

Order Independent Transparency with Per-Pixel Linked Lists

Pál Barta*

Balázs Kovács†

Supervised by: László Szécsi‡ and László Szirmay-Kalos§

Budapest University of Technology and Economics
Budapest / Hungary

Abstract

This paper proposes a method for rendering scenes of both opaque and transparent objects. Transparency depends on the attenuation coefficient and the thickness of the transparent object we wish to render. To get the visible radiance, the volume rendering equation should be solved. Instead of marching a ray, we build a list that contains the intersection points of the ray and object surfaces. In the second phase of rendering, the GPU sorts and processes the lists and evaluates the attenuation integrals analytically, considering also the order of the segments. This solution is mathematically correct even if objects intersect, i.e. it does not involve drastic simplifications, and provides high framerates even on moderately complex scenes, outperforming previous methods. In addition to transparent objects, the technique is also appropriate to visualize natural phenomena represented by particle systems.

Keywords: Transparency, Direct3D 11, Linked-lists, GPU, Particle Systems.

1 Introduction

The technique of alpha blending has a long history in two- and three-dimensional image synthesis. There are many ways to blend colors [10], but the most important issue is that we can only get realistic results if we sort transparent objects by their distance from the camera. Unfortunately, this requirement is not compatible with incremental rendering and z-buffer based visibility determination, which allow the processing of objects in an arbitrary order. Sorting objects or even triangles in a way that occluders follow objects occluded by them is difficult and is usually impossible without further subdivision of objects. The problem is that an object is associated with a depth interval and not with a single distance value, so no direct ordering relation can be established. A possible solution for non-intersecting triangles is the application of the painters algorithm [1], but this has super-linear complexity and its

GPU implementation is prohibitively complicated.

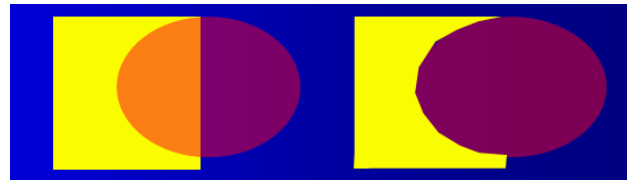


Figure 1: Order matters when the scene contains transparent objects.

If objects may intersect each other, then the situation gets even worse. A typical case of intersecting transparent objects are particle systems, which are tools to discretize, simulate and visualize natural phenomena like fog, smoke, fire, cloud, etc. The simplest way of their visualization applies planar billboards, but this approach results in abrupt changes where particles intersect opaque objects. The solution for this problem is the consideration of the spherical extent of the particle during rendering, as proposed in the concept of *spherical billboards* [9], also called *soft particles*. Spherical billboards nicely eliminate billboard clipping and popping artifacts at a negligible additional computational cost, but they may still create artifacts where particles intersect each other. Most importantly, when the z-order of billboards changes due to the motion of the particle system or the camera, popping occurs. This effect is more pronounced if particles have non-identical colors or textures.

Instead of executing the sorting for the objects, we can as well ensure the correct order on the level of fragments. This approach does not require the sorting of the objects on the CPU, which is emphasized by its name, *order independent transparency*. The family of such methods is usually referred to as *depth peeling*. The basic idea of depth peeling is that the fragment shader may discard fragments that are not farther than a previously selected threshold and the depth buffer will identify the closest fragment from the not discarded points. Thus, the scene is rendered multiple times and each time we ignore the already identified layers. Intuitively, we peel layer surfaces from the scene. Depth peeling has been used in global radiosity [6] and in transparency [3, 8] calculation as well. Unfortunately, depth peeling needs to render the scene multiple times, de-

*brazil.hu@gmail.com

†kockafely@gmail.com

‡szecsi@iit.bme.hu

§szirmay@iit.bme.hu

pending on the *depth complexity* of the scene (the depth complexity is defined as the maximum number of intersections a ray has in a given scene). Even its advanced versions, like *Dual Depth Peeling* [5], *Reverse Depth Peeling* [8] etc. could not be used effectively in real-time scenes without limiting the sorting to just the first few layers.

