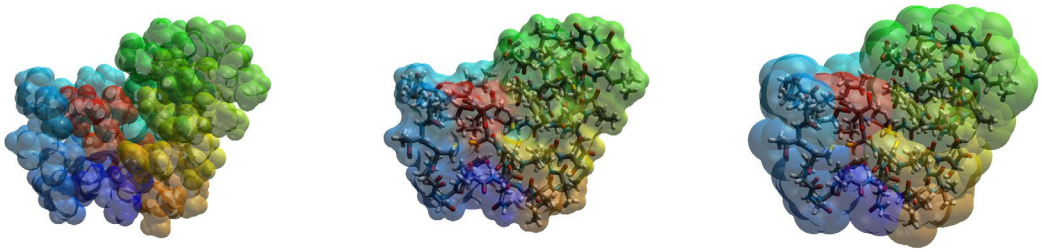
Eurographics Symposium on Parallel Graphics and Visualization (2013)
F. Marton and K. Moreland (Editors)

Rendering Molecular Surfaces using Order-Independent Transparency

D. Kauker, M. Krone, A. Panagiotidis, G. Reina, and T. Ertl

Visualization Research Center, University of Stuttgart, Germany



(a) Van der Waals (b) Solvent Excluded Surface (c) Solvent Accessible Surface

Figure 1: *Visualizations of different molecular surfaces of a small protein rendered using our order-independent transparency algorithm. Note that the interior Stick model is correctly combined with the transparent surfaces.*

Abstract

In this paper we present a technique for interactively rendering transparent molecular surfaces. We use Puxels*, our implementation of per-pixel linked lists for order-independent transparency rendering. Furthermore, we evaluate the usage of per-pixel arrays as an alternative for this rendering technique. We describe our real-time rendering technique for transparent depiction of complex molecular surfaces like the Solvent Excluded Surface which is based on constructive solid geometry. Additionally, we explain further graphical operations and extensions possible with the Puxels approach. The evaluation benchmarks the performance of the presented methods and compares it to other methods.*

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Boundary Representations I.3.3 [Computer Graphics]: Picture/Image Generation—Display Algorithms J.3 [Computer Applications]: Life and Medical Sciences—Biology and Genetics

1. Introduction

Many fields of research like chemistry, physics, and biomedicine work with molecular data. This data can originate from purely computational sources (e.g. molecular dynamics simulations, normal mode analysis, or fitting calculations) or from measurements (e.g. x-ray crystallography or scanning tunneling microscopy). High-quality visualizations are an important tool for the analysis of these data sets. Many different molecular models have been developed to that end. Among these models, molecular surface representations are widely used. They are beneficial for many applications since they show the boundary of a molecule with respect to a certain property. Naturally, transparent surfaces do not fully obscure other parts of the scene. That is, users can see through objects and spot features or changes on the backside immediately without changing the perspective. Furthermore, users are able to see inside molecules. This makes transparent molecular surfaces especially suited for complex analysis tasks where the user wants to see the molecular surface but also another, sparse representations of the molecule like a stick model (cf. Figure 1). Multiple nested representations of the molecule can be used to provide more information to the user.

DOI: 10.2312/EGPGV/EGPGV13/033-040

Traditionally, geometric objects like molecular su faces are triangulated for rendering. Rendering m tiple complex objects as transparent triangle mesh can be time-consuming, though. The standard rend ing pipeline uses blending to accumulate the color p fragment. However, the triangles of all objects ha to be sorted according to their depth for a corre result. Since the advent of programmable GPUs, m tiple algorithms for order-independent transparen (OIT) have been presented (see Section 2.2) that not require explicit sorting before rendering.

Molecular surfaces are defined as a set of impli surface patches. Thus, they can be rendered usi GPU-based ray casting [Gum03,RE05]. This results higher visual quality and faster rendering while ad tionally saving the computation time and memory for the triangulation. However, for transparent rendering, it poses an even greater problem than triangulated surfaces. The implicit surface patches might not only overlap and generate multiple fragments which have to be sorted according to their depth, but there are also parts of these surface patches which have to be removed to obtain a correct final image.

In this paper we present a rendering method for transparent molecular surfaces. The main contributions of our paper can be summarized as follows:

- We describe the *Puxel* method for order-independent transparency (OIT), which uses per-pixel linked lists and per-pixel arrays.
- We explain how the Puxel algorithm is capable of Constructive Solid Geometry (CSG) modeling.
- We detail how this method can be used to render transparent molecular surfaces in real-time.
- We compare Puxels to a freely available existing OIT approach from the NVIDIA SDK.

