre-compute OBBs, whereas for animated meshes, e.g., skinning animations, we use a parallel reduction on the GPU [H*07] to determine OBB bounds. Once a collision has been detected, we intersect the bounding volumes of the collision pair to obtain a new OBB (*intersecting volume*) that conservatively bounds the overlapping region.

Culling Geometry within Overlap The overlap region is usually much smaller than the object's OBB, so we can gain significant performance by sorting out geometry outside this region. The following conservative culling approach runs entirely on the GPU: First, we compute axis-aligned bounding boxes (AABBs) for every patch in the space spanned by the intersecting volume. Since we use feature adaptive subdivision [NLMD12] for rendering, we obtain a nested set of bi-cubic B-Spline patches of the Catmull-Clark surface. In order to obtain tight patch bounds, we convert all B-Spline patches into Bézier basis and make use of the convex hull property. Note that this conversion is applied directly in the GPU culling kernel without the need for additional storage or memory I/O. Second, we incorporate displacement offsets

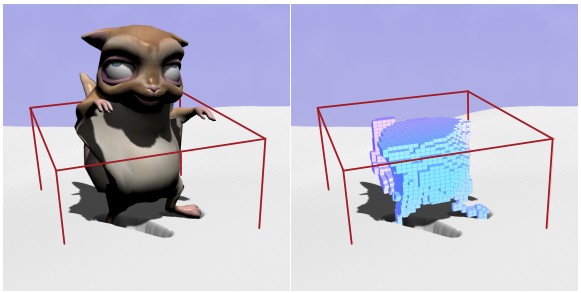

Figure 3: *Our approach first determines the intersecting volume between the oriented bounding boxes of the animated object (left) and the deformable object (i.e., the terrain below). Second, we voxelize patches within that intersection (right) in order to compute corresponding surface deformations. Note that per patch culling efficiently reduces the amount of patches that need to be voxelized and deformed.*

(if present) by computing a cone of normals following Sederberg and Meyers [SM88]. This approximate cone of normals variant has been proven to be computationally very efficient [MHTAM10, NL12, NSSL13] and provides similar quality to the accurate variant by Shirmun and Abi-Ezzi [SAE93]. Note that patches on all subdivision levels obtained by adaptive subdivision (cf. Section 3) are cull-tested against the intersecting volume. The cull decision of non-culled patches is propagated to faces of the base mesh. In order to reduce work on subsequent pipeline stages, we run a parallel compaction on the cull decision buffer. For patches of deformable objects, we use the obtained compacted culling information to reduce deformation update costs (see Section 6). For rigid objects, we only consider non-culled patches for the binary voxelization (see below).

Voxelization of Overlap For non-zero intersecting OBBs, we compute the binary voxelization of the corresponding rigid object following the approach of Schwarz [Sch12]. The voxel space is thereby defined by the base vectors of the intersecting volume. By using the previously determined compacted culling information, we can safely omit patches outside that volume to speed up computations (compare Figure 2 center left). However, clipping objects against intersecting volumes results in non-closed surfaces that cannot be handled by the original approach. In order to obtain a correct voxelization, we adopt the modifications proposed by Nießner et al. [NSSL13], i.e., we use forward-backward voxelization direction distinction and AABB extension. An example of a binary voxelization for a given intersecting volume is shown in Figure 3.

While we allocate a single voxelization buffer (shared by all objects) of a fixed size, we anisotropically scale the voxel grid with respect to the extent of a particular intersecting volume. In our examples, we use a budget of 2^{24} voxels, thus requiring about 2MB of GPU memory.

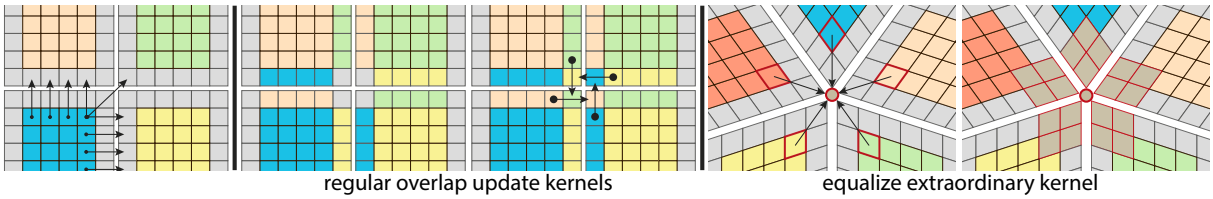

Figure 4: *Overlap update (from left): For regular patches, edge and corner data needs to be copied to adjacent patches. First, a GPU kernel copies edge data. Second, another GPU kernel copies corner data. Note that no diagonal adjacency pointers are required. Irregular patch configurations are similar, but require an additional processing step. We enforce displacement consistency at extraordinary vertices by incorporating all adjacent corner values. Therefore, we* equalize *all four neighboring texels of each patch (cf. right, gray colored) according to the average of the original corner values (second from right).*

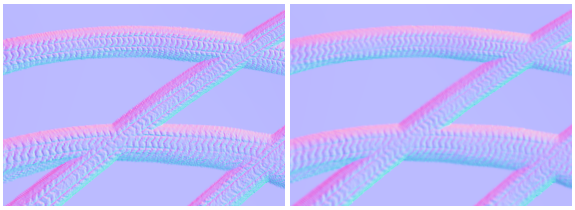

Figure 5: *The voxelization may cause discretization artifacts in the displaced surface as shown in the normal map visualization (left). Jittering the orientation of the intersected volume used for voxelization allows for a much better approximation of the true surface (right).*

6. Surface Deformation

In the previous step, we generated the voxelization of rigid objects for all non-zero intersecting volumes. Now, all patches of the deformable surface within the intersection volume are to be displaced such that they no longer intersect with the (voxelization of the) rigid object. Therefore, we look at all *control points* of the displacement B-Spline (of non-culled patches) and compute their corresponding world space positions. More precisely, these are Catmull-Clark surface points, evaluated at the knot points of the displacement B-Spline, with applied displacement (cf. Figure 2 center left). If such a control point lies within the deforming object, we move it in the negative base surface normal direction (we only have scalar displacements) until it leaves the deforming object (red control points in Figure 2). To this end, we cast a ray that originates at the control point's corresponding world space position and pointing along the negative base surface normal. Since this step involves evaluating the Catmull-Clark surface, we make use of the regular B-Spline patches obtained by adaptive subdivision. In addition, if there already exist patch displacements, we account for the previous surface offset. We now traverse the rays through the binary voxelization using a three-dimensional digital differential analyzer (DDA) [AW*87]. If the ray leaves the voxelization in the first traversal step, we assume that we started at the boundary and perform no displacement update at all. Otherwise, the negative traveled distance is added to the current displacement, such that the new control point is outside of the voxelization, and thus outside of the deforming object. Control points outside of the voxelization (yellow one in Figure 2) and outside of the overlap region (red ones) are left unchanged.

Note that we do not fit the displaced surface. Instead, we directly constrain the scalar-valued control points. Hence, the bi-quadratic B-Spline function is approximated rather than interpolated. This avoids typical ringing artifacts caused by B-Spline interpolation, and results in more realistic displacements. The small remaining overlap is hardly visually noticeable, but the boundaries look much more convincing and the computation is much simpler. One could also apply further smoothing on displacements in order to reflect the underlying material properties. This is trivial since patch tiles store a one texel overlap [NL13].

Our system relies on the voxelized approximation of an object instead of the true surface when computing displacement offsets. This approximation may cause discretization artifacts as shown in Figure 5 (left). In order to avoid these artifacts, we randomly jitter the direction which is used to generate the voxelization. That is, in every frame, we slightly rotate the intersecting volume. Note that we conservatively bound the rotated volume such that the original volume is fully contained. As a result, we obtain a better approximation of the true surface as shown in Figure 5 (right).

Once surface deformations are computed, we need to update tile overlap in order to enforce displacement data consistency (e.g., for rendering). Therefore, we employ precomputed edge and corner adjacency tables (cf. Section 3) in a set of GPU compute kernels. That is, we need to copy edge and corner data to neighboring tiles (Figure 4). Instead of storing adjacency information to diagonal patches, corners are updated by two consecutive kernels using only direct neighbor adjacency. Note that we only propagate displacement information originating from updated tile textures using the compacted cull-decision buffer, which greatly reduces runtime overhead. After updating edge and corner overlap, we provide consistent surface evaluation at extraordinary vertices by equalizing corner displacements in an extra kernel that is executed for all extraordinary vertices (see Figure 4 right).

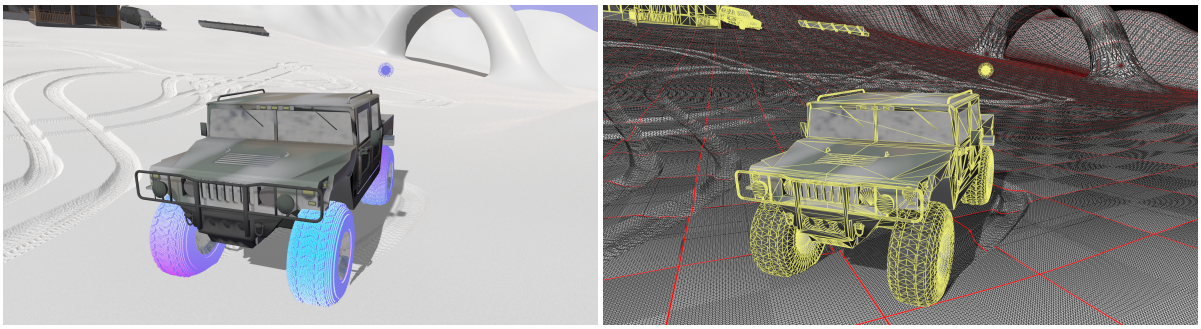

Figure 6: *Our technique applied to a driving car on snowy terrain. Collisions from the car wheels cause surface deformations on the terrain. Deformations are determined based on the voxelization of deforming objects. In contrast to standard decal or bump mapping approaches, we modify the actual surface geometry, as shown in the wireframe rendering (right).*

7. Results

We implemented our approach using DirectX 11 with all shaders written in HLSL running on an NVIDIA GTX 780. While we use the standard graphics pipeline for visualization (i.e., rendering with hardware tessellation) and voxelization, we employ compute shaders for culling, compaction, ray casting the voxelization (DDA), and for updating the displacement tile overlap. For the physics simulation we employ the bullet library [C*06]. In the following, we provide a comparison of visual quality with a set of test scenes and performance measurements in milliseconds. Unless otherwise mentioned, we use a tile size of 128×128 in our examples.

Example Scenes The first example scene, shown in Figure 1 (left), is a snowy, deformable terrain with static geometry such as houses and trees. In addition, the scene also includes dynamic objects, such as a car and barrels, which are simulated using the bullet physics engine [C*06]. Overall, the scene contains about 150k triangles (non-deformable) and 8800 patches for the deformable terrain. Our approach allows for fine-scale deformations solely based on the geometric representation and physical behavior of the car. As a result, the car wheels leave tracks in the snow field, incorporating both geometric and surface normal deformations. Compared to standard decal rendering or bump mapping, our approach updates the actual surface geometry. Figure 6 shows a close-up of the car model on the snow field where the deformations can be seen in the wireframe rendering. In Figure 7, we also show an example with fine-scale surface detail, where the Spikeball causes deformations at all contact points with the snow field. In the same scene, a skinning animation character is used to deform the surface, as shown in Figure 1 (right). Again, we refer to the accompanying video for the full sequence.

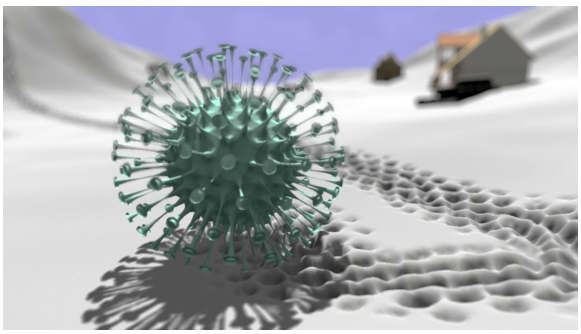

Figure 7: *Example of high-frequency deformation on surface: the rolling high-detail spike ball leaves a geometric trail on the traversed surface.*

The second scene (see Figure 9) shows the *Monsterfrog* model sculpted using different geometric tools. The frog model consists of 1292 base mesh patches which corresponds to 41274 render patches at feature adaptive subdivision level six.

Performance Measurements A performance overview for our test scenes (see above) is provided in Table 1. We measure performance for different patch tile sizes (displacement map resolution), break out timings for different pipeline stages of our algorithm, and summarize the resulting overhead to apply deformations per colliding object. Note that each pipeline stage corresponds to a particular draw call or GPU compute kernel.

We provide timings for patch culling, compaction and object voxelization, which need to be determined for each non-zero intersecting volume (i.e., each collision pair). Finally, we show timings for calculating the depth penetration of the voxel volume into deformable objects via ray casting, as well as measurements for the overlap updates of the displacement tiles once surface deformations have been computed. The measurements in Table 1 show that the choice of the per-patch tile size has a direct impact on ray casting time. While more displacement control points must be con-

Scene	Terrain				Frog			
base patches	8769				1292			
render patches	12679				41274			
draw subD	0.563				0.374			
tile size	32^2	128^2	128^2 no culling	128^2 no compaction	32^2	128^2	128^2 no culling	128^2 no compaction
culling	0.164	0.164	-	0.164	0.368	0.368	-	0.368
compaction	0.005	0.005	-	-	0.004	0.004	-	-
voxelization	0.006	0.006	0.006	0.006	0.006	0.006	0.006	0.006
ray casting	0.053	0.073	10.11	0.073	0.083	0.114	45.87	0.114
overlap	0.025	0.025	0.228	0.228	0.025	0.028	0.178	0.178
sum overhead	0.254	0.273	10.31	0.471	0.486	0.521	46.054	0.666

Table 1: *Performance measurements on test scenes for different per-patch tile resolutions in milliseconds. For each colliding object pair the deformable culled against the overlapping region followed by stream compacting the cull decision results. Then the colliding rigid object is voxelized. Then rays are cast from the deformable object into the voxel volume to determine the penetration depth and stored as displacement. Finally, the boundaries of the tile storage scheme are updated.*

sidered for a higher tile resolution during deformation computation, the increase of time is less than expected. We attribute this to the low GPU occupancy since only a few tiles are affected, leaving us with the option to further increase the tile resolution. The most expensive step of our deformation pipeline is patch culling, since all patches need to be processed for cull testing. Without culling and a 128×128 tile size, ray casting takes 10.11 ms and 45.87 ms for the terrain and frog scene, respectively. That is several orders of magnitudes slower than the default setting with culling enabled. Thus, culling is easily paying off the additional computational overhead.

Updating displacement tile overlap has only marginal impact, as only actual deformed patches need to propagate tile boundary texels to the neighbors. Without stream compacting the culling buffer, overlap computation takes 0.228 ms and 0.178 ms for the terrain and monsterfrog scene, respectively. Thus, the additional overhead of the compaction step is also easily amortized.

Overall, the overhead of our deformation approach is well below a millisecond for our test scenes (with our standard tile size of 128×128) which makes it ideal for real-time applications. We also never encountered any performance fluctuations since timings were constant up to $\pm 2\%$. In Figure 8, we show a comparison of the resulting deformation quality using different tile resolutions for the trail of a car wheel on the terrain. We require less memory than traditional displacement mapping approaches since no normal map is required (16 bit float values per displacement value are sufficient). Memory overhead for adjacency pointers and voxelization buffers is negligible (cf. Section 5).

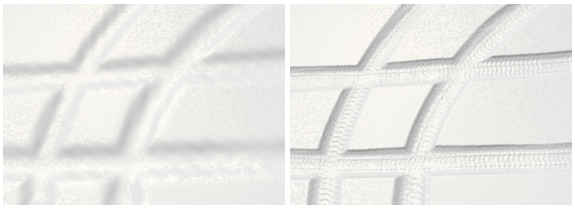

Figure 8: *Comparison of resulting deformation using a 16x16 (left) and 128x128 (right) per patch tile resolutions.*

Sculpting Demo Our method supports the deformation of subdivision surfaces with existing displacement as shown with the *Monsterfrog*. As an example, we show a demo where we use meshes as a deformation tools to modify a given surface; see Figure 9. The tools consist of triangle and subdivision surface meshes.

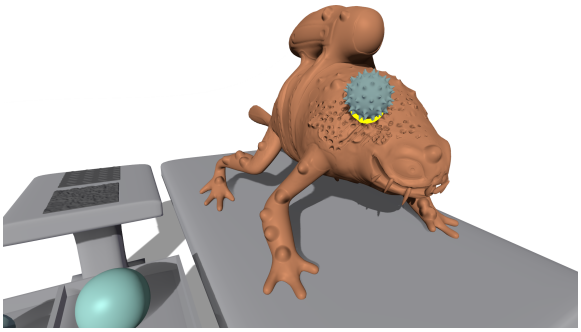

Figure 9: *Deformation on a subdivision surface with existing displacements using a pen-device-controlled sculpting geometry. For constraining the penetration depth, the pressure of the pen device is used to apply forces to the tool controlled by the physics engine.*

8. Conclusion

In this paper, we presented a system for the real-time computation of deformations on subdivision surfaces from collisions with dynamic and possibly animated meshes. We show how deformations, represented as displacements, can be updated on-the-fly very efficiently using a novel tool chain running completely on the GPU, without costly memory transfer. With our approach, we achieve deformations even in complex scenes well below render time, so we believe it is well suited for many interactive applications such as video games or authoring tools.

However, it must be emphasized that the generated deformations are only aiming at a more detailed and dynamic visual appearance, but cannot be considered as a physics simulation. We also do not consider elasticity, volume preservation, friction or topological changes. We believe that these current limitations make for interesting future directions.

Acknowledgements

This work is co-funded by the German Research Foundation (DFG), grant GRK-1773 Heterogeneous Image Systems.

References

[AW*87] AMANATIDES J., WOO A., ET AL.: A fast voxel traversal algorithm for ray tracing. In *Proc. EG'87* (1987), vol. 87, pp. 3–10. 5

[BL08] BURLEY B., LACEWELL D.: Ptex: Per-Face Texture Mapping for Production Rendering. *CGF 27*, 4 (2008), 1155–1164. 3

[C*06] COUMANS E., ET AL.: Bullet physics library. *Open source: bulletphysics.org* (2006). 4, 6

[CC78] CATMULL E., CLARK J.: Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-Aided Design 10*, 6 (1978), 350–355. 2, 3

[DCB*04] DONG Z., CHEN W., BAO H., ZHANG H., PENG Q.: Real-time voxelization for complex polygonal models. In *Computer Graphics and Applications* (2004). 3

[Doo78] DOO D.: A Subdivision Algorithm for Smoothing Down Irregularly Shaped Polyhedrons. In *Proceedings on Interactive Techniques in Computer Aided Design* (1978), IEEE, pp. 157–165. 3

[ED06] EISEMANN E., DÉCORET X.: Fast scene voxelization and applications. In *Proc. I3D'06* (2006). 3

[ED08] EISEMANN E., DÉCORET: Single-pass GPU solid voxelization for real-time applications. In *Proc. Graphics Interface '08* (2008). 3

[GOM*06] GALOPPO N., OTADUY M. A., MECKLENBURG P., GROSS M., LIN M. C.: Fast simulation of deformable models in contact using dynamic deformation textures. In *Proc. Computer Animation '06* (2006), pp. 73–82. 2

[H*07] HARRIS M., ET AL.: Optimizing parallel reduction in CUDA. *NVIDIA Developer Technology 2* (2007). 4

[LNE11] LOOP C., NIESSNER M., EISENACHER C.: Effective back-patch culling for hardware tessellation. In *Proc. VMV'11* (2011). 3

[MHTAM10] MUNKBERG J., HASSELGREN J., TOTH R., AKENINE-MÖLLER T.: Efficient bounding of displaced bézier patches. In *Proc. HPG'10* (2010). 3, 4

[Mic09] MICROSOFT CORPORATION: Direct3D 11 Features, 2009. http://msdn.microsoft.com/en-us/library/ff476342(VS.85).aspx. 3

[Mil07] MILLINGTON I.: Game physics engine development, 2007. 2

[Nie13] NIESSNER M.: *Rendering Subdivision Surfaces using Hardware Tessellation*. Dissertation. Dr. Hut, 2013. 3

[NL12] NIESSNER M., LOOP C.: Patch-based occlusion culling for hardware tessellation. In *Proc. CGI'12* (2012), vol. 2. 3, 4

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

[NLMD12] NIESSNER M., LOOP C., MEYER M., DEROSE T.: Feature Adaptive GPU Rendering of Catmull-Clark Subdivision Surfaces. *ACM Transactions on Graphics* (2012). 3, 4

[NMC13] NYKL S., MOURNING C., CHELBERG D.: Interactive Mesostructures. In *Proc. I3D'13* (2013), pp. 37–44. 3

[NSSL13] NIESSNER M., SIEGL C., SCHÄFER H., LOOP C.: Real-time Collision Detection for Dynamic Hardware Tessellated Objects. In *EG short papers* (2013), Eurographics. 4

[SAE93] SHIRMUN L., ABI-EZZI S.: The cone of normals technique for fast processing of curved patches. *CGF 12*, 3 (1993). 3, 4

[Sch12] SCHWARZ M.: Practical binary surface and solid voxelization with Direct3D 11. In *GPU Pro 3*. A K Peters/CRC Press, 2012. 2, 3, 4

[SKNS14] SCHÄFER H., KEINERT B., NIESSNER M., STAMMINGER M.: Local Painting and Deformation of Meshes on the GPU. *CGF* (2014). 3

[SKS13] SCHÄFER H., KEINERT B., STAMMINGER M.: Real-time Local Displacement using Dynamic GPU Memory Management. In *Proc. HPG'13* (2013), ACM. 2, 3

[SKU08] SZIRMAY-KALOS L., UMENHOFFER T.: Displacement Mapping on the GPU-State of the Art. *CGF 27*, 6 (2008), 1567–1592. 3

[SM88] SEDERBERG T., MEYERS R.: Loop detection in surface patch intersections. *Computer Aided Geometric Design 5*, 2 (1988). 3, 4

[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. 3

[SPM*12] SCHÄFER H., PRUS M., MEYER Q., SÜSSMUTH J., STAMMINGER M.: Multiresolution attributes for tessellated meshes. In *Proc I3D'12* (2012), ACM, pp. 175–182. 3

[SS10] SCHWARZ M., SEIDEL H.: Fast parallel surface and solid voxelization on gpus. *Transactions on Graphics 29*, 6 (2010). 3

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

[WRM05] WROTEK P., RICE A., MCGUIRE M.: Real-time collision deformations using graphics hardware. *Journal of Graphics, GPU, and Game Tools 10*, 4 (2005), 1–22. 2

[YKH10] YUKSEL C., KEYSER J., HOUSE D. H.: Mesh colors. *ACM Transactions on Graphics 29*, 2 (2010), 15. 3

[Yus12] YUSOV E.: Real-Time Deformable Terrain Rendering with DirectX 11. *Gpu Pro 3: Advanced Rendering Techniques 3* (2012), 2. 3