Real-time Rendering of Small-scale Volumetric Structure on Animated Surfaces

Artemiy Leshonkov [1] and Vladimir Frolov [1,2]

[1] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russia*
[2] *Keldysh Institute of Applied Mathematics, Miusskaya sq., 4, Moscow, 125047, Russia*

Abstract

There are a lot of methods for rendering of shell-space geometry, represented through voxel texture, known for today. While the topic is well studied in terms of techniques for applying this geometry onto surfaces, a little attention was paid to representation of sub-pixel details of the geometry. Such details are prone to produce aliasing artifacts and reduce performance due to bad cache utilization. In this paper we solve these problems by introducing levels of detail for voxel textures within shell mapping technique. The main problem here is that less detailed levels begin to contain semi-transparent voxels on the edge of an encoded surface, which requires additional handling. For this we present a new approach for order independent transparency rendering based on depth peeling. We extend the algorithm by adding additional resolving pass which allows to fully utilize hardware z-buffering to reduce amount of overdraw. This significantly reduces cost of each subsequent peeling pass. Empirically, 3-4 of such passes is enough to produce good quality results in most cases. Another issue with shell mapping techniques is that shell geometry is constructed offline, making base surface to be static. By slightly modifying the method, we made the construction to be performed on-the-fly on GPU and be applicable for animated surfaces.

Keywords

real-time rendering, shell mapping, levels of detail, order independent transparency

1. Introduction

Simulation of small-scale features on a surface is important task in many real-time applications (fig.1). In garment prototyping there are knitwear, fur and other complex materials. In videogames and, for example, flight simulators there is often a need to render grass, forests or other plants at far distance. In architecture we often have surfaces with small repetitive patterns, such as fences, bricks etc. Usually it's being done by simply applying 2D texture, representing such details, or by rendering lots of textured quads. Such approach is fast, but coarse. So, there are multiple techniques to accurately represent complex structure on surfaces either directly by geometry [1-3] or implicitly through volumetric texturing [4-6] without need to construct all of the details by hand.

Simulation through geometry allows us to get accurate results by the cost of vertex processing and memory consumption. For example, to render a chainmail using geometry, we have to model each ring through primitives and then process all of their vertices during rasterization. The more rings we have, the more vertices we need to process, thus making rendering to be expensive when rings are small and there is too much of them.

GraphiCon 2021: 31st International Conference on Computer Graphics and Vision, September 27-30, 2021, Nizhny Novgorod, Russia
EMAIL: artemiy.leshonkov@graphics.cs.msu.ru (A. Leshonkov); vfrolov@graphics.cs.msu.ru (V. Frolov);
ORCID: 0000-0003-2144-7250 (A. Leshonkov); 0000-0001-8829-9884 (V. Frolov);

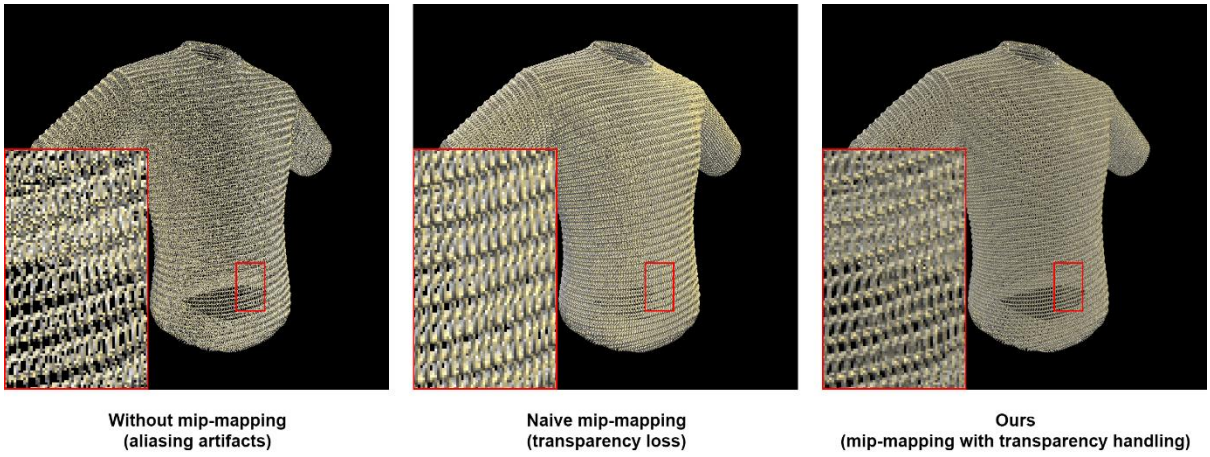

Figure 1: Demonstration of our approach for small-scale details rendering

In the case of implicit representation of structure through a volumetric texture, either procedural or as three-dimensional array of texels, detail applying becomes similar to usual texture mapping and there is no need to process vertices of the details themselves. Therefore, the rendering time is defined by the amount of pixels being rasterized and vertex count of the surface, to which details are applied. Further we will refer to such surface as a base surface or a base mesh as in [4]. As example, we can take only small voxelized piece of, again, a chainmail and repetitively apply it to the base surface as a texture, without need to simulate each ring separately.