The remainder of this paper is structured as follows: Section 2 explains molecular surface rendering and summarizes related work. In Section 3 we describe the contribution of our work. We explain our rendering technique for molecular surfaces based on Puxels in Section 4. Benchmarking results and comparisons to other approaches are presented in Section 5. Section 6 discusses extensions and further applications of the Puxels framework. Finally, a summary and directions for future work are given in Section 7.

2. Related Work

In this section, we describe previous work in the field of molecule rendering and order-independent transparency rendering.

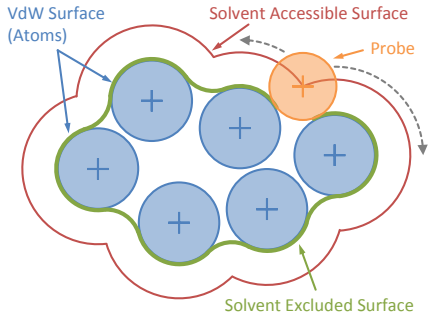


Figure 2: *Schematics of the Solvent Accessible Surface (red) and the Solvent Excluded Surface (green), both defined by a probe (depicted orange in a sample position) rolling over the molecule atoms (blue).*

2.1. Molecular Surface Definitions

There are several definitions for molecular surface models, which are based on physical properties of the atoms.

The *Van der Waals* (VdW) surface shows the extent of a molecule. All atoms are represented by spheres with the corresponding VdW radius of their element, which is derived from the distance between non-bonded atoms. Figure 2 shows a 2D schematic of the VdW surface (colored blue).

A fundamental drawback of the VdW surface is that it has no correlation with the surrounding medium (e.g. solvent molecules or ligands). The *Solvent Accessible Surface* (SAS) [Ric77] includes this information. It is defined by the center of a spherical probe which rolls over the VdW surface of the molecule. Thus, the SAS represents the surface that is directly accessible for solvent molecules the size of the probe. Figure 2 shows a 2D schematic of the SAS (colored red). The SAS closes small gaps and holes in the VdW surface, which cannot be entered by solvent molecules.

The *Solvent Excluded Surface* (SES) [Ric77] is the topological boundary of the union of all possible probe spheres that do not intersect any atom of the molecule [GB78]. The SES can also be defined by a spherical probe rolling over the VdW surface. Here, the surface of the probe traces out the SES (in contrast to the SAS, where the probe center is used). Figure 2 shows a 2D schematic of the SES (colored green). Consequently, the SES consists of three types of geometrical primitives: convex spherical patches, concave spherical triangles, and saddle-shaped toroidal patches. The spherical patches are the remainders of the VdW spheres. They occur when the probe is rolling over the surface of an atom and touches no other atom. The toroidal patches are formed when the probe is in contact with two atoms and rotates around the

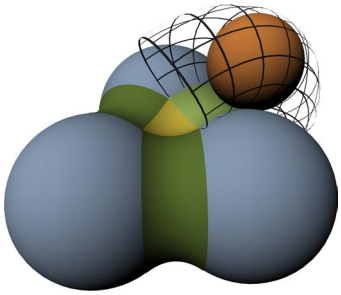


Figure 3: *3D schematics of the SES depicting the spheres (blue), the spherical triangle (yellow) and the toroidal patches (green) defined by the probe (orange).*

axis connecting the atom centers (see Figure 3). The inward-looking, saddle-shaped patch of the resulting torus is part of the SES. While rolling, the probe traces out a small circle arc on each of the two VdW spheres. These small circle arcs are part of the boundaries of the aforementioned spherical patches. Spherical triangles occur if the probe is simultaneously in contact with three VdW spheres. Here, the probe is in a fixed position, meaning that it cannot roll without losing contact to at least one of the atoms. Please note that the probe can in theory be in contact with more than three atoms, however, these cases can be divided into multiple atom triplets for simplicity.

Similar to the SAS, the SES closes small gaps and holes, but it does not inflate the molecule. The extent of the VdW surface is retained. Therefore, the SES is the most versatile molecular surface and suitable for many applications and analysis tasks like docking or solvation. Other molecular surface definitions like the *Molecular Skin Surface* (MSS) [Ede99] or *Metaballs* [Bli82] are less commonly used.