AMD presented a special solution at the Game Developers Conference 2010 [4]. The latest series of ATI Radeon supports DirectX11, which opened the door for new rendering algorithms. The latest Shader Model 5.0 GPUs have new features like the *read/write structured buffers* or the *atomic operations* to manipulate them. Recall that before Shader Model 5.0, shaders may either read memory (e.g. input textures) or write it (e.g. render target), but not both, and writes are always exclusive, so no synchronization is necessary. This limitation has been lifted by Shader Model 5.0, and now we do not have to wait for the end of a pass before reading back the result in a shader processor. With the use of these features, we are able to process the incoming fragments in a complex way instead of writing them to the frame buffer. The fragments can be stored in linked lists, which creates new ways to implement order-independent alpha-blending.

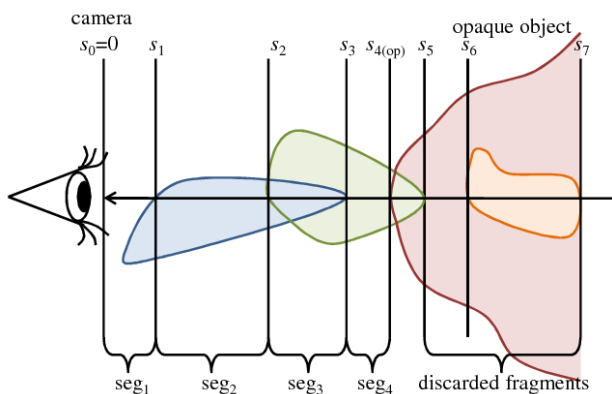


Figure 2: Model of a scene that contains opaque and transparent objects. Each transparent object is homogeneous and they can intersect each other.

This paper proposes a new algorithm to render transparent, possibly intersecting objects and particles based on DirectX11's structured buffers and atomic operations. In Section 2, we first survey the model of light transport in homogeneous transparent objects. Then, in Section 3 the new, GPU-based algorithm is discussed. Finally, we present results and conclusions.

2 Model of light transport in homogeneous objects

In case of scenes having only opaque objects the radiance is constant along a ray and scattering may occur just on object surfaces. Participating media, however, may scatter

light not only on their boundary, but anywhere inside their volume. Participating media can be imagined as some material that does not completely fill the space. Thus the photons have the chance to go into the media and to travel a random distance before collision. To describe light-volume interaction, the basic rendering equation should be extended [7, 2]. The volumetric rendering equation is obtained considering how the light goes through participating media (Figure 3).

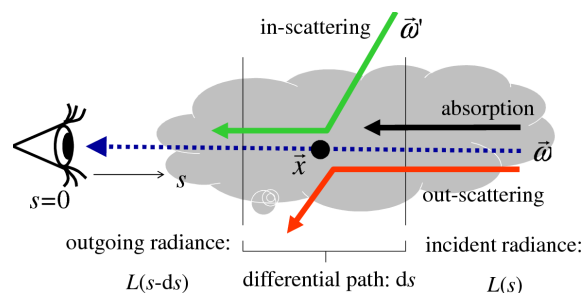


Figure 3: Change of radiance in participating media.

The change of radiance L on a path of differential length ds and of direction $\vec{\omega}$ depends on different phenomena:

Absorption and out-scattering: Photons may collide with the material and the material may or may not reflect the photon after collision. The intensity change is proportional to the number of photons entering the path, i.e. the radiance and the probability of collision. If the probability of collision in a unit distance is τ , then the probability of collision along infinitesimal distance ds is τds . After collision the particle is reflected with the probability of *albedo* a , and absorbed with probability $1 - a$. Collision density τ and the albedo may also depend on the wavelength of the light. Summarizing, the total radiance change due to absorption and out-scattering is $-\tau L ds$.