So, in theory, we can simulate repetitive volumetric structure on a surface at any scale without extra cost by simply adjusting its texture coordinates. But in real case, without changing texture resolution we will lose performance and get aliased noisy image. This is because as we scale details down, we will have bigger distance in texture space between neighboring pixels of the final image. It causes worse cache usage, when sampling the texture, and aliasing artifacts appearing, as we omit more texture data.

This is a common problem with texture mapping and it's being solved by using a mip map [7] – a set of images, representing different resolutions of the texture from high to low. In 2D case we just sample corresponding mip image based on the scale of texture we apply. However, this is not so trivial when we represent some volumetric objects through a 3D texture. In lower resolution mips we should take into account such phenomena, as self shadowing, view dependent partial transparency and increasing roughness. In this paper we focus on the problem of partial transparency appearing on the edges of these volumetric objects in 3D mip images (further just mips). It appears because of blending between transparent and opaque texels during mips downscaling process. Opaque texels represent the detail objects, while transparent represent empty space.

So, our main contribution is a method to render, in general, semi-transparent details applied to a base surface. The approach takes its advantage from the case when semi-transparency originally comes from lower resolution mips of a texture representing some opaque object. With it a lot of shading calculations is being omitted by culling invisible parts of the details we simulate. It allows us to use mip-mapping for detail's textures, thus making implicit detail representation to be scalable without extra cost and without strong artifacts.

For applying a 3D texture on a base surface we use a real-time adaptation [8] of a method named shell mapping, originally introduced in [4]. Applying is performed by constructing a shell of tetrahedrons over the base surface and then, when rendering, tracing the texture inside each one using sphere tracing [9]. A disadvantage here is that the shell's tetrahedrons are constructed in preprocessing step on CPU. It prevents from using the method on arbitrarily animated surfaces. For example, some real-time soft body physics simulation can't be performed on the base surface. However, the construction process can be easily modified to be doable on-the-fly on GPU.

So, our second contribution is extension of current methods to be applicable to animated surfaces.

2. Related work

The methods for applying volumetric details onto a base surface can be roughly divided into two main categories by way of detail's representation: directly through geometry and implicitly through textures.

Methods from the first category, **geometry based methods**, allow us to produce accurate results, including non-repetitive structures over base mesh, directly through geometry.

Some methods construct the detail geometry in preprocessing step. Mesh quilting [1] uses synthesis of geometry over surface from given example patch. The method eliminates distortions, produced by texture mapping techniques. But all the generated geometry should then be stored in memory, which makes this method inappropriate for simulating big amount of small details.

Recent investigations made it possible to transfer style of example geometry onto a basic surface using neural networks [3]. All the generated geometry should then be stored in memory and be processed when rendering. So, it shares the same disadvantages as the previous method.

Deferred warping [2] allows you to attach detail geometry without need to store all of it in memory. While it's useful for attaching mid or high scale details, it's still suffers, as other geometric methods, from increasing computational cost when the amount of details becomes big, while their scale becomes small. The method focuses mostly on cloth rendering in garment prototyping applications. Also there is an approach for cloth rendering on fiber level [8], but it's applicable only for this specific cloth visualization.

The common disadvantage of all of these methods is the necessity to process all of the detail geometry, which makes them not scalable in terms of processing time: without special handling through levels of detail, it will increase if scale of the details will decrease. Also, only [2] directly supports free deformation of the base surface.

Methods from the second category, **image based methods**, allow us to easily scale and transform details by simply adjusting texture coordinates. But their rendering process is more complicated – we need to somehow trace objects inside the texture, which we apply onto the base surface. We will refer to such texture as input texture.

The simplest of the methods [11, 12] use height maps to represent relief on a surface. Extension of such methods was proposed in [13], which allow simulation of more complex surfaces. All of the methods require storing only 2D texture and doesn't require definition of shell space. However, they're limited by representing only relief details, without ability to represent random structures.

To represent such structures, shell mapping was proposed [4]. The main idea of the method is to construct a volumetric shell of tetrahedrons over the base surface and use each tetrahedron for bijective mapping between world and texture spaces. This way we can render each tetrahedron separately by tracing the input texture inside it, making the method appropriate for interactive and real-time rendering through rasterization of the tetrahedrons.