2.2. Molecular Surface Visualization

The computation and rendering of the VdW surface and SAS is quite trivial, since they only consist of spheres. If the surface is drawn opaque, the interior parts of overlapping spheres do not have to be removed since they are not visible anyway. However, for transparent renderings, the interior parts have to be removed. If the spheres are composed of triangles, the intersections for each triangle have to be computed. Laug et al. [LB02] presented methods for high-quality meshing of molecular surfaces. Today, implicit surfaces like spheres are commonly rendered using high-quality GPU-based ray casting [Gum03, RE05]. Clipping the interior parts would require neighborhood information. Since this would result in a plethora of additional intersection tests in some cases, available tools often do not clip the interior parts, which results in a cluttered representation when using transparency.

The computation of the SES is more complex and costly. Connolly [Con83] presented the analytical equations to compute the SES. Based on his work, several methods to accelerate the computation of the SES have been introduced. Edelsbrunner and Mücke [EM94] presented the α-*shape*. Sanner [SOS96] developed the *Reduced Surface*. Krone et al. [KBE09] later used this algorithm for dynamic simulation data and were the first to render the SES using fast GPU-based ray casting of the individual patches instead of triangulations. Varshney et al. [VBW94] described a parallelizable algorithm based on Voronoi diagrams. Lindow et al. [LBPH10] showed that this method can also be used to compute the MSS. Totrov and Abagyan [TA95] proposed a contour-buildup algorithm for the SAS, from which the SES can be derived. This algorithm was recently shown to be parallelizable on multi-core CPUs [LBPH10] and GPUs [KGE11]. Parulek and Viola [PV12] presented an implicit representation for the SES using a ray casting approach. With the exception of [PV12], all aforementioned methods compute the patches of the SES. The technique best suited for the application may depend on different factors, e.g. hardware capabilities.

2.3. Transparency Rendering

Molecular models can be rendered using ray tracing by sending primary rays from the view point into the scene. For every hit with an object, secondary rays are generated to capture reflection and refraction. Ray tracing generates very accurate images, but is computationally demanding. With the performance of current CPUs and GPUs, real-time ray tracing of complex scenes has become possible [Wal04]. This, however, requires elaborate acceleration structures. BALLView [MHLK06], a molecule viewer, offers such real-time ray tracing [MDG*10]. Although ray tracing—including secondary rays—achieves interactive frame rates, point-based GPU ray casting [RE05] has a considerably higher performance. The performance of ray tracing further decreases when using transparency. As we do not include effects which require secondary rays, we use GPU ray casting.

The Painter's algorithm [FvDFH90] is the most straightforward way for drawing scenes that contain transparent faces: draw the polygons ordered by their depth from farthest to closest. Hereby, the correct rendering of the transparent fragments is given implicitly. Chen et al. [CSN*12] use view point dependent presorted meshes and render using the painter's algorithm. Zhang et al. [ZP07] do single-pass deferred blending using point-based rendering, also achieving transparency effects.

Depth Peeling [Eve01] is a multi-pass algorithm that captures fragments using multiple depth tests. Fragments that have been captured in earlier render passes are rejected, thereby avoiding explicit sorting. There are several optimizations to the basic concept, for example peeling multiple layers per pass [BM08, LHLW09], offering z-fighting awareness [VF11], or using only constant memory [BE11]. Storing multiple fragments and their attributes is a concept first described as *A-buffer* [Car84] as part of the REYES renderer for the CPU, and has since been extended to GPUs as *R-buffer* [Wit01] and stencil-routed *k-buffer* [BCL*07]. The k-buffer uses textures as multiple render targets to store k fragments per render pass, avoiding read-modify-write hazards while enabling various effects like order-independent transparency, depth-of-field, and volume rendering. Several limitations, like artifacts and approximations, are addressed by Yang et al. [YHGT10] with per-pixel linked lists for modern GPUs. In this paper, we use an implementation similar to their method.

3. Rendering Methods

In the following, we explain our Puxel algorithm used for rendering transparent molecular surfaces and detail possible implementations.

3.1. Puxels Algorithm

The Puxels (pixel tubes) framework is similar to the A-buffer. It collects and stores all fragments for later processing. This is achieved in a single render pass using per-pixel linked lists [YHGT10] or in two render passes using per-pixel arrays, similar to a k-buffer with varying k for each pixel. The final image is created by sorting—either globally all fragments or locally for each per-pixel list—and then compositing each pixels' fragments front to back. After the data has been sorted, it is also possible to perform operations like CSG on the fragments (see Section 3.2).