In-scattering: Photons originally flying in a different direction may be scattered into the considered direction. The expected number of scattered photons from differential solid angle $d\omega'$ equals to the product of the number of incoming photons and the probability that the photon is scattered from $d\omega'$ to $\vec{\omega}$ in distance ds . The scattering probability is the product of the collision probability (τds), the probability of not absorbing the photon (a), and the probability density of the reflection direction $\vec{\omega}$, given that the photon arrived from direction $\vec{\omega}'$, which is called *phase function* $P(\omega', \omega)$. Following an ambient lighting model, we assume that the incident radiance is L^a in all directions and at every point of the scene. Taking into account all incoming directions Ω' , the radiance in-

crease due to in-scattering is:

$$\tau a ds \left(\int_{\Omega'} L^a P(\omega', \omega) d\omega' \right) = \tau a L^a ds$$

since the phase function is a probability density, thus its integral over the full directional domain equals to 1.

Adding the discussed changes, we obtain the following *volumetric rendering equation* for the radiance L of a ray at $s - ds$ having taken step ds toward the eye:

$$L(s - ds, \vec{\omega}) = (1 - \tau ds) L(s, \vec{\omega}) + \tau a L^a ds. \quad (1)$$

Subtracting $L(s)$ from both sides and dividing the equation by ds , the volumetric rendering equation becomes a differential equation.

$$-\frac{dL(s, \vec{\omega})}{ds} = -\tau L(s, \vec{\omega}) + \tau a L^a. \quad (2)$$

In homogeneous media, volume properties τ and a are constant. In our model, the scene contains homogeneous objects having different materials. Thus, in our case, the properties are piece-wise constant functions along a ray.

2.1 Solution of the simplified volumetric equation

The radiance along a ray is described by an inhomogeneous first-order linear differential equation, which can be solved analytically. Assuming that the background radiance is zero, the radiance at the eye position ($s = 0$) is:

$$L(0, \vec{\omega}) = \int_0^{\infty} \tau(s) a(s) L^a e^{-\int_0^s \tau(x) dx} ds$$

where direction $\vec{\omega}$ points from the pixel towards the eye. Let us now exploit the fact that material properties $a(s)$ and $\tau(s)$ are piece-wise constant functions, they may change where the ray intersects the surface of an object. Let us denote the distance values of ray surface intersections by s_1, s_2, \dots, s_n and extend this ordered list by $s_0 = 0$ and $s_{n+1} = \infty$. The ray, i.e. the domain of the integration is partitioned according to the segments between the intersection points, where albedo $a(s)$ and attenuation parameter $\tau(s)$ are equal to a_i and τ_i in segment $[s_{i-1}, s_i)$, respectively:

$$L(0, \vec{\omega}) = \sum_{i=1}^{n+1} \int_{s_{i-1}}^{s_i} \tau_i a_i L^a e^{-\int_0^s \tau(x) dx} ds.$$

Then, we also partition the $[0, s]$ interval according to the intersection points in the attenuation formula, assuming that s is in $[s_{i-1}, s_i]$:

$$\int_0^s \tau(x) dx = \tau_i (s - s_{i-1}) + \sum_{j=1}^{i-1} \tau_j (s_j - s_{j-1}).$$

Thus, the exponential decay is:

$$e^{-\int_0^s \tau(x) dx} = e^{-\tau_i (s - s_{i-1})} \prod_{j=1}^{i-1} e^{-\tau_j (s_j - s_{j-1})}.$$

Substituting this back into the eye radiance, we obtain:

$$L(0, \vec{\omega}) = \sum_{i=1}^{n+1} a_i L^a \int_{s_{i-1}}^{s_i} \tau_i e^{-\tau_i (s - s_{i-1})} ds \prod_{j=1}^{i-1} e^{-\tau_j (s_j - s_{j-1})}.$$

