Pacific Graphics 2015
N. J. Mitra, J. Stam, and K. Xu
(Guest Editors)

Volume 34 (2015), Number 7

Order-Independent Transparency for Programmable Deferred Shading Pipelines

Andre Schollmeyer Andrey Babanin Bernd Froehlich

Bauhaus-Universität Weimar

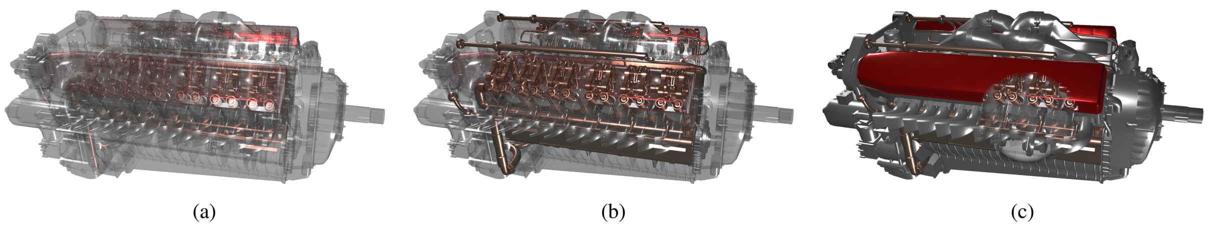


Figure 1: These are some image results for rendering an engine model using our pipeline. For the fully transparent engine (a), our system performs at about 60Hz. If only some parts are transparent (b), it runs at about 180Hz. The opacity remains programmable at fragment level which enables adaptive interaction tools, e.g. virtual see-through lenses (c).

Abstract

In this paper, we present a flexible and efficient approach for the integration of order-independent transparency into a deferred shading pipeline. The intermediate buffers for storing fragments to be shaded are extended with a dynamic and memory-efficient storage for transparent fragments. The transparency of an object is not fixed and remains programmable until fragment processing, which allows for the implementation of advanced materials effects, interaction techniques or adaptive fade-outs. Traversing costs for shading the transparent fragments are greatly reduced by introducing a tile-based light-culling pass. During deferred shading, opaque and transparent fragments are shaded and composited in front-to-back order using the retrieved lighting information and a physically-based shading model. In addition, we discuss various configurations of the system and further enhancements. Our results show that the system performs at interactive frame rates even for complex scenarios.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Display Algorithms

1. Introduction

In interactive 3D applications, transparency is a highly desired feature as it increases realism, spatial perception and the degree of immersion. However, supporting transparent objects has always been a challenge in real-time rendering systems. Hardware-accelerated rasterization is well-designed for rendering opaque geometry. It adapts the Z-buffer algorithm [Cat74], which keeps visible front-most surfaces, but discards the hidden. For visualizing non-opaque surfaces, a correct result is only possible if semi-transparent surfaces are sorted and blended in either front-to-back or back-to-front order. Presorting the geometry before rendering is computationally expensive and results in artifacts at triangle intersections. In contrast to geometry presorting, order-independent transparency (OIT) refers to a class of rendering techniques that achieve the correct result on a per-pixel basis. Most recent GPUs have gained support for atomic gather/scatter operations. These capabilities can be used to implement an A-Buffer [Car84], which stores the fragments generated during rasterization, to enable order-independent transparency and a large number of other multi-fragment effects. In particular, it has already been used for screen-space ambient occlusion [BKB13], depth of field [YWY10], screen-space collision detection [JH08], illustrative visualization for computer aided design (CAD) applications [CFM*12], constructive solid geometry (CSG) operations [RFV13] or nearest-neighbor search algorithms [RBA08, BGO09].

We designed a deferred shading pipeline with support for order-independent transparency by introducing transparency in the material concept. In our material description, transparency remains a programmable property which is either the result of user-defined computations, a texture look-up or simply a constant. During fragment processing, only the transparent fragments are routed into the A-Buffer while all other fragments are stored in a multi-layered geometry buffer. A light-culling pass is used to determine per-pixel lighting information. Once this information is acquired, we shade and blend all fragments in front-to-back order. In addition to our novel pipeline concept, we compared different state-of-the-art techniques for the generation of per-pixel linked-lists (PPLL) to find the most efficient approach for an A-Buffer implementation on recent graphics hardware.