The whole Puxel algorithm consist of four rendering stages: *clear*, *render*, *order*, and *display*. First, the *clear* shader resets all data structures. Then, the *render* shader is used to store the fragments as mentioned above. This results in unordered lists or arrays for each pixel, which need to be sorted according to fragment depth for blending. Sorting can be done either in a separate step or directly before blending. We use a combination of both: for long arrays or lists, the data is sorted in global memory using an optional *order* shader. Short arrays and lists are cached in a local array in the *display* shader and sorted directly prior to blending. Since everything is re-generated every frame, the cost of sorting the fragments can be assigned to any of the stages. This shader also composes

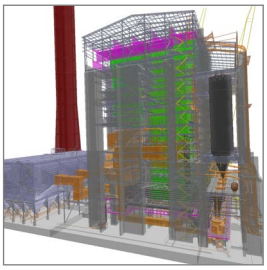


Figure 4: *Close-up view of the UNC power plant model rendered with Puxels.*

the depth-sorted data front to back and sends them to the OpenGL framebuffer. Figure 4 shows the UNC power plant model rendered with our Puxel method.

Data is collected and processed using OpenGL shaders and `GL_SHADER_STORAGE_BUFFER`s introduced in OpenGL 4.3. For each pixel of the viewport, the header buffer contains the entry index to the data buffer which stores the fragment attributes.

For the per-pixel array, the number of fragments per array is counted in the first render pass, so the entry index can be calculated using a prefix sum. In a second render pass, each fragment is stored in the respective array within the data buffer.

An element of the data buffer contains the color and depth per fragment and may store additional attributes (e.g. fragment normal or texture coordinate). Per-pixel lists are created during rendering by appending the attributes of a fragment to the list in the fragment shader. A global atomic counter which contains the index of the next free element in the data buffer is incremented in a thread-safe manner (`atomicCounterIncrement`). The arrays are created in the same way by using an atomic per-pixel counter that contains the index of the next free element in the per-pixel array. Existing fragment shaders and rendering methods can easily use Puxels by replacing assignments like `gl_FragColor = color;` and `gl_FragDepth = depth` with `puxelsStore(color, depth)`.

When a scene additionally contains opaque fragments, such as the sticks in Figure 1(b) and Figure 1(c), they are rendered into a separate framebuffer object first. Transparent fragments that lie behind opaque ones can be discarded during rendering, shortening the resulting lists.

3.2. Constructive Solid Geometry

Using the Puxels approach, all fragments in the scene are available for compositing. Thus, the constructive solid geometry (CSG) operation *union* can be implemented as follows: when composing the sorted frag-

ments, we count the layers and only render a fragment when the counter is incremented from 0 to 1 (entering the object) or decremented from 1 to 0 (leaving the object). This represents the outer border and no internal structure or overlapping geometry is composed into the final image. The counter increases for fragments facing the viewer and decreases for fragments facing away. We determine the orientation of a fragment in the *render* stage and store it as sign of the depth value. Therefore, all depth-dependent checks are performed on the absolute depth value for consistency.

The two other basic CSG operations—difference and intersection—can be implemented analogously. More complex CSG operations, e.g. *XOR*, are logical combinations of these three basic operations.

4. Transparent Molecular Surface Rendering

The VdW and the SAS representations consist only of spheres. As these spheres intersect each other, all parts of a sphere which are inside another sphere have to be removed since they would be visible when using transparency. The Puxel method allows to remove these interior parts using the CSG *union* operation explained in Section 3.2. That is, all sphere fragments are stored in the Puxel data buffer during rendering. Interior fragments are discarded in the composing stage.

As explained in Section 2.1, the SES consists of three types of patches: concave spherical triangles, torus segments, and convex spherical patches. The spherical triangles can be ray casted straightforwardly, since they have no interior parts which have to be clipped. Krone et al. [KBE09] showed that the outer part of the torus, shown as wireframe in Figure 3, can be removed using a sphere-intersection test. The interior parts of the torus (shown as dotted lines in Figure 5) can be removed by intersection tests with two planes. Simply speaking, we cut a pie slice out of the torus ring. The spherical patches are the parts of the VdW spheres that are accessible by the rolling probe. The parts that are interior to the small circles traced out by the rolling probe while forming a torus have to be removed. These parts can be cut away using a clip plane through the small circle. However, there might be an inner remain which also has to be removed (shown as purple patch in Figure 5). If we use the Reduced Surface (RS) for the SES computation, these problematic remains lie inside the RS. The RS is a closed triangle surface. The RS triangle vertices connect atom centers that share a torus patch. The RS triangle faces span atoms which share a spherical triangle. Please refer to [SOS96, KBE09] for more details. That is, we can use Puxel CSG to remove the spherical remains. Please note that our method could also use α-shapes [EM94] or Power Cells [VBW94] to compute the SES and remove the inner remains of the spheres since they are essentially dual to the RS.