So, in [14] was proposed a method for rendering semi-transparent surface structure using shell mapping. The big disadvantage of the method is that the constructed tetrahedrons should be sorted every frame, which makes performance of the method to be highly dependent on the number of polygons in the base mesh.

In [8] was proposed a real-time rendering approach for shell mapping, which doesn't require tetrahedrons sorting. The brief description of the method is as follows. In preprocessing step we first make an offset surface by extruding the base surface towards vertex normals. Then corresponding triangles of the base and the offset mesh are connected, forming "prisms" (not in their mathematical sense). Each prism is then being split into 3 tetrahedrons, each forming an affine mapping between world and texture spaces. Now, by rasterizing all the tetrahedrons and tracing the texture inside each one, using affine mapping, we are getting volumetric details on the base surface. Tracing inside the texture space is done by using sphere tracing [9]. An SDF for it is stored in a separate texture. To discard invisible elements a depth buffer is used.

As previous methods use tetrahedral representation of the shell, which causes texture distortions due to affine mapping, smooth and curved shell mapping [5] was developed to eliminate them by the cost of rendering time.

All of the methods, however, don't take into account possible levels of detail for the input texture through mip mapping. Another general disadvantage of the methods is that shell construction is done in the preprocessing step, which prevents using arbitrary animations on the base surface.

In [6] was proposed a different approach for rendering shell space textures. Instead of tracing a texture inside prisms, it searches areas of intersection between planes facing towards camera and shell prisms, then render these areas from back to front. Such a method is capable of drawing semi-transparent details, thus allowing usage of levels of detail for volumetric textures. However, as we have to draw all of the areas without discarding invisible parts, there is significant overdraw, especially when the surface to be textured is big, as we have to cover all of the space by these planes.

3. Proposed methods

As was said in the introduction, we use real-time adaptation of shell mapping [8] as a method for applying details, represented through volumetric texture, to a base surface. Note, that there is no strict dependency on this method and the proposed algorithm for semi-transparent details rendering can be also used with smooth and curved shell mapping [5].

First, we describe our approach for rendering shells with levels of detail, where semi-transparent parts appear in downscaled mips. For this we proposed an algorithm for order independent transparency based on depth peeling [15]. The main point of the method is to discard invisible parts with hardware z-test, thus reducing rendering time each pass. This is archived by splitting each pass into multiple steps, which also minimizes amount of expensive lighting calculations.

After that we describe the proposed approach for shell geometry construction. By simplifying prism pattern defining process, we make the tetrahedrons or prisms to be constructible on GPU. To optimize memory and time consumption it's being done in two stages – first for vertex transformations and second for shell construction and calculation of intersections with tetrahedrons planes, which is needed for further ray tracing when performing rendering.

3.1. Shell rendering with levels of detail

Construction of level of detail for volumetric textures involves appearance of semi-transparent parts on the edges of the surface, encoded in a texture. To render them, we need to implement order independent transparency, as sorting of tetrahedrons is too expensive. Especially, first K semi-transparent layers need to be found to get a stable and good quality result. So techniques with linked list construction [16], with transmittance approximation [17] or stochastic transparency [18] aren't appropriate. An accepted method is depth peeling, introduced in [15]. It has different more advanced variants, such as [19, 20]. All of the approaches for depth peeling, though, have the following disadvantages.

- Culling of parts, which are on or in front of the last rendered layer, is performed within shader instead of with hardware depth test. It involves discard operations, which depend on texture reads. This, first, makes each pass more expensive. Second, all of the geometry should be re-rendered every pass, even parts, which are from already rendered layers.
- Lighting for a current layer is conditionally done in the same draw call with depth finding. So, a very high wave divergence appears when performing fragment shading stage, especially when we have discards in a shader (in our case we discard pixels for which there are no intersections with the implicit surface). It causes the hardware to perform both lighting and layer depth finding, instead of something one, most of the time.

These disadvantages make every depth peeling pass equally expensive due to bad hardware utilization. So the idea of our method is to utilize hardware depth test to cull as much pixels as possible. This way we can avoid a lot of unnecessary GPU work.

Basically, our algorithm is a modification of depth peeling technique, where we render transparent layers one by one until we reach the last layer or exceed maximum amount of layers. This maximum is a user defined parameter. Each rendering pass we render one new transparency layer and blend it with

previously rendered ones. The key difference is that we utilize hardware z-buffer for culling already rendered or invisible parts but not the slow software culling inside shaders.

Each pass is performed in 4 steps. First step is for finding new layer depth, as in depth peeling, but without shading. We move shading into a separate step to minimize amount of lighting calculations and to perform them only for visible pixels. Two more steps are intermediate steps, which are used to construct and update the depth buffer based on current layer's depth, found on the first step, and opaqueness info, updated on the shading step. We can write the maximum depth value into the depth buffer to cull these pixels in subsequent passes. This way a lot of pixels will be culled because every mip-level represents some opaque surface and has transparent parts only on the edges. So usually the first rendered layer will be already opaque for most of the pixels and all the pixels behind them from subsequent layers will be invisible.

