Real-time Collision Detection for Dynamic Hardware Tessellated Objects

M. Nießner[1] and C. Siegl[1] and H. Schäfer[1] and C. Loop[2]

[1]University of Erlangen-Nuremberg
[2]Microsoft Research

Abstract

We present a novel method for real-time collision detection of patch based, displacement mapped objects using hardware tessellation. Our method supports fully animated, dynamically tessellated objects and runs entirely on the GPU. In order to determine a collision between two objects, we first find the intersecting volume of the corresponding object oriented bounding boxes. Next, patches of both objects are tested for inclusion within this volume. All possibly colliding patches are then voxelized into a uniform grid of single bit voxels. Finally, the resulting voxelization is used to detect collisions. Testing two moderately complex models containing thousands of patches can be done in less than a millisecond making our approach ideally suited for real-time games.

1. Introduction

Hardware tessellation allows efficient rendering of smooth surfaces, including subdivision surfaces, by processing patch primitives in parallel. On top of a smooth base surface displacement maps can be added to provide high-frequency geometric detail. Surfaces can be animated by updating only the patch control points while displacement values remain constant. Another benefit is the realization of level-of-detail schemes by computing patch tessellation density at runtime. However, this also makes collision detection a challenging task since geometry is generated on-the-fly by the GPU. Transferring geometry to the CPU would involve significant memory I/O and is not feasible in real-time applications. Thus, traditional collision detection approaches that maintain a hierarchy of proxy primitives are too costly and provide only loose results. To the best of our knowledge, providing a satisfactory collision detection scheme for hardware tessellation has not previously been done.

We tackle this problem by considering the geometry that is actually used for rendering. Surface geometry in our case is based on dynamic tessellation factors and displacement values. The first step of our approach is to test two objects for collision by testing their oriented bounding boxes (**OBBs**) for intersection. If there is no intersection, there can be no collision (early exit); otherwise we compute the intersection of the OBBs. Next, we determine all patches that lie within this shared volume by performing an inclusion test. Included patches are then voxelized into identical grids, one for each object. Finally, the resulting voxelizations of both objects are compared to determine collisions. We can also determine collision positions and their normals based on these voxelizations. We provide an extension of our method that will report colliding patch IDs and uv coordinates. These can be used to evaluate object patches at collision points for accurate physics handling.

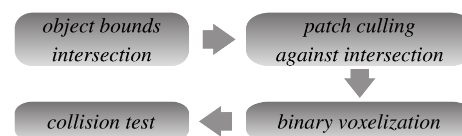

Figure 1: *Overview of our collision detection approach.*

For simplicity, we demonstrate our approach using bicubic Bézier patches. However, our algorithm can be applied to any patching scheme (e.g., [LSNC09], [NLMD12]) whose patches can be bounded. An overview of our algorithm pipeline of is shown in Figure 1. Our experimental results show that a collision test between two objects consisting of thousands of patches can be performed in less than a millisecond (see 1). This is well within the processing budget of real-time applications such as video games. While we currently use our own rudimentary physics simulation for demonstration, our approach could be also integrated into any physics engine.

To sum up, our approach allows

- real-time collision detection for hardware tessellation
- supports animated objects with displacements

© The Eurographics Association 2013.

DOI: 10.2312/conf/EG2013/short/033-036

delivered by EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

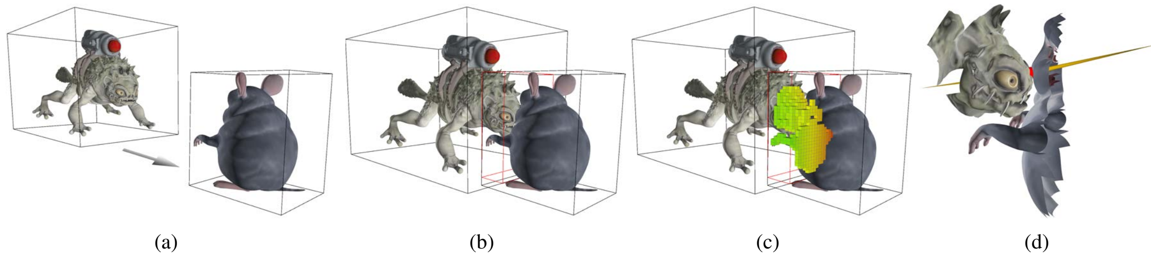

Figure 2: *Simple test setup visualizing our approach with the Frog moving towards the Chinchilla (a). At one point the OBBs of both objects intersect (b, red box) and we voxelize the containing geometry (c). This allows us to determine the collision point and corresponding surface normals (d). Patches shown in the last image could not be culled against the intersecting OBB and thus are potential collision candidates contributing to the voxelization.*

2. Previous Work

Collision detection: An essential part of physics simulations (e.g., in games [Mil07]) is the ability to detect collisions. Surveys of traditional approaches are shown in [JTT01], [Eri04].