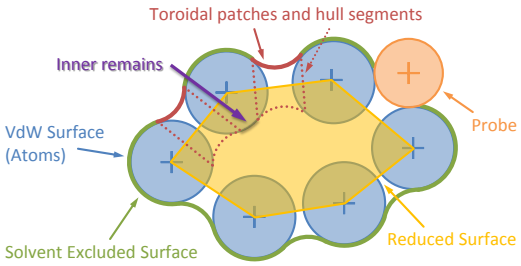


Figure 5: *Schematic representation of the SES and Reduced Surface of a molecule depicting the inner remains of a VdW sphere which has to be removed.*

For the VdW and SAS, we only performed CSG on visible objects. However, we can also render invisible objects (alpha equals zero) which will not contribute to the final image, but can be considered for the CSG operations. Hence, we can render the triangles of the RS with alpha set to zero. Consequently, the fragments of the atom spheres inside the RS will be removed in the Puxel compositing stage without changing the algorithm.

However, not all parts of the sphere which are located within the inner torus ring are also inside the RS. These fragments have to be removed as well. For that purpose, we close the torus. That is, we draw the two planes delimiting the torus patch and the interior parts of the torus ring (red dotted lines in Figure 5) as invisible objects as well. Therefore, the CSG operation additionally will remove all fragments inside the torus object. The final rendering will only show the outer parts of the VdW spheres, the spherical triangles, and the outer torus patches. Thus, the transparent SES can be rendered correctly.

5. Evaluation

As mentioned in Section 2.2, Parulek et al. [PV12] defined an implicit function for the SES. They render the SES using a sphere tracing of this function. Their performance evaluation for opaque surfaces shows similar frame rates on a NVIDIA Geforce GTX 480 as we achieve on a GTX 680 for transparent renderings. Since the sphere tracing is only executed until the first hit is found, more steps would be necessary for transparency rendering. This would drastically lower the performance of their method for large numbers of surface layers. Additionally, their visualization approach might have visible artifacts if the number of neighboring atoms is set too low [PV12]. While this is only a minor distraction in opaque surfaces, the artifacts might become prominent in transparent renderings.

Table 1: *Comparison of Puxels with other approaches for different polygon models using an* NVIDIA *GTX 680 on a* 1024×1024 *viewport. The numbers for the rendering methods are frames per second.*

	Dragon	Protein	UNC
Vertices	435 k	1,046 k	10,975 k
Faces	871 k	348 k	12,748 k
Fragments	1,157 k	2,227 k	14,765 k
Depth Layers	10	16	186
Screen Coverage	48.4 %	70.2 %	76.1 %
OpenGL	599.5	693.0	110.4
Dual Depth Peeling	89.8	62.9	0.8
Puxel Lists	135.2	97.0	3.9
Puxel Arrays	102.4	82.8	4.2

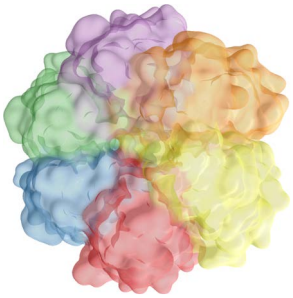
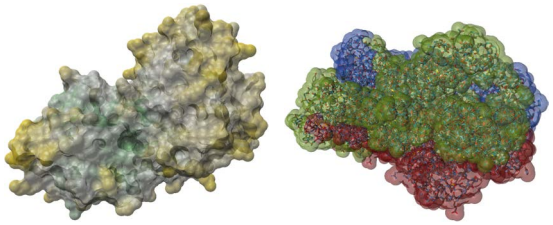


Figure 6: *Polygon model of the 1GKI protein surface rendered using Puxels. The mesh was generated using QuickSurf [KSES12] and exported from the freely available Visual Molecular Dynamics tool [HDS96].*

We compare the performance of the Puxels method with dual depth peeling (DDP) [BM08] and, for reference, with pure (unsorted) OpenGL blending. DDP was chosen since it guarantees full visual quality whereas other accelerated depth peeling methods, e.g. bucket depth peeling [LHLW09], may introduce visual artifacts. We use the freely available implementation of DDP from the NVIDIA SDK. For benchmarking, we used an Intel i7 920 CPU with an NVIDIA Geforce GTX 580 (Fermi) and an NVIDIA Geforce GTX 680 (Kepler), driver version 310.90. We used freely available meshes and protein data sets from the Protein Data Bank [BWF*00] for our tests.