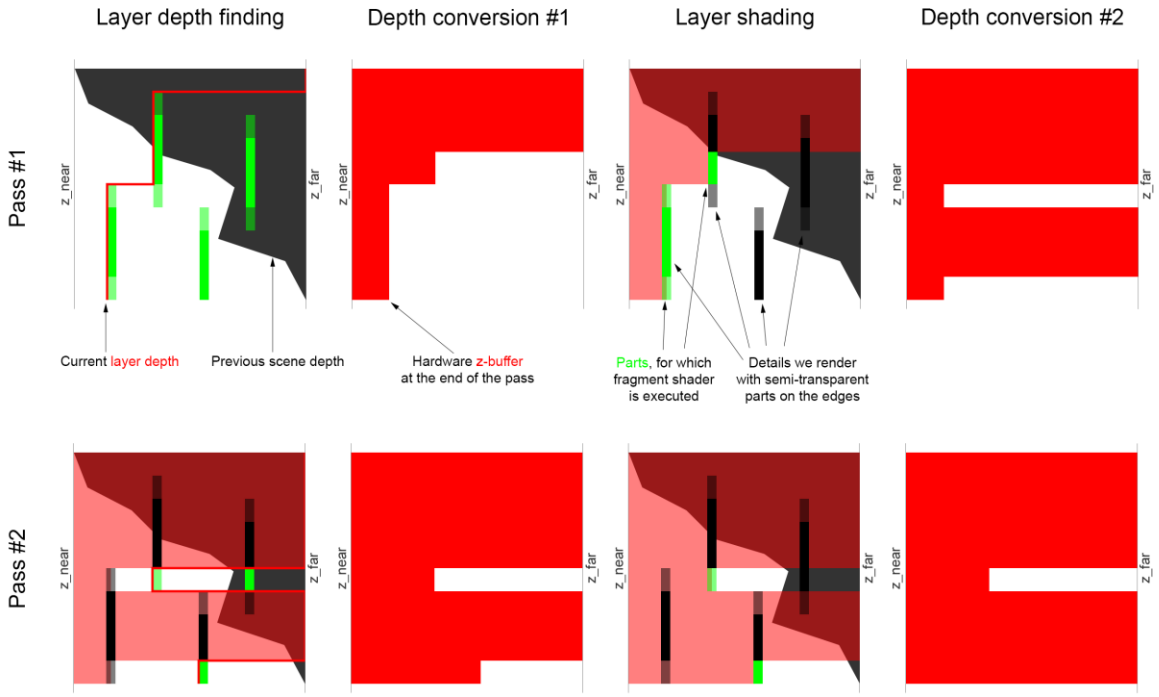

Figure 2: Schematic example of our OIT approach. Here we have 4 objects (thick vertical lines) with semi-transparent parts on edges and depth from previously rendered scene (grey triangle). Each pass 4 steps are performed: finding depth of a new layer, converting it into a z-buffer with regard to existing scene depth, rendering the layer, extending the z-buffer with regard to accumulated transparency. Left side of each image corresponds to the near plane and right to the far. Red color represents the z-buffer constructed or used for culling on each step. Green color denotes parts, for which fragment shader is being executed. Note that lighting operations are performed only for fragments, which impact on the final image

The detailed description of each step within a pass is following (schematic example of them is shown at Figure 2).

1. **Layer depth finding.** On this step the depth of a new layer is being searched. It's being done by setting previous layer depth as z-buffer with read-only mode. As in original depth peeling, the nearest depth is being written as output color with 'min' blending function. The problem here is that we have to search intersection with the encoded surface to determine whether the fragment should be discarded or not. Without it we'd have transparent parts occupying each layer and the algorithm would require way more passes to produce a correct image.
2. **Depth conversion #1.** This full-screen step is used for several purposes. First, it converts the found depth of a new layer into z-buffer by changing fragment depth in the shader. Note that we can't just copy a texture into hardware z-buffer and should do this trick because of hardware restrictions. Second, we compare depth of a new layer with the previous scene's depth: if the layer's

depth is greater than one from the previous scene, the layer's fragment should be culled and a value, corresponding to far culling plane (z_max), be written into z-buffer as all layers behind will also have greater depth.

3. **Layer shading.** On this step shading is performed for the new layer. The step is performed with z-function set to 'equal'. This allows to perform expensive lighting computations without overdraw. Also we can use here hardware alpha-blending with the previously rendered layers.
4. **Depth conversion #2.** This step is the same as #1, except that we additionally perform the opaqueness test as follows. The accumulated opaqueness is being checked if it's greater than some user-defined threshold (we use 0.95 here). If so, we write z_max into z-buffer as all subsequent layers we treat as invisible in this case and can cull them.