We can obtain better quality rendering with shading without significantly slowing down the rendering. We are using Rayleigh shading where the phase function is:

$$P(\cos \theta) = \frac{3}{16\pi} (1 + \cos^2 \theta)$$

θ is the angle between the light and the view direction. For simplicity we assume the light source is directional, thus we can foil the Rayleigh term before the integral because θ will be constant in a segment. Therefore we can substitute L^a in our equations by

$$L^s = L^a + L^r \frac{3}{16\pi} (1 + \cos^2 \theta).$$

We introduce a shorthand notation for the *color contribution* C_i of a segment:

$$C_i = a_i L^s \int_{s_{i-1}}^{s_i} \tau_i e^{-\tau_i (s - s_{i-1})} ds = a_i L^s \left(1 - e^{-\tau_i (s_i - s_{i-1})} \right). \quad (3)$$

Note that this formula remains valid also for the $\tau_i = 0$ case, i.e. when the ray travels in free space.

Similarly to the contribution, *segment transparency* T_i can also be applied to segment i

$$T_i = e^{-\tau_j (s_{j+1} - s_j)}. \quad (4)$$

With these shorthand notations, the radiance associated with a particular pixel is

$$L(0, \vec{\omega}) = \sum_{i=1}^{n+1} \left(C_i \prod_{j=1}^{i-1} T_j \right). \quad (5)$$

The evaluation of this formula requires the intersection points to be sorted and segments to be visited in the sorted order. If the distances are sorted in ascending order, at each segment two equations should be evaluated iteratively, starting with $L = 0$ and $T = 1$:

$$\begin{aligned} L &\leftarrow L + C_i T, \\ T &\leftarrow T \cdot T_i. \end{aligned}$$

When the last segment is processed, variable L contains the radiance of the ray.

There are two critical issues concerning the iterative evaluation of these formulae. First, segments should be

stored and sorted on the GPU, without knowing in advance how many segments a particular ray has. On the other hand, the properties of the segments, including the albedo and the attenuation parameter, should be determined from the object properties. As objects may intersect each other, this question cannot be simply answered by checking whose surface the ray has most recently crossed. These problems are solved by the algorithm presented in the next section.

3 The rendering algorithm

The algorithm consists of two main steps: At first we collect every intersection point between all rays and object surfaces in lists associated with rays. Then, in the second phase, the lists are sorted and processed to get the pixel colors.

3.1 Data structures

In order to get not only the first ray surface intersection, but all intersections of an eye ray, the latest features of the DirectX 11 compatible GPUs are exploited, including structured buffers and atomic writes.

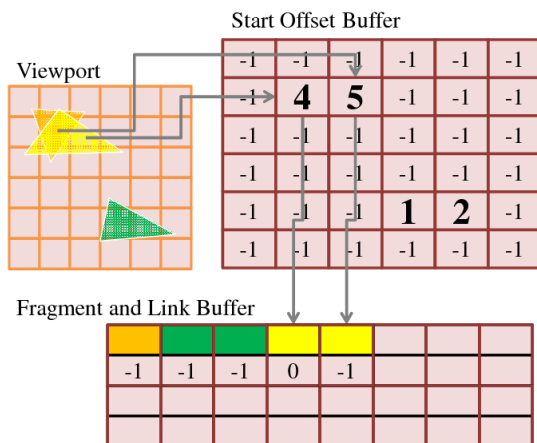


Figure 4: Data structures.

In particular, we use types of “Byte Address Buffer” and “Read/Write Structured Buffer”. The first type is used as the starting element of the linked lists, while the buffer of the other type is filled with the fragment data structure. We need also a texture containing the opaque scene with depth information. The main data stores are the “Fragment and Link Buffer”, the “Start Offset Buffer”, and the “Opaque Scene Texture” (Figure 5).