Table 1 shows the performance in frames per second of the Puxels per-pixel linked lists and per-pixel arrays compared to the dual depth peeling method for polygon meshes: a single Stanford Dragon, a protein surface mesh generated by VMD [HDS96] (see Figure 6) and the UNC power plant (see Figure 4).

As DDP can only peel two depth layers per rendering pass, the framerate is very low compared to our implementations and to the OpenGL reference due to the number of render passes.



(a) 1VIS, SES colored by temperature factor (b) 4ADJ, SAS colored by chain with Sticks inside

Figure 7: *Two proteins (1VIS and 4ADJ) used for the benchmarks shown in Table 2.*

We assume that the difference between the list and the array approach is due to the different data structure and the additional operations necessary. Per-pixel arrays require two render passes and the calculation of the prefix sum. The benefit is a more structured memory layout as each pixels' fragments are stored consecutively. By contrast, the list approach has a scattered memory layout.

Table 2 details the performance for the ray casted molecules (see Figures 1 and 7). The surface representation directly influences the fragment count, especially due to inner fragments and hull fragments needed for CSG. The SES generates about twice as many fragments as the VdW representation. The fragment count of the SAS is three to four times the count of VdW. This is also the approximate loss in the frame rate for the surface representations. A higher number of fragments slows down the performance when sorting and blending. The same effect as seen in Table 1 occurs here, too. Per-pixel arrays outperform per-pixel linked lists only for very high numbers of fragments. For small and medium fragment counts, the per-pixel linked lists are preferable performance-wise.

Another interesting fact is the performance difference between the GTX 680 and GTX 580. On average for the selected benchmarks, the GTX 680 based on the newer Kepler architecture is about 40% slower than the older Fermi architecture of the GTX 580. This is not a Puxels related effect but, according to our experience, holds for other applications as well.

6. Extended Applications

The main advantage of the Puxels technique is that all fragments of the scene are available and can be used for later operations. In the following, we detail extensions for Puxels.

6.1. Splatting

Instead of rendering a full screen quad, the Puxel data can be displayed directly using splatting [Wes90]. For

Table 2: *Performance evaluation for different molecules and surface models rendered with our Puxel framework. The numbers for the rendering methods are frames per second.*

Surface		Van der Waals			Solvent Excluded Surface			Solvent Accessible Surface		
Molecule		1YV8	1VIS	4ADJ	1YV8	1VIS	4ADJ	1YV8	1VIS	4ADJ
Atoms		641	2482	9720	641	2482	9720	641	2482	9720
Fragments		2.30 M	6.98 M	9.34 M	4.64 M	12.33 M	18.00 M	8.87 M	23.92 M	32.16 M
Screen Coverage		18.0 %	48.4 %	42.6 %	18.0 %	48.7 %	42.8 %	23.8 %	57.1 %	47.1 %
Depth Layers		44	66	92	117	135	188	128	160	234
GTX 580	Puxel Array	69.5	23.8	14.7	21.3	10.2	3.8	14.0	4.7	2.5
	Puxel Lists	85.4	29.6	16.6	31.0	11.2	6.2	14.7	4.8	1.8
GTX 680	Puxel Array	58.2	20.9	12.9	16.0	6.5	2.1	8.0	2.6	1.4
	Puxel Lists	76.6	26.1	13.9	22.1	7.7	4.2	10.3	3.3	1.3

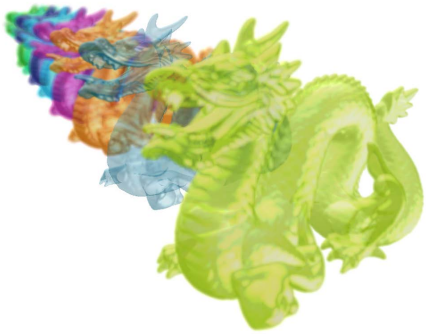


Figure 8: *Multiple Stanford dragons with depth-of-field effect rendered using our extended Puxel method.*

this purpose, each element needs to store the full 3D position and color. Because of the flexible extensibility of the Puxels data structure, those attributes can simply be added to the elements of the data buffer. After rendering the scene into the Puxel buffers, the data is mapped to CUDA and sorted using the Thrust library. This destroys the structure of our buffer, making the data effectively an array of vertices ordered by depth. The complete buffer is mapped as vertex buffer and rendered as points. This can be extended, for example to create a depth-of-field effect (see Figure 8), using a geometry shader that creates quads whose size depends on the distance to the focal point.