Although depth resolving pass isn't really cheap as it's performed in full-screen, it adds only constant cost. At the same time, every pass require less and less time, which compensates the cost of additional full-screen pass. Also, empirically results show, that 3-4 passes are enough in many cases. For example, you can see on Figure 3 that after the second pass there are almost no changes in the resulting image. Also you can see there, how rendering time is reduced with each subsequent pass.

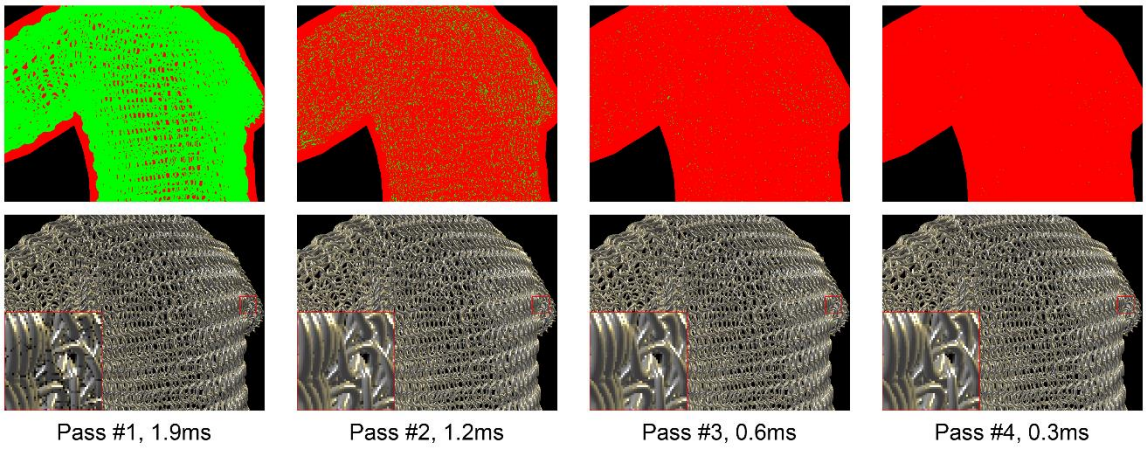

Figure 3: Example of rendering process with our method. On the top images green color highlights pixels, which pass depth test at the lighting step. On the bottom images corresponding results are shown

3.2. Construction of shell geometry on-the-fly

In [4, 5, 8, 14] shell construction is performed during preprocess step. The consistency between prisms is archived by maintaining correct edge patterns between adjacent prisms. It's done by randomly assigning them first and then correcting inconsistencies. However this makes the whole process to be data dependent and thus non-parallelizable.

But there is a way simpler approach. As we have indexes of every vertex on a base mesh, they can be used to determine directions of edges, which split sides of the prisms. An edge is rising, if it goes from vertex of the base mesh with lower index to a vertex with higher index, and falling otherwise. This method doesn't require any work to assign patterns to the prisms and guarantee their consistency between each other. Also it's data independent, so the tetrahedrons can be constructed in parallel for each face of the base mesh. And thus, allowing implementation on GPU in compute shaders as follows.

To reduce computing time and memory usage, the process is split into two steps. On the first step the base mesh S is transformed into S_t as it would be in vertex shader, then an additional offset surface S'_t is constructed by shifting vertices of S_t toward vertex normals at some predefined distance. All of the data produced is stored into a buffer for vertex data. On the second step shell tetrahedrons with all necessary data are constructed as in [8] with determining edge directions as described above. The stage is performed per face. The results are stored into separate buffer for shell data. To connect these two

buffers for access from vertex shader, a third buffer, which stores indexes of vertex data for each tetrahedron, is being constructed. Scheme of the method is depictured below in Figure **4**.

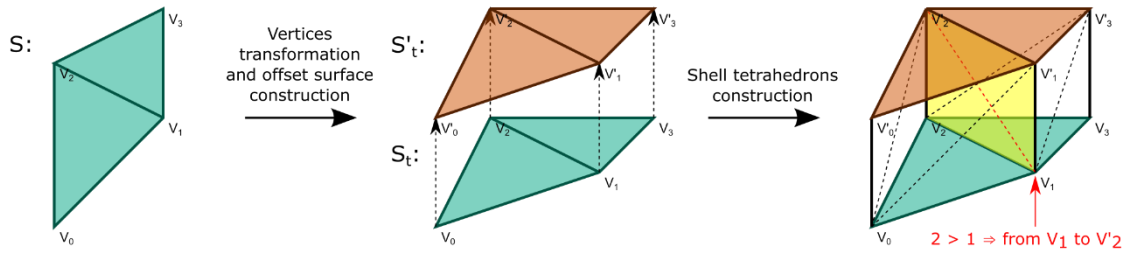

Figure 4: Schematic example of shell construction in two steps. On the first step the base mesh S is transformed into S_t, after what offset surface S'_t is constructed by shifting vertices of S_t towards their normals. On the second step shell tetrahedrons (or only prisms) are constructed. For preserving consistency between prism sides, direction of splitting edge is determined based on base vertex indexes