Fragment and Link Buffer: The output data of the first-phase pixel shader contains the radiance reflected at the surface, the volumetric attenuation coefficient and the albedo of the object, a flag whether or not the surface is front facing, and the distance of the fragment

from the eye. We would like to store these values in a structure and build a list of them for each pixel. Instead of writing data to the frame buffer, we have to store them in a special buffer called “Read/Write Structured Buffer”. It is a generic buffer that contains the declared type of structure. We have to append a pointer to each structure, which addresses the next fragment in the list. The value -1 as address denotes the end of the list. The size of the buffer depends on the estimated amount of transparent fragments in the viewing frustum. If the allocated memory is not enough for all the transparent fragments, then it will overflow and we will lose important data. We set a counter for the buffer, its initial value is 0. When the shader appends a new structure of fragment data to the buffer, the value of the counter will provide the address of the new element. Afterwards we increment the counter, so it always addresses the next free slot in the Fragment and Link Buffer. We should update the counter with atomic operators, because it is parallelly used by a number of shader units.

Start Offset Buffer: The type of this buffer is the “Byte Address Buffer”, which holds 4-byte addresses. The function of this data structure is to refer to the first element of the linked list for every pixel. The first element is always the last fragment that was processed for that list. When a new structure is appended, the pointer of the structure will get the value of the associated element in the “Start Offset Buffer”. Accordingly, we write the address of the newly stored structure in the buffer. We have to allocate memory to store one address for each pixel on the viewport.

Opaque Scene Texture: This shader resource holds the RGB values of the opaque scene, and the distance of the fragment from the eye, which is stored in the alpha channel. When processing the list, the color read from this texture is also blended and the fragments being farther than this opaque fragment are discarded.

3.2 Collecting the fragments

The method starts with rendering opaque objects. The alpha channel is used to store the distance of the fragment from the camera, so later the list of fragments can be cropped based on this value.

The next step is to render the transparent objects. The drawing function sets up the pipeline and passes the vertex information to the GPU. It is important that the culling of the back-facing fragments must be disabled, otherwise the GPU discards them and lot of important data about the transparent scene will be lost. The vertex shader performs the standard transformations, the novel part of the rendering comes with the pixel shader. When the pixel shader gets a fragment as input, it allocates and fills a structure, which contains the surface radiance, the volumetric attenuation and albedo, the distance from the cam-

era, and the orientation of the fragments. Then the shader stores the structure in the “Fragment and Link Buffer” at the next free slot and sets the “Start Offset Buffer” using the screen coordinates of the currently processed fragment. The pointer in the “Start Offset Buffer” at the address of the screen coordinate will reference this structure, and the pointer of the newly inserted structure gets the former value of the address buffer. This way a new element is inserted into the linked list.

3.3 Loading and sorting the lists

After the lists are created for each pixel and all the visible fragment information is in the memory of the GPU, the second phase of the rendering begins. The next step is to parse each list in a local buffer, then sort the elements in ascending order of the distance from the camera. Loading the lists into the local buffer of the shaders at first is important because working with local data is easier and faster than reading each structure separately from a shared resource. To avoid copying larger set of data, an index buffer should also be created, so during the sort the GPU moves only indices instead of structures.

3.4 The sorting

Every sorting algorithm has its own characteristic, which offers advantages and disadvantages depending on the environment parameters of the sorting. These algorithms are based on different concepts like partitioning, merging, selection or insertion, some of them providing bigger overheads or different efficiency under different conditions. The implementation of the methods can also have a slight effect on their performance. At first we have to give an estimated number of the elements of the array we have to sort. In our case the fragments for each pixel are stored in these arrays, the GPU will sort each array independently and will run the same sorting algorithm for each pixel. So we have one array of fragments in our shader code, which belongs to the currently processed pixel. Most of the cases the rendered scene contains some transparent objects in an opaque environment, but even if we are experimenting with more complex composition, it is reasonable to say that the average depth complexity of a transparent scene is lower than 100 layers. The algorithms have average, worst and best case scenarios, since the GPU will run the sorting many thousand times for each frame, we have to analyze the average speed of the method. Another important attribute to consider is whether the elements are presorted in some way. We can assume here, that the fragments are collected randomly. However, since we have information about the order of the objects drawn to the screen, some kind of presorting could be possible, but right now this enhancement is left for the future. The computational complexity is maybe the most important parameter of an algorithm along with memory usage, we try to find a sorting algorithm that