Most modern rendering engines are based on deferred shading [ST90]. Some of these engines, e.g. Unity or the Unreal Engine 4 [Oli12], already have basic support for transparent objects. However, we found that all existing systems lack at least one of the following properties: programmability, performance or extensibility. The main reasons for this are the challenges to tackle when integrating an A-Buffer into a deferred shading pipeline. Therefore, we designed a new pipeline concept for the open-source rendering framework guacamole [SLB*14] that supports all of the aforementioned properties. The main features and contributions of our work are:

- An integration of order-independent transparency into an extensible deferred shading pipeline
- A programmable and easy-to-use material concept in which the transparency can be set at fragment level
- An efficient solution for light accumulation for transparent fragments that adapts the idea of tile-based shading
- Compile-time shader optimization is used to avoid an overhead for opaque objects

2. Background

The main challenge of integrating transparencies into a deferred shading pipeline is finding an efficient combination of two contradictory concepts of fragment processing. While a correct blending of transparencies requires the consideration of all the semi-transparent fragments per pixel, deferred shading pipelines are designed for opaque objects because they store only the front-most fragment for shading.

2.1. Partial Coverage and Blending

Porter and Duff [PD84] introduced compositing algebra, which defines a set of operations on images with partial coverage information (alpha channel). In particular, the **over**-operator is used to overlay one surface on top of the other assuming that both surfaces are partially transparent and no refraction is taking place when light passes through the medium. For a foreground surface A and background surface B, the **over**-operator is defined as follows:

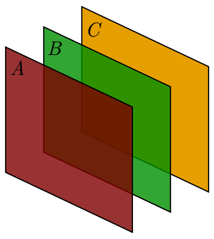


Figure 2: This Figure illustrates the correct compositing result of the surfaces A, B, C using the **over**-operator. It is applicable either in front-to-back $(A$ **over** $B)$ **over** C or back-to-front A **over** $(B$ **over** $C)$ order.

$$\mathbf{p}' = \mathbf{p}_A + (1 - \alpha_A)\mathbf{p}_B, \qquad (1)$$

where $\mathbf{p}_A$, $\mathbf{p}_B$ are image pixels of A and B, both pre-multiplied by their transparency α_A and α_B, respectively. The output $\mathbf{p}'$ is the resulting image pixel. A pixel $\mathbf{p}$ is defined as a quadruple (r, g, b, α) that holds three color components and its coverage α. The compositing of multiple surfaces is accomplished iteratively. However, the **over**-operator is not commutative. The transparent surfaces must be ordered either front-to-back or back-to-front to obtain the correct result, as shown in Figure 2.

2.2. Deferred Shading

In forward rendering, every fragment passing the depth test is shaded and stored in the frame buffer until it is replaced by a new fragment passing the depth test. For scenes with many lights and sophisticated shading, this approach may become inefficient because it needs to perform expensive shading computations for occluded fragments also. In deferred shading, geometry and light processing are decoupled [ST90].

In the first step, scene geometry is rendered without performing any shading computations. Instead, the data necessary for shading is gathered and stored in a so-called geometry buffer (G-buffer). In the second step, only the visible fragments stored in the G-buffer are shaded, which avoids wasting resources for occluded fragments. For scenes with many lights, it can also be advantageous to accumulate the light contributions in a separate pass, a technique also referred to as deferred lighting [AMHH08].

Unfortunately, standard deferred shading does not consider transparency effects. However, we do not want to reject the deferred approach as it performs very well for opaque geometry, which is probably dominant in most scenes. Instead, we want to find a way to combine them both, thereby benefiting from efficient rendering of opaque geometry and realistic transparency effects.