4. Results

The proposed method was tested on multiple scenes against shell mapping without texture mips, both using tetrahedral representation of the shell. Additionally on a single scene it was tested against simple instancing without using levels of detail for instanced geometry. Testing was done on a PC with NVidia GeForce RTX 2080 GPU.

Averaged performance results are shown in Figure 5. As we can see, the basic method is faster on smaller sizes, as cache is better utilized and we need to perform rendering only once. But with texture size increasing its performance significantly reduces, as cache hit rate becomes lower. At the same time the proposed method almost saves it's speed and outperforms the basic one on bigger sizes.

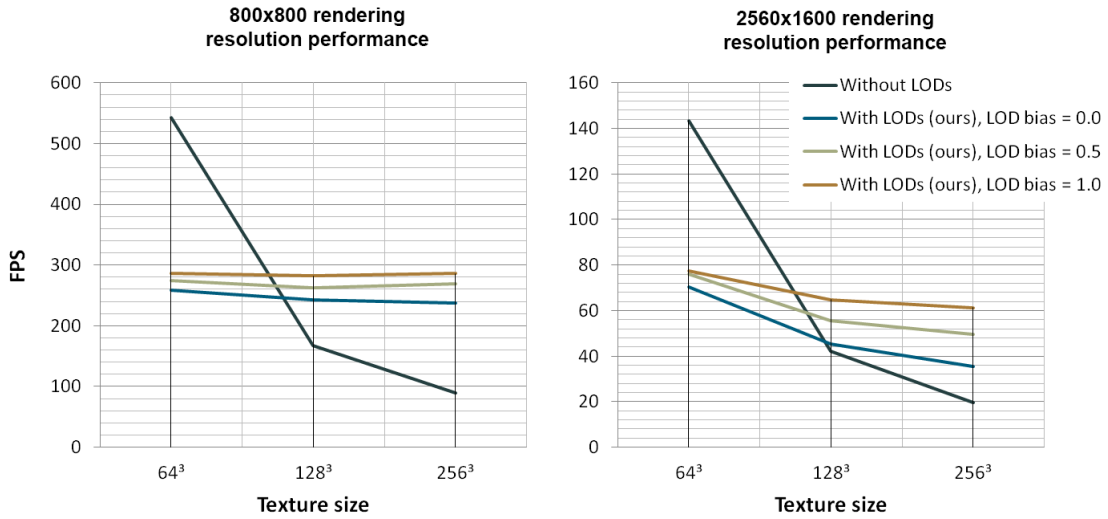

Figure 5: Performance comparison between our method with texture LODs, and basic shell mapping without them. We put here average FPS results between 3 scenes, shown on Figure 6, with different rendering resolutions: 800x800 (left) and 2560x1600 (right)

Visual appearance is shown on Figure 6. We can see the main advantage of the method: using levels of detail gives anti-aliased results almost comparable to super-sampled ones, while having good performance. Although some color altering can be noticed. First, there is darkening on parts, where surface is nearly perpendicular to view projection plane – in that case a lot of prisms meets in the view direction. It appears because the OIT approach doesn't handle all the layers. For the screenshots we

used 4 layers. Second, some color altering appears because we use simple linear downscaling for mips construction, which, as discussed before, isn't correct. Especially in our case, transparency should be constructed anisotropically. For example, when the texture represents some plane, it should be opaque when the view direction is perpendicular to that plane, and almost transparent, when it's parallel. There is some research about correct handling of downscaled volumetric textures [21, 22], but it focuses on offline rendering. So adopting it to real-time is a topic for future work.