performs the best under the above mentioned conditions and needs the lowest amount of memory space. Quicksort is one of the most popular sorting algorithms and its optimized versions are considered the fastest of them. As a general rule this statement can be true, however, it is a bit more complex than the simpler methods, which provides remarkable overhead. Additionally the current GPUs do not support recursive calls, and the basic quicksort is a recursive algorithm. Its non-recursive version needs also a stack to implement generating more overhead. In the case of relatively small lists a much simpler algorithm can perform better and the insertion sort is a popular choice. Its disadvantages begin to come forward on longer lists, while sorting an array of 100 items the insertion sort outperforms the other sorting algorithms. (<http://warp.povusers.org/SortComparison/integers.html>) Additionally its easy to implement, does not change the relative order of elements with equal keys, only requires a constant amount of additional memory space and performs even more faster on pre-sorted lists. These attributes makes us insertion sort the best choice here.

3.5 Processing the lists

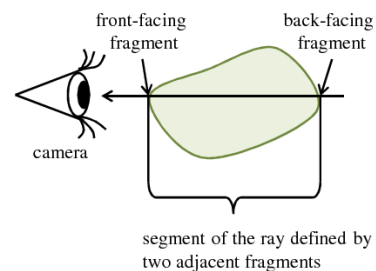


Figure 5: Distinction of front and back facing surfaces.

After the sorting, the shader blends the fragments according to the physical model of light absorption, and every segment yields its contribution in the final value. When the GPU finishes sorting the list, the sorted array of fragments divides the ray into *segments*. In our model, both the intersecting and the stand-alone objects divide the space into homogenous regions, which are represented by the segments defined by the list of the fragments. Each segment possesses properties of contributed color C_i and transparency T_i (equations 3 and 4). These values are based on the attributes of the objects and the length between consecutive intersections.

If objects intersect, the contributed color and the transparency should be computed from the length the ray travels in the intersection and the combined attenuation and albedo. Suppose that the attenuation coefficients are τ_1 and τ_2 in object 1 and object 2, respectively, and similarly their albedos are a_1 and a_2 . In their intersection, the particles of both objects are present, so the probability density of photon-particle collision is the sum of the elementary

probability densities:

$$\tau = \tau_1 + \tau_2.$$

Considering the albedo, i.e. the conditional probability that reflection happens given that the photon collides, we can obtain the following combined value:

$$a = \frac{\tau_1 a_1}{\tau_1 + \tau_2} + \frac{\tau_2 a_2}{\tau_1 + \tau_2}.$$

The properties of the segments can be determined while traversing the list. The list items represent surfaces, i.e. object boundaries, where the optical properties of only the respective object can be fetched. Thus, the optical properties of the segments, including the combination of the albedos and the attenuation coefficients, need to be computed on the fly while we are traversing the list.

Suppose, we maintain two running variables a and τ that represent the albedo and the attenuation coefficient of the current segment. When a list item is processed, i.e. when we cross a surface, these running variables are updated. If the list item represents a front facing surface, then we enter object o whose parameters are stored in the list, in variables a_o and τ_o . Consequently, the running albedo and attenuation are updated to reflect the properties of this new object:

$$\begin{aligned} a &\leftarrow (\tau a + \tau_o a_o) / (\tau + \tau_o), \\ \tau &\leftarrow \tau + \tau_o. \end{aligned}$$

When the traversal encounters a back-facing fragment, we leave an object, thus its attenuation and albedo should be removed from the combined values. To execute this, the inverse of the previous formulae should be evaluated:

$$\begin{aligned} \tau &\leftarrow \tau - \tau_o, \\ a &\leftarrow (\tau a + \tau_o a - \tau_o a_o) / \tau. \end{aligned}$$

In order to initialize these iterations when we start the list traversal from the eye position, we should determine which objects contain the eye position. This information can be found by traversing the ordered list backward starting at the farthest intersection point, where we assume that we came from free space, thus $\tau = 0$ and $a = 1$. During this initial traversal, the same operations are performed, just the roles of front-facing and back-facing segments are exchanged.