Bounding and culling of displaced patches: A key feature of our approach is to reduce the number of possibly colliding patches using patch-based culling. Therefore, patch normals need to be bound. This can be achieved by computing an accurate [SAE93] or approximate [SM88] cone of normals. Munkerberg et al. [MHTAM10] and Nießner et al. [NL12] make use of the approximate variant since resulting quality is similar but computational costs are an order of magnitude smaller. While we use the same normal bounds, our application is different since we cull against a shared volume given as a bounding box.

Real-time voxelization: A binary voxelization is a memory efficient representation to distinguish between empty and occupied space. It can be obtained in different ways on modern GPUs in real-time [DCB*04], [ED06], [ED08]. More recent approaches also provide for conservative voxelizations [SS10]. However, a conservative method does not fit our needs since it would be computationally too expensive. Instead we rely on the more practical solid binary voxelization proposed by Schwarz [Sch12]. In contrast to his approach we do not voxelize closed meshes; therefore we modify the approach accordingly.

3. Collision Candidates

A crucial part of our approach is to first reduce the number of potentially colliding patches. Therefore, we first consider the collision of the respective object bounds. Since we use bi-cubic Bézier patches we are able to obtain bounds using the convex hull property. Note, we could also use other surface types such as B-splines, subdivision surfaces, or triangle meshes. We compute the OBBs by applying principal component analysis (**PCA**) on the patch control points.

For the collision test between two objects we then determine the intersection of the two corresponding OBBs. If there is no intersection, there is no collision (early exit). Otherwise, we obtain a set of contact points by intersecting the faces of one box with the edges of the other and vice versa. OBB corner points that are inside the other box are also considered to be contact points. We then compute a new OBB based on the obtained contact points; again by using PCA. In some cases we may need to enlarge the intersecting OBB in order for our binary voxelization to work properly (see Section 4). For at least one pair of opposite faces, one face must be outside of one input object OBB and the opposite face must be outside of the other OBB (see Figure 3). We select the OBB axis that minimizes this enlargement.

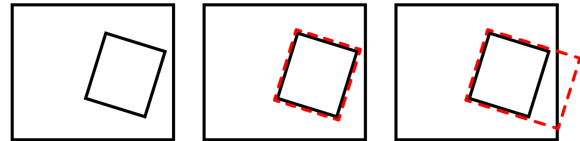

Figure 3: *OBBs of two objects (black boxes, left) and joint volume (dotted red box). Extension of joint volume to ensure that one face of is outside of the OBB (right).*

A compute kernel is used to determine patches that are included in the intersecting OBB in parallel. Each thread processes one patch and computes its OBB. This is done by transforming patch control points into the space of the intersecting OBB; in that space the OBB is the unit cube. As a result we obtain axis-aligned bounding boxes (**AABB**) in the respective OBB space per patch. If an AABB is outside of the unit cube a patch can be culled. However, we also need to incorporate displacements into the patch AABB computation. For each patch we compute a cone of normals that bounds the patch normals. Therefore, an exact [SAE93] or approximate [SM88] but also conservative method can be taken into account. We rely on the approximate variant since its computation is an order of magnitude less expensive and provides similar results. Given the cone axis $\vec{a}$ and cone aperture α we enlarge the bounding box of a patch according to the minimum $D_{\min}$ and maximum $D_{\max}$ displacement value (see [NL12]):

© The Eurographics Association 2013.

$$\begin{array}{ll} if(\vec{a}_x \geq \cos(\alpha)) & \delta_x^+ = 1 \\ else & \delta_x^+ = \max(\cos(\arccos(\vec{a}_x)+\alpha), \cos(\arccos(\vec{a}_x)-\alpha)) \end{array}$$

$$\begin{array}{ll} if(-\vec{a}_x \geq \cos(\alpha)) & \delta_x^- = 1 \\ else & \delta_x^- = \max(\cos(\arccos(-\vec{a}_x)+\alpha), \cos(\arccos(-\vec{a}_x)-\alpha)) \end{array}$$

This allows to modify the patch AABBs:

$$\begin{array}{l} AABB_{\max} = AABB'_{\max} + \max(D_{\max} \cdot \delta^+, -D_{\min} \cdot \delta^-) \\ AABB_{\min} = AABB'_{\min} - \max(D_{\max} \cdot \delta^-, -D_{\min} \cdot \delta^+) \end{array}$$

We precompute displacement extrema $D_{\min}$ and $D_{\max}$ per patch since displacements are considered to be static.

Note that it would be feasible to orient patch bounding boxes based on patch normals [MHTAM10]. However, we obtain better results if both patch and object intersecting OBB share the same axes. In the end we are able to identify all patches that are within the common bounding volume intersection. Only those patches need to be considered for collision detection; we mark those by maintaining a flag list that contains a single binary value per patch.