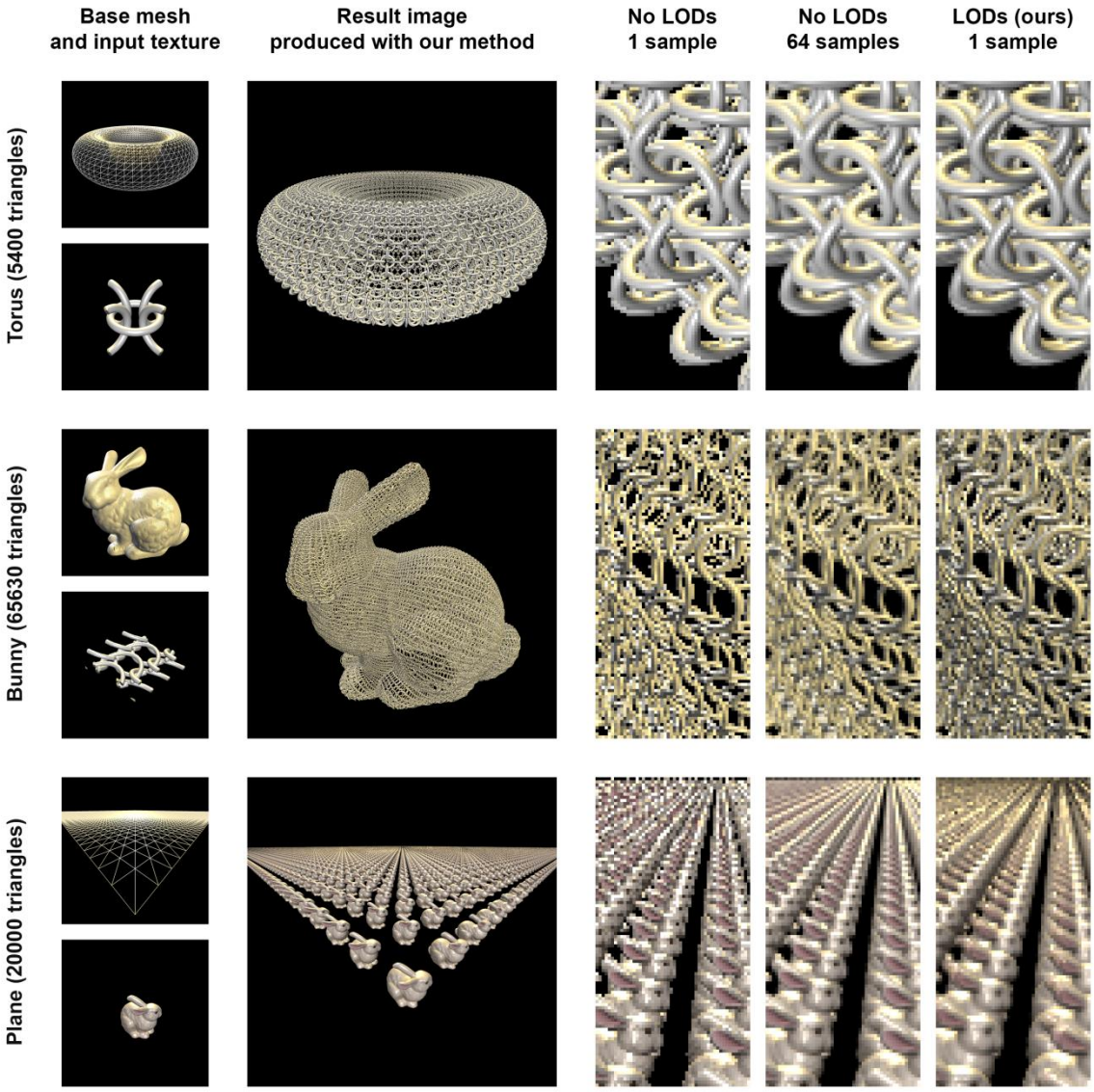

Figure 6: Visual comparison between our method, described in section 3, which uses texture LODs, and basic shell mapping without them on different scenes. To see the difference, we took small parts of the resulting images. For basic approach we produced pictures with 1 and 64 samples per pixel. The second gives true anti-aliasing at cost of 64x slower rendering time and is used as ground truth here

5. Conclusion

We described a method for rendering small-scale features on polygonal meshes, which is archived by constructing levels of detail for volumetric textures and handling their partial transparency. For latter a method for order independent transparency, based on depth peeling, was proposed. It utilizes the fact, that we render opaque details, and, with use of hardware depth test for culling, allows to reduce amount of fragment shader invocations each subsequent pass. We used very basic lighting calculation here, so

performance gain can be more when some advanced methods will be used, as there is no overdraw when performing it. To make the technique applicable for animated surfaces, we proposed a way to construct the shell on-the-fly on GPU. This way it can be easily embedded and used with existing rendering pipelines.

Some topics are left untouched though. First, simple mip-map construction with linear downscaling doesn't produce correct results in case of 3D textures, especially on very low-scale mips. Second, we still have to rasterize all of the tetrahedrons each pass. Even though we perform culling with z-test, the rasterization and vertex processing work should be done before that. With modern hardware, which introduced hardware ray tracing, it's it possibly will be more efficient to trace shell geometry instead of rasterizing it. So, both of the problems are the topics for the future work.