4 Results

The presented algorithm has been implemented in a DirectX 11/HLSL environment on an ATI Radeon 5700 graphics card. The modeled scene consists of 50000 opaque triangles and 150000 transparent triangles, the resolution for the tests is set to 800×600 . The frame rate mainly depends on the depth complexity of the currently

rendered scene. If no transparent object is present, the performance is about 110 FPS because of the overhead of the two-step process. As the depth complexity and the transparent area grow, the frame rate decreases. If there are more than 20-30 layers of transparent surfaces, the performance falls below 10 FPS.

The test scene for the table above consists of full screen layers, so each frame's linked list contains the same number of layers.

Table 1: Results with 800×600 resolution

Layer count	FPS
2	90
4	49
6	32
8	23
10	18
12	14
14	12
16	10

Table 2: Results with 1024×768 resolution

Layer count	FPS
2	51
4	27
6	18
8	13
10	10
12	8
14	6
16	5

5 Conclusions

The paper introduces a real-time method for rendering transparent, and possibly intersecting objects considering the order of the fragments. The implementation is capable of running relatively high framerates and provides a mathematically correct and more realistic result. The previous approaches were able to render simple scenes with low resolution and frame rates, thus, the method presented here demonstrates a significant advancement. In the future this technique can be used in particle systems, improving the quality of particles and eliminating artifacts when interacting with each other. Currently, we can handle about a hundred particles in real-time. The plans for the future also involve the shading of the transparent objects. The current solution uses only ambient lights and Rayleigh shading, the introduction of various light sources and shadows would significantly improve the rendering quality.