6.2. Distributed Rendering

An important use case for Puxels is distributed parallel rendering of (partially) transparent, object-space decomposed models. This can be easily implemented by having each render node send its buffers to a display node. In a merge step all incoming per-pixel data is combined into the per-pixel data of the display node. Afterwards, the normal rendering pass takes place including the depth-sorting of the collected fragments. Alternatively, the fragments can be sorted remotely and simply merged on the display node in $\mathcal{O}(n)$ steps.

6.3. Alternate SES Rendering

In addition to the transparent SES rendering explained above, there is another way which we want to detail here. As the problem is to determine whether a fragment of a VdW sphere may be drawn or not, one has to determine if this fragment is inside the surface or not. This can be achieved by rendering the SAS and projecting these fragments back to the original VdW spheres. A fragment from an atom is only visible if the corresponding SAS sphere fragment is visible. Although this method can be implemented straightforwardly, the problem here is the massive overhead of SAS rendering. As the evaluation shows, our SES is faster than the SAS rendering and, thus, the alternative SES generation was not inspected further.

7. Summary and Future Work

We presented an approach for rendering transparent molecular surfaces using the Puxels framework for order-independent transparency. We extended existing methods and detailed the rendering of the transparent molecule surfaces. In particular, our method solves the difficult removal of internal fragments of the SES using CSG operations. Additionally, we discussed further visualization applications possible when using Puxels. The performance of our approach was compared to and outperforms dual depth peeling, which serves as a ground truth OIT rendering method.

For future work, we want to investigate how to combine post-processing effects like ambient occlusion or cel shading with transparency which might help to understand complex visualizations. To make the OIT rendering generally applicable to even larger models and more complex scenes, we want to extend the Puxels framework to distributed network rendering. Here, especially the amount of fragments generated and sent over the network needs attention to minimize the data overhead. Additionally, we plan to use the Puxel data for advanced effects, like global illumination, reflection and refraction, and volume rendering.

Acknowledgments

This work was partially funded by Deutsche Forschungsgemeinschaft as SFB 716 projects D.3 & D.4, and by the Federal Ministry of Education and Research of Germany as part of FeToL and MCSimVis.

References

[BCL*07] Bavoil L., Callahan S. P., Lefohn A., Comba J. L. D., Silva C. T.: Multi-fragment effects on the GPU using the k-buffer. In *Symposium on Interactive 3D graphics and games* (2007), pp. 97–104. 4

[BE11] Bavoil L., Enderton E.: *Constant-Memory Order-Independent Transparency Techniques*. 2011. 4

[Bli82] Blinn J.: A Generalization of Algebraic Surface Drawing. *ACM Trans. Graph. 1* (1982), 235–256. 3

[BM08] Bavoil L., Myers K.: *Order Independent Transparency with Dual Depth Peeling*. 2008. 4, 6

[BWF*00] Berman H., Westbrook J., Feng Z., Gilliland G., Bhat T., Weissig H., Shindyalov I., Bourne P.: The Protein Data Bank. *Nucl. Acids Res. 28* (2000), 235–242. http://www.pdb.org. 6

[Car84] Carpenter L.: The a -buffer, an antialiased hidden surface method. *SIGGRAPH Comput. Graph. 18*, 3 (1984), 103–108. 4

[Con83] Connolly M. L.: Analytical Molecular Surface Calculation. *J. Appl. Cryst. 16* (1983), 548–558. 3

[CSN*12] Chen G., Sander P. V., Nehab D., Yang L., Hu L.: Depth-presorted triangle lists. *ACM Trans. Graph. 31*, 6 (Nov. 2012), 160:1–160:9. 3

[Ede99] Edelsbrunner H.: Deformable smooth surface design. *Discrete & Computational Geometry 21*, 1 (1999), 87–115. 3

[EM94] Edelsbrunner H., Mücke E. P.: Three-dimensional alpha shapes. *ACM Trans. Graph. 13*, 1 (1994), 43–72. 3, 5

[Eve01] Everitt C.: *Interactive Order-Independent Transparency*. NVIDIA Corp., 2001. 4

[FvDFH90] Foley J. D., van Dam A., Feiner S. K., Hughes J. F.: *Computer graphics: principles and practice (2nd ed.)*. Addison-Wesley Longman, 1990. 3