6. References

[1] K. Zhou, X. Huang, X. Wang, Y. Tong, M. Desbrun, B. Guo, H. Y. Shum, Mesh quilting for geometric texture synthesis, in: ACM SIGGRAPH 2006 Papers, SIGGRAPH '06, 2006, 690–697. doi:10.1145/1179352.1141942.

[2] M. Knuth, J. Bender, M. Goesele, A. Kuijper, Deferred Warping, IEEE Computer Graphics and Applications, 37 (6) (2017) 76–87. doi:10.1109/MCG.2016.41.

[3] A. Hertz, R. Hanocka, R. Giryes, D. Cohen-Or, Deep geometric texture synthesis, ACM Transactions on Graphics, 39 (4) (2020) 1–11. doi:10.1145/3386569.3392471

[4] S. D. Porumbescu, B. Budge, L. Feng, K. I. Joy, Shell maps, ACM Transactions on Graphics, 24 (3) (2005) 626–633. doi:10.1145/1073204.1073239.

[5] S. Jeschke, S. Mantler, M. Wimmer, J. Kautz, S. Pattanaik, Interactive smooth and curved shell mapping, in: Proceedings of the 18th Eurographics conference on Rendering Techniques (EGSR'07), Eurographics Association, Goslar, DEU, 2007, 351–360.

[6] P. Decaudin, F. Neyret, Volumetric billboards, in: Computer Graphics Forum, 28(8) (2009) 2079–2089. doi:10.1111/j.1467-8659.2009.01354.x.

[7] L. Williams, Pyramidal parametrics, in: Proceedings of the 10th annual conference on Computer graphics and interactive techniques (SIGGRAPH '83), Association for Computing Machinery, New York, NY, USA, 1983, 1–11. doi: 10.1145/800059.801126.

[8] N. Ritsche, Real-time shell space rendering of volumetric geometry, in: Proceedings - GRAPHITE 2006: 4th International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia, 1(212) (2006) 265–274. doi:10.1145/1174429.1174477.

[9] J. C. Hart, Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces, The Visual Computer, 12(10) (1996) 527-545.

[10] K. Wu, C. Yuksel, Real-time fiber-level cloth rendering, in: Proceedings - I3D 2017: 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, 2017, doi:10.1145/3023368.3023372.

[11] M. McGuire, M. McGuire, Steep Parallax Mapping, I3D 2005 Poster, 2005. URL: http://www.cs.brown.edu/research/graphics/games/SteepParallax/index.html.

[12] N. Tatarchuk, Dynamic parallax occlusion mapping with approximate soft shadows, in: Proceedings of the Symposium on Interactive 3D Graphics, March, 2006, 63–70. doi:10.1145/1111411.1111423.

[13] X. Wang, X. Tong, S. Lin, S. Hu, B. Guo, H.-Y. Shum, Generalized displacement maps, in: Eurographics Symposium on Rendering, January, 2004, 227–233. doi:10.2312/EGWR/EGSR04/227-233.

[14] J.-F. Dufort, L. Leblanc, P. Poulin, Interactive Rendering of Meso-Structure Surface Details using Semi-Transparent 3D Textures, in: Proceedings of Vision, Modeling, and Visualization, 2005, 399–406.

[15] C. Everitt, Interactive order-independent transparency, 2001.

[16] P. Barta, B. Kovács, Order Independent Transparency with Per-Pixel Linked Lists, The 15th Central European Seminar on Computer Graphics, 2011. doi:10.1.1.309.4525.

[17] M. McGuire, L. Bavoil, Weighted Blended Order-Independent Transparency, Journal of Computer Graphics Techniques, 2(2), (2013) 122–141. URL: http://jcgt.org/published/0002/02/09/.

[18] E. Enderton, E. Sintorn, P. Shirley, D. Luebke, Stochastic transparency, in: IEEE transactions on visualization and computer graphics, 17(8) (2010) 1036-1047. doi: 10.1145/1730804.1730830.
[19] L. Bavoil, K. Myers, Order Independent Transparency with Dual Depth Peeling, Image Rochester NY, 107 (2008) 22020–22025.
[20] F. Liu, M. C. Huang, X. H. Liu, E. H. Wu, Efficient depth peeling via bucket sort, in: Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware, 2009, 51–57.
[21] S. Zhao, L. Wu, F. Durand, R. Ramamoorthi, Downsampling scattering parameters for rendering anisotropic media, ACM Transactions on Graphics, 35(6) (2016) 1–11. doi:10.1145/2980179.2980228.
[22] G. Loubet, F. Neyret, A new microflake model with microscopic self-shadowing for accurate volume downsampling, in: Computer Graphics Forum, 37(2) (2018) 111–121. doi:10.1111/cgf.13346.