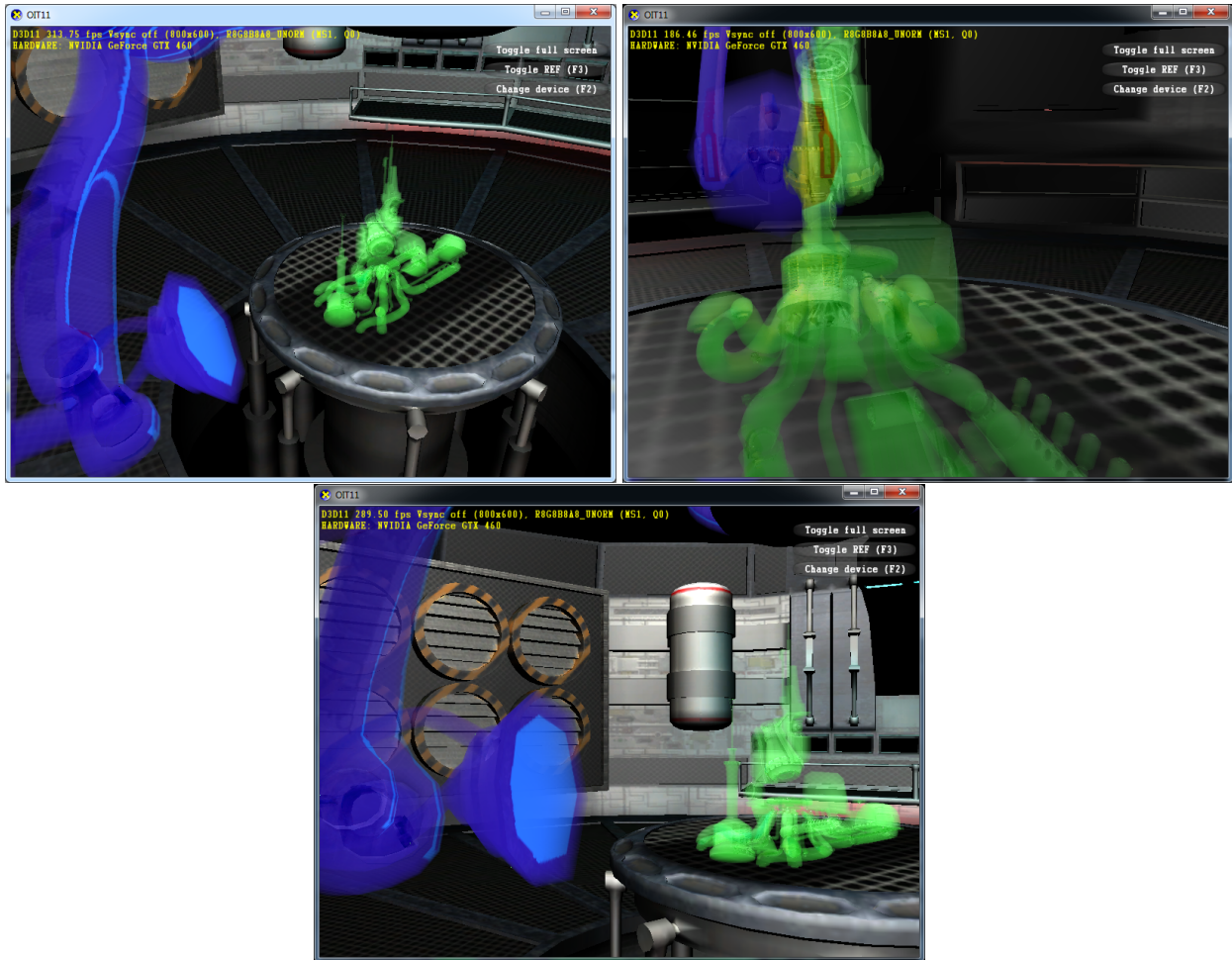


Figure 6: Rendering results (internal view of a space ship).

Acknowledgements

This work has been supported by OTKA K-719922, and by the scientific program of the “Development of quality-oriented and harmonized R+D+I strategy and functional model at BME” (Project ID: TMOP-4.2.1/B-09/1/KMR-2010-0002). The authors are grateful to NVIDIA for donating the GeForce 480 GPU cards.

References

- [1] M. de Berg. *Efficient Algorithms for Ray Shooting and Hidden Surface Removal*. PhD thesis, Rijksuniversiteit te Utrecht, The Netherlands, 1992.
- [2] Oskar Elek and Petr Knoch. Real-time spectral scattering in large-scale natural participating media. In Helwig Hauser and Reinhard Klein, editors, *Proceedings of Spring Conference on Computer Graphics 2010*, pages 83–90. Comenius University, Bratislava, 2010.
- [3] C. Everitt. Interactive order-independent transparency. Technical report, NVIDIA Corporation, 2001.
- [4] Nicolas Thibieroz Holger Gruen. Oit and indirect illumination using dx11 linked lists. In *Game Developers Conference*, 2010.
- [5] Kevin Myers Louis Bavoil. Order Independent Transparency with Dual Depth Peeling. 2008. http://developer.download.nvidia.com/SDK/10/opengl/src/dual_depth_peeling/doc/DualDepthPeeling.pdf.
- [6] L. Szirmay-Kalos and W. Purgathofer. Global ray-bundle tracing with hardware acceleration. In *Rendering Techniques '98*, pages 247–258, 1998.
- [7] L. Szirmay-Kalos, L. Szécsi, and M. Sbert. *GPU-Based Techniques for Global Illumination Effects*. Morgan and Claypool Publishers, San Rafael, USA, 2008.
- [8] N. Thibieroz. Robust order-independent transparency via reverse depth peeling in Direct3D 10. In Wolfgang Engel, editor, *ShaderX 6: Advanced Rendering Techniques*, pages 211–226. Charles River Media, 2008.
- [9] T. Umenhoffer, L. Szirmay-Kalos, and G. Szijártó. Spherical billboards and their application to rendering explosions. In *Graphics Interface*, pages 57–64, 2006.
- [10] Phil Willis. Projective alpha colour. *Computer Graphics Forum*, 25(3):557–566, 2006.