[GB78] Greer J., Bush B. L.: Macromolecular shape and surface maps by solvent exclusion. In *Proceedings of the National Academy of Science* (1978), pp. 303–307. 2

[Gum03] Gumhold S.: Splatting Illuminated Ellipsoids with Depth Correction. In *Proceedings of VMV* (2003), pp. 245 – 252. 2, 3

[HDS96] Humphrey W., Dalke A., Schulten K.: VMD – Visual Molecular Dynamics. *Journal of Molecular Graphics 14* (1996), 33–38. 6

[KBE09] Krone M., Bidmon K., Ertl T.: Interactive visualization of molecular surface dynamics. *IEEE Trans. Vis. Comp. Graph. 15*, 6 (2009). 3, 5

[KGE11] Krone M., Grottel S., Ertl T.: Parallel Contour-Buildup Algorithm for the Molecular Surface. In *IEEE Symposium on Biological Data Visualization (biovis'11)* (2011), pp. 17–22. 3

[KSES12] Krone M., Stone J. E., Ertl T., Schulten K.: Fast Visualization of Gaussian Density Surfaces for Molecular Dynamics and Particle System Trajectories. In *EuroVis Short Papers* (2012), vol. 1, pp. 67–71. 6

[LB02] Laug P., Borouchaki H.: Molecular Surface Modeling and Meshing. *Engineering with Computers 18*, 3 (2002), 199–210. 3

[LBPH10] Lindow N., Baum D., Prohaska S., Hege H.-C.: Accelerated Visualization of Dynamic Molecular Surfaces. *Computer Graphics Forum 29* (2010), 943–952. 3

[LHLW09] Liu F., Huang M.-C., Liu X.-H., Wu E.-H.: Efficient depth peeling via bucket sort. In *Proceedings of the Conference on High Performance Graphics 2009* (2009), ACM, pp. 51–57. 4, 6

[MDG*10] Marsalek L., Dehof A., Georgiev I., Lenhof H.-P., Slusallek P., Hildebrandt A.: Real-Time Ray Tracing of Complex Molecular Scenes. In *14th International Conference on Information Visualization (IV)* (2010), pp. 239–245. 3

[MHLK06] Moll A., Hildebrandt A., Lenhof H.-P., Kohlbacher O.: Ballview: A tool for research and education in molecular modeling. *Bioinformatics 22*, 3 (2006), 365–366. 3

[PV12] Parulek J., Viola I.: Implicit Representation of Molecular Surfaces. In *IEEE Pacific Visualization Symposium* (2012), pp. 217–224. 3, 5

[RE05] Reina G., Ertl T.: Hardware-Accelerated Glyphs for Mono- and Dipoles in Molecular Dynamics Visualization. In *EuroVis05: IEEE VGTC Symposium on Visualization* (2005), pp. 177–182. 2, 3

[Ric77] Richards F. M.: Areas, Volumes, Packing, and Protein Structure. *Annual Review of Biophysics and Bioengineering 6*, 1 (1977), 151–176. 2

[SOS96] Sanner M. F., Olson A. J., Spehner J.-C.: Reduced Surface: An efficient way to compute molecular surfaces. *Biopolymers 38*, 3 (1996), 305–320. 3, 5

[TA95] Totrov M., Abagyan R.: The contour-buildup algorithm to calculate the analytical molecular surface. *Journal of Structural Biology 116* (1995), 138–143. 3

[VBW94] Varshney A., Brooks F. P., Wright W. V.: Linearly Scalable Computation of Smooth Molecular Surfaces. *IEEE Computer Graphics and Applications 14*, 5 (1994), 19–25. 3, 5

[VF11] Vasilakis A. A., Fudos I.: Z-fighting aware depth peeling. In *ACM SIGGRAPH 2011 Posters* (2011), SIGGRAPH '11, ACM, pp. 77:1–77:1. 4

[Wal04] Wald I.: *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004. 3

[Wes90] Westover L.: Footprint evaluation for volume rendering. SIGGRAPH '90, ACM, pp. 367–376. 6

[Wit01] Wittenbrink C. M.: R-buffer: a pointerless a-buffer hardware architecture. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (2001), 73–80. ACM ID: 383529. 4

[YHGT10] Yang J. C., Hensley J., Grün H., Thibieroz N.: Real-time concurrent linked list construction on the GPU. *Computer Graphics Forum 29*, 4 (2010), 1297–1304. 4

[ZP07] Zhang Y., Pajarola R.: Deferred blending: Image composition for single-pass point rendering. *COMPUTER & GRAPHICS 31* (2007), 175–189. 3