2.3. Transparency Effects in Real-time Rendering

The non-commutativity of Equation 1 used for compositing requires the surfaces to be sorted in either front-to-back or back-to-front order. For rasterization-based pipelines, this presents a major challenge because triangles are handled independently, disregarding their distance and orientation to the viewport. According to [MCTB11], the methods of sorting can be classified into the following categories: Depth-sorting independent, probabilistic approaches, geometry sorting and fragment sorting.

Sorting-independent techniques [MB13, BM08] approximate the compositing result without explicit ordering by depth. They can be performed in a single pass and do not need any buffer to store fragments. Despite their simplicity and high performance, these techniques do not guarantee the correct compositing of semi-transparent surfaces and in most cases, they produce visual artifacts. Therefore, their usage is limited to simple cases where quality is less important than performance. As a remedy, Maule et al. propose a hybrid approach [MCTB13] which performs fragment-sorting and correct compositing only for the front-most fragments, thereby balancing image quality, memory consumption and performance.

Stochastic transparency [ESSL11] is an example of the probabilistic approach. In their work, transparency effects are achieved by filling a multi-sampled texture by evaluating alpha-to-coverage probability based on a random sub-pixel stipple pattern. However, this technique suffers from severe noise if not enough samples are generated.

Geometry-sorting approaches explicitly sort all primitives by depth before drawing. Potential artifacts due to interpenetrating triangles or cyclic overlaps can be resolved by splitting the corresponding triangles. In most cases, however, sorting at the primitive level is too expensive. Therefore, some systems accept visual artifacts and use a coarse object-based depth-sorting instead.

In contrast to geometry sorting, fragment-sorting techniques work at lower granularity. After rasterization, the fragments are stored and sorted per pixel such that primitive presorting is not required. Early implementations such as depth-peeling [Eve01] did not scale well with an increasing amount of geometry. However, recent hardware advancements enable various approaches [YHGT10] [LHL14] [J14] for an efficient A-Buffer implementation. We compared existing techniques (see Section 5) in order to find the most efficient approach on the latest hardware and redesigned the rendering pipeline of guacamole [SLB*14] to support order-independent transparency.

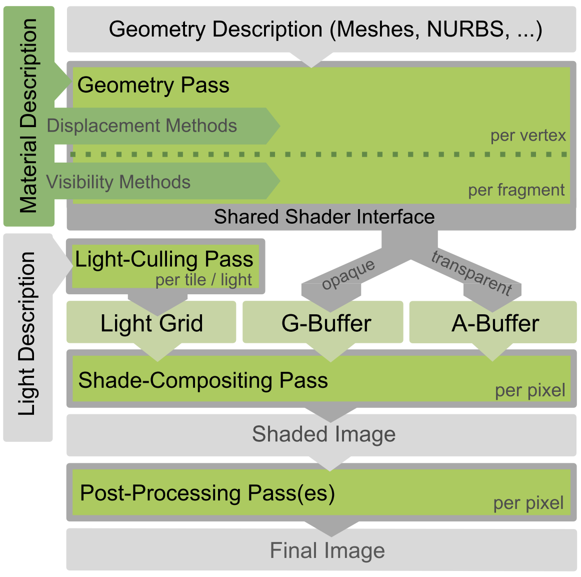


Figure 3: Rendering is accomplished by a configurable multi-pass pipeline based on deferred shading. In the first pass, the geometry descriptions are rendered into the G- and A-Buffer using the corresponding renderers. A shared shader interface and meta-programming methods are used to insert the material-dependent shader code which makes the system extensible for different geometry descriptions.

3. System Overview

Guacamole is an extensible, lightweight open-source scene graph and rendering engine based on deferred shading. Our redesigned pipeline concept presented in this paper does not depend on this specific framework, but should be applicable to any other deferred shading pipeline, as well. However, we will use our integrated system design to describe and discuss the general ideas of our approach.

Figure 3 illustrates our novel rendering concept in guacamole. In contrast to the originally proposed design [SLB*14], we employ a fixed G-buffer layout and physically-based rendering [KG13] for shading. It is based on a configurable multi-pass pipeline in which the user can define passes and their processing order. The minimal set of pipeline passes required to render a scene consists of the following three steps:

- **Geometry Pass**: This pass is responsible for the rasterization of the geometry descriptions in the scene. For all polygonal objects, a standard renderer is provided. In addition, the system can easily be extended with any kind of geometric representation and the corresponding rendering algorithm. This includes multi-pass techniques, as well as ray-casting based rendering approaches. This extensibility is demonstrated by the current support for trimmed NURBS [SF09], level-of-detail point clouds and 3D video avatars [BKKF13]. During fragment processing, each renderer passes the information necessary to defer shading to a *shared shader interface* which is independent of the type of geometry. The material computations are then applied to the fragment. Transparent fragments are submitted into an A-Buffer and, respectively, all opaque fragments into the G-buffer, as described in Section 4.1.

- **Light-Culling Pass**: Lighting information is necessary to shade the fragments gathered in the intermediate buffers. In this pass, light proxy geometries are rendered into a low-resolution grid. The result is a list of active lights for each grid cell. This idea adapts from tile-based shading [OA11], which was originally proposed for the efficient handling of a large number of light sources. In the context of OIT, we exploit the generated light grid to avoid frequent traversal of the fragments stored in the intermediate buffers. A detailed description of the light-culling pass is given in Section 4.3.
- **Shade-Compositing Pass**: Once the fragment and lighting information is gathered, shading and compositing is performed. For each pixel, we retrieve the list of active lights from the light grid and start traversing the fragments stored in the A- and G-Buffer. In front-to-back order, the fragments are shaded and blended using Equation 1 until the pixel's alpha value reaches a desired threshold. For details, see Section 4.4.

After the shade-compositing pass, the shaded image can be processed by additional screen-space passes. However, in this paper we do not elaborate on the possibilities of post-processing effects.

4. System Design and Pass Descriptions

In our system, rendering a given scene is accomplished by a *pipeline*. The resulting image can be used as input to another pipeline, which allows for multi-pass rendering. A pipeline is configured by the user by defining a set of passes and their order of execution. Some passes, such as post-processing, are optional, but the geometry pass, the light-culling pass and the shade-compositing pass, as well as their processing order, are compulsory. After culling and serialization, rendering is initiated by passing the scene objects to the geometry pass.

4.1. Geometry Pass

All renderable objects consist of a geometric description and a material. A material is programmable and consists of user-defined input and the corresponding shader code. Using a shared shader interface for all geometry representations allows us to insert the material-dependent source code into the geometry-specific programs at shader-compile time. During fragment processing, the inserted material methods may manipulate the transparency. Opaque fragments are then passed to the G-buffer, while transparent fragments are inserted into an A-buffer.

```
"displacement_stage" : [],
"visibility_stage" : [
{
"name" : "pbr_lens_fade_out",
"uniforms" :
[
{"name": "lens_pos", "type": "vec2", "value": "(0.5 0.5)"},
{"name": "lens_rad", "type": "float", "value": "0.3"},
{"name": "fade_dst", "type": "float", "value": "0.1"},
{"name": "roughtex", "type": "sampler2D", "value": "0"},
{"name": "alphatex", "type": "sampler2D", "value": "1"},
...
]
"source" :
void pbr_lens_fade_out()
{
  // set material coefficients and initial alpha
  gua_roughness = texture(roughtex, gua_texcoords).r;
  gua_alpha     = texture(alphatex, gua_texcoords).r;
  // ...

  // fade out close to see-through lens
  float lens_dist = length(lens_pos - gua_position.xy);
  float lens_fade_out = lens_dist / lens_rad;
  gua_alpha *= clamp(lens_fade_out, 0.0, 1.0);

  // fade out close to camera
  float ndepth = gl_FragCoord.z / gl_FragCoord.w;
  float depth_fade_out = ndepth / fade_dst;
  gua_alpha *= smoothstep(0.0, 1.0, depth_fade_out);
}
} ]
```

Figure 4: In a material description, the built-in variable *gua_alpha* can be used to set the transparency. In this example, in the visibility stage, the transparency is first initialized using a texture and then increased if the fragment is either close to the near plane or inside the radius of a virtual see-through lens.

4.1.1. Material Description

In our system, a material is an instance of a material description which consists of a set of user-defined input parameters and the corresponding shader code that performs the desired computations. A material description may provide methods for two stages: displacement and visibility. In the *displacement stage*, all material effects are applied that operate on vertex level, e.g. displacement mapping. The *visibility stage* operates per fragment and may modify all shading relevant parameters such as normal or albedo, as well as the transparency.

In contrast to other systems, the differentiation between opaque and transparent is carried out at fragment level, not per object. This is quite advantageous, especially if the opacity does not depend on the objects themselves, but on the current view or other parameters. For example, in many virtual-reality applications, it is desirable to fade out objects close to the viewer because the stereoscopic perception becomes uncomfortable. Figure 4 shows an example of a material description in our system. In this material, the transparency of a fragment depends on multiple parameters: an alpha texture, a virtual see-through lens (as shown in Figure 1c) and the distance to the viewer.

```
@define_fragment_shader_interface@
// inserts shared interface for geometry and material, e.g.
//   vec3  gua_world_position;
//   float gua_alpha;
//   ...

@define_material_uniforms@
@define_material_methods@

void main () {
  @map_rasterization_output@

  perform_ray_casting();

  @invoke_material_methods@
  @submit_fragment@ // to G- or A-buffer (see Fig. 5)
}
```

Figure 5: This simplified pseudo-code example illustrates a fragment program for a ray-casting based renderer in our system. The placeholders in between @ are replaced by the corresponding source code before shader compilation. After ray casting, the material is applied and the fragment is submitted into the corresponding buffer.

4.1.2. Shared Shader Interface

Non-trivial geometry descriptions typically require sophisticated rendering algorithms, e.g. ray casting or multi-pass approaches. In most cases, this involves designated shader programs. However, the rendering technique itself should be independent of the applied material. A decoupling between rendering algorithm and material computations could be achieved by a two-pass solution using an additional set of off-screen render targets as an intermediate result. However, this would increase the bandwidth requirements considerably and the support for transparencies would magnify this overhead. Instead, we provide a generic interface which helps us to merge the shader code of the geometry and the material description using meta-programming techniques.

Figure 5 shows a pseudo-code example of the fragment stage of a geometry program used in our system. The shared interface consists of placeholders which are replaced before shader compilation with the corresponding definitions or invocations. Based on this interface, all geometry and material computations are performed in a single program. The material may modify all shading-relevant data (position, normal, alpha, albedo, etc.) or even discard the fragment before it is stored in one of the intermediate buffers.

4.2. A-buffer Generation

The A-buffer is implemented using the lock-free insertion sort with early termination, as described in [LHL14]. In our performance experiments, we compared this approach to other techniques and it shows the best results (see Section 5) on most recent hardware. More importantly, it does not require a separate sorting pass because the fragments are sorted during insertion. This has two major advantages. First, it reduces the heavy workload and register-usage of the

```
void submit_fragment() {
  manual_depth_test(); // discard hidden fragments

  // try to insert transparent fragments in A-buffer
  if (gua_alpha < 1.0) {
    if (gua_write_to_A_buffer()) {
      discard; // success, fragment can be discarded
    } else {
      // failure, saturation reached -> write depth
      gua_write_to_G_buffer();
    }
  } else {
    gua_write_to_G_buffer(); // write opaque fragments
  }
}
```

Figure 6: This Figure illustrates the submission of a fragment. If saturation is reached, the depth is written to the G-buffer to enable manual Z-culling for further fragments.

compositing pass which already performs all shading and blending computations. Secondly, it enables early termination based on the accumulated opacity of the pixel.

In the presence of non-opaque objects, the geometry pass decides in which buffer a fragment is stored, depending on its final alpha value. In general, a fragment with an opacity of less than 100 % is inserted into the A-buffer and then discarded. Otherwise, it is written to the G-buffer, as illustrated in Figure 6.

Utilizing the meta-programming capability of guacamole allows the user to manipulate the opacity value at fragment level. This gives a maximum flexibility in managing object transparency, which is especially useful for the implementation of sophisticated interaction techniques such as show-through techniques in co-located collaborative virtual environments [AKK*11] or group navigation with fading-out obstacles [KKB*11]. Furthermore, it enables advanced materials with a view-dependent transparency, e.g. based on the Fresnel factor. At the same time, it maintains high performance because all opaque fragments are routed into the G-buffer. Moreover, for materials that do not manipulate the alpha value, the shader optimization will automatically remove the entire A-buffer decision path.

In contrast to the original lock-free insertion [LHL14], we employ two enhancements to improve performance and memory usage. We will elaborate on these improvements in the following paragraphs.

For all potentially transparent materials, the opacity might depend on external information, for example, originating from some input parameter or a texture. This information is not known at shader compile-time and might also vary at run-time. As a consequence, the graphics driver makes some assumptions about shader execution, e. g. register usage or enabling/disabling rasterization optimizations. In particular, write operations to global GPU memory in a fragment shader disable the hardware's early-Z test. This behavior is caused solely by the presence of these operations in the shader assembly, even if they are never called. The early-Z test could be explicitly enforced. In this case, all per-fragment tests (depth, stencil, occlusion queries) would be performed not after, but prior to fragment-shader execution, and the corresponding buffers would be updated accordingly. However, this is not applicable for our approach as the geometry pass uses discard operations to prevent writing transparent fragments to the G-buffer. Thus, performing the depth test for those fragments before shader execution would corrupt the depth buffer. However, an early termination of occluded fragments is highly desirable. Therefore, we bind the current depth buffer and perform manual conservative Z-culling. Our results, which are presented in Section 5, show that this workaround is quite effective in rejecting occluded fragments.

Furthermore, the lock-free insertion is capable of discarding fragments that are considered almost hidden. For each pixel, the algorithm stores a depth-sorted list of fragments. While inserting a new fragment, the list needs to be traversed to find the correct place of insertion. During this traversal, the resulting opacity is accumulated and if it exceeds a predefined threshold, the current, as well as all further fragments, are considered hidden. We modified the algorithm in such a way that, if it fails to insert a fragment due to its accumulated opacity, instead of just discarding, it is written to the depth buffer, as indicated in Figure 6. As a result, newly generated fragments are culled by our manual Z-culling prior to the insertion if they fall behind the current depth. This also prevents memory allocation for those fragments and thereby decreases the algorithm's memory footprint.

4.3. Light-Culling Pass

Once the geometry is rasterized into the G- and A-buffer, we need to gather light information in order to shade the fragments. A straightforward implementation on top of the deferred shading pipeline may be inefficient in terms of scalability with an increasing number of light sources. The reason is that, for light accumulation, the proxy geometries of the light sources are rasterized, and for each affected pixel the lighting contribution is typically accumulated in the G-buffer. This is sufficient for conventional deferred shading, but inefficient for transparent fragments as they are stored in per-pixel linked lists and their frequent traversal would cause many un-coalesced memory accesses.

In order to resolve this issue, we exploit the idea of deferred tile-based shading [OA11]. In this approach, the non-relevant light sources are culled per pixel or, respectively, per screen-space tile in a separate pass. Thus, lighting computations are deferred to the shading pass. Therefore, the frame buffer is covered with a screen-space grid (light grid) with a fixed tile size (see Figure 7a). Then the lights whose volumes intersect the tile's frustum are stored within a tile. Subsequently, every pixel is shaded for all lights assigned to the corresponding tile. While the tile-based shading approach

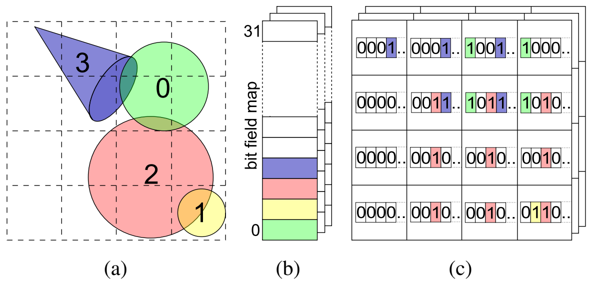


Figure 7: The light grid is populated by rendering the light proxies (a) into a multi-layered texture (c) in which each bit corresponds to a light. The mapping is stored in a bit-field light map (b).

was initially designed for shading opaque data, it is also quite beneficial for the transparent fragments stored in the A-buffer. It resolves the aforementioned inefficiencies during light accumulation. In the compositing pass, the corresponding per-pixel linked lists are only traversed once, performing fragment shading and compositing on-the-fly.

There are various ways to generate the light grid. In our system, the grid is represented by a multi-layered 2D texture in which the two-dimensional coordinates address grid cells. Each texture layer corresponds to a bit field, as shown in Figure 7b. If a bit is set, the corresponding light has a contribution to at least one of the tile's fragments. Therefore, each texture stores the information for up to 32 lights, as shown in Figure 7c. If the number of lights is higher than 32, multiple texture layers are used. However, of course, not all scene lights, but only those visible for the current view, are enumerated in the bit field.

The grid is populated by rasterizing the light volumes and setting the corresponding bit for each light fragment using the atomic OR-operation. The resolution of the viewport is set to the grid resolution, so tile dimensions become equal to one pixel. However, traditional rasterization evaluates the coverage only at the pixel center. We ensure proper light assignments by enabling conservative rasterization. This type of rasterization generates fragments for every pixel if it at least partially overlapped by a primitive. For hardware which does not support this extension, there are fallback solutions based on either geometry shaders [HAMO05], multi-sampling or full-screen rendering, as illustrated in Figure 8.

4.4. Shade-Compositing Pass

In this full-screen pass, the fragments stored in the G-buffer and the A-buffer are shaded and blended into the final image. Compositing is performed from front-to-back by iterating the A-buffer and accumulating the pixel's color. We continue until the the depth of the current fragment is greater than the depth stored in the G-buffer or there are no more transparent fragments left to shade. After that, the current

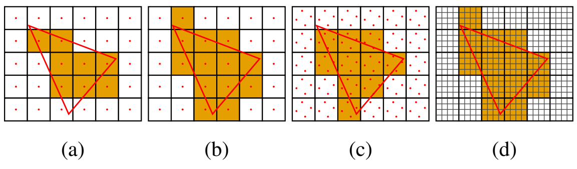


Figure 8: In contrast to traditional rasterization (a), conservative rasterization (b) generates fragments also for partially covered tiles. If this feature is not available, either multisampling (c) or fullscreen (d) fallbacks may be used.

pixel color can be blended with the shaded result of the G-buffer content.

Since tile-based shading is used, each fragment needs to be shaded for all light sources affecting the corresponding tile. For that, we loop over all bits in the tile's bit field and perform shading only for those lights whose bit is set. This procedure is identical for both G-buffer and A-buffer fragments. This way of shading has the following advantages. The data necessary to shade a fragment is loaded only once. Common terms in the rendering equation can be factored out which is beneficial as we employ a computationally expensive physically-based shading approach [KG13]. Furthermore, fragments within the same tile have coalesced access to light information.

4.5. Post-processing Pass

After shading and compositing, additional screen-space effects are applied to the shaded image through a set of optional post-processing passes. Their configuration and order of execution is defined by the programmer. They have access to all intermediate buffers (light grid, G-buffer and A-buffer) which enables a variety of sophisticated rendering effects.

5. Results and Discussion

All tests were performed on a 3.33 GHz Intel Core i7 workstation with 12