4. Voxelization

The key idea of our collision test is to voxelize the rendering geometry of the current frame. For two objects those patches lie within the intersection of the objects' bounding boxes (see Section 3). We then setup an orthogonal camera matrix that spans that space and perform a solid binary voxelization. Therefore, we use a modified version of the algorithm proposed by Schwarz [Sch12]. The voxelization is performed within the rasterization pipeline and can be applied to any object with or without displacements. Since DirectX 11 (or OpenGL 4.0 or above) allows scattered memory writes in the pixel shader there is no need for a render target. The voxel grid is represented as a linear buffer of 32 bit integer values allowing us to store 32 voxels per entry. In the pixel shader we compute the voxel index depending on the x, y and depth value. For each fragment atomic XOR-operations are used to flip all voxels behind that fragment. Since this is an integer operation, we process 32 voxels per instruction. In the end only voxels within the volume will remain set resulting in a solid voxelization.

In contrast to the binary voxelization by Schwarz, our potentially colliding patches may not form a closed surface. Therefore, we construct the intersecting OBB to have at least one face that lies completely outside of the original object OBB (see Section 3). Thus, we can fill voxels towards the opposite face. We use two different kernels to perform the solid voxelization and fill voxels backwards or forward, respectively. Pseudo code of our backward voxelization kernel is shown below (the forward kernel is similar):

```
//Backward solid voxelization
addr = p.x * stride.x + p.y * stride.y + (p.z >> 5) * 4;
atomicXor(voxels[addr], ~(0xffffffff << (p.z & 31)));
for (p.z = (p.z & (~31)); p.z > 0; p.z -= 32) {
      addr -= 4;
      atomicXor(voxels[addr], 0xffffffff);
   }
}
```

Note that backface culling must be turned off for the voxelization. Performance scales linearly with the number of patches that need to be voxelized making patch culling as described in Section 3 essential. The resolution of the voxel grid is adaptive and computed based on the size of the intersecting OBB. However, we require the resolution in z-direction to be a multiple of 32 to align with four byte integer values. The use of a solid instead of a surface voxelization is essential. It prevents from missing collisions where objects entirely penetrate a surface within one time step. In addition, a resulting voxelization that is close to the original mesh can be obtained with a single render pass. In some cases we may loose one voxel width around the visual hull due to non-conservative rasterization.

5. Collision Detection

We perform collision detection based on binary voxelizations as shown in Section 4. Our basic approach is to determine collision positions and corresponding normals based on the voxelization. In addition, we propose an extended variant that provides patch IDs and uv coordinates of collision points; e.g., for accurate surface re-evaluation.

5.1. Basic Collision Test

Determining collisions given two solid voxelizations that occupy the same space can be obtained by performing pairwise voxel comparisons. There is a collision if equivalent voxels are set. Since our voxel representation is binary, we can perform 32 voxel comparisons using a single bitwise AND-operation of the two corresponding integer values. Therefore, we use a compute kernel with one thread for each integer value of the linear voxel buffer (each value contains 32 binary voxel entries). In addition to collision positions, we obtain corresponding normals based on voxel neighborhoods. Collisions are written into a GPU buffer using atomics that can be accessed from the CPU.

5.2. Extended Collision Test

The extended collision test first executes the basic variant. Then another pass is used to obtain patch IDs and uv coordinates of collision points. Therefore, we use an orthographic camera setup that conservatively contains the intersecting OBB and points in the direction of the average approximate collision normal obtained from the basic collision test. Thus, a maximum number of fragments is generated near collision points. Passing patch IDs and uv coordinates to the pixel shader allows us to store these in a global linked list with an atomic counter (i.e., append buffer in DirectX 11). This allows us to accurately evaluate surface geometry on the CPU at all collision points and to determine corresponding attributes such as surface normals. In order to obtain collision attributes for both colliding objects, we must perform this test twice; once for each object while testing against the voxelization of the other object, respectively.

© The Eurographics Association 2013.

6. Results

Our implementation uses the DirectX 11 graphics pipeline and compute shaders. Performance measurements were made using an NVIDIA GeForce GTX 680 and an Intel Core i7 at 2.80 GHz. In our examples we use the same tessellation density for collision detection that is used for rendering.

Figure 2 shows a simple test scene with two objects. The Frog (w/ displacements) is moving (a) towards the Chinchilla until we detect a collision (b). We also visualize the voxelization that we use for collision computation (c). In addition, patches that could not be culled against the intersecting OBB and the obtained collision point with its corresponding surface normals is shown (d). We used a relatively low voxel grid resolution for visualization purposes in this figure. For all our tests we use a volume of 128^3 adapted anisotropically to the intersecting OBB requiring no more than 256KB of GPU memory.

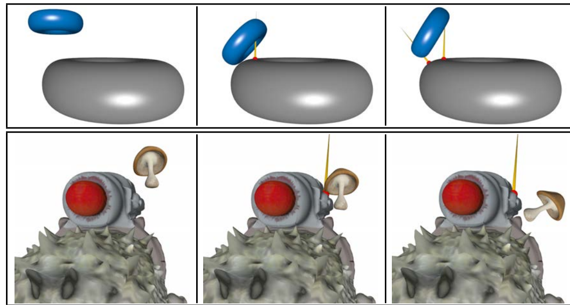

Figure 4: *Basic physics using our collision detection approach. In each sequence one object is falling onto another.*

In order to validate the practicality of our approach we implemented a rudimentary physics engine. We perform physics computations based on the collisions and corresponding attributes of our extended collision test. Figure 4 shows two simple examples. In the first sequence a torus is falling until it hits its counterpart, then rotating slightly, hitting the other torus again and bouncing off. In the second sequence a mushroom is falling on the Frog model with only a single hit point. Note that there are two normals at collision points (one for each object), however, the visualization is occluded by the larger objects, respectively. More examples are provided in the supplemental video.

Performance results of our approach for two setups are shown in Table 1. With the presented approach, we are able to perform both the basic and the extended collision test including all overhead in less than a millisecond.

7. Conclusion and Future Work

We have presented a method for real-time collision detection of dynamic hardware tessellated objects with displacements. Our approach considers the rendering geometry of the current frame and runs entirely on the GPU. We have shown

	Frog / Chinchilla		Frog / Mushroom	
Patches	1292 / 4270		1292 / 744	
Tess Factor	4	8	4	8
Draw	0.065	0.169	0.024	0.067
OBB Intersect	0.008	0.008	0.011	0.011
Patch Culling ×2	0.016	0.016	0.012	0.012
Voxelization ×2	0.059	0.092	0.037	0.053
Collision	0.117	0.117	0.114	0.114
Collision Attributes ×2	0.210	0.262	0.222	0.232
Sum Test Basic	**0.275**	**0.341**	**0.223**	**0.255**
Sum Test Extended	0.695	0.865	0.667	0.719

Table 1: *Performance measurements in milliseconds of our approach for the two test scenes (see Figures 2 and 4). We provide numbers for rendering (Draw) and for each step of our collision pipeline (OBB Intersect, Patch Culling, Voxelization, Collision, Collision Attributes). In addition, we show the overall performance for a pairwise collision test for both the basic and extended variant of our approach.*

that collision tests can be performed with minimal overhead and that our method can be used for real-time physics. We can perform both basic collision tests and extended collision tests that find attributes at collision positions. We have also shown that the overhead of our approach for objects containing thousands of patches is less than a millisecond.

References

[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). 2

[ED06] Eisemann E., Décoret X.: Fast scene voxelization and applications. In *Proc. I3D'06* (2006). 2

[ED08] Eisemann E., Décoret X.: Single-pass gpu solid voxelization for real-time applications. In *Proceedings of graphics interface* (2008). 2

[Eri04] Ericson C.: *Real-time collision detection*. Morgan Kaufmann, 2004. 2

[JTT01] Jiménez P., Thomas F., Torras C.: 3D collision detection: a survey. *Computers & Graphics 25*, 2 (2001). 2

[LSNC09] Loop C., Schaefer S., Ni T., Castaño I.: Approximating subdivision surfaces with gregory patches for hardware tessellation. *ACM Trans. Graph. 28*, 5 (2009). 1

[MHTAM10] Munkberg J., Hasselgren J., Toth R., Akenine-Möller T.: Efficient bounding of displaced bézier patches. In *HPG* (2010). 2, 3

[Mil07] Millington I.: *Game physics engine development*. Morgan Kaufmann Pub, 2007. 2

[NL12] Niessner M., Loop C.: Patch-based occlusion culling for hardware tessellation. *Computer Graphics International* (2012). 2

[NLMD12] Niessner M., Loop C., Meyer M., Derose T.: Feature-adaptive gpu rendering of catmull-clark subdivision surfaces. *ACM Trans. Graph. 31*, 1 (2012). 1

[SAE93] Shirmun L., Abi-Ezzi S.: The cone of normals technique for fast processing of curved patches. In *CGF* (1993), vol. 12. 2

[Sch12] Schwarz M.: Practical binary surface and solid voxelization with Direct3D 11. In *GPU Pro 3*. 2012. 2, 3

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

[SS10] Schwarz M., Seidel H.: Fast parallel surface and solid voxelization on gpus. In *ACM Trans. Graph.* (2010), vol. 29. 2

© The Eurographics Association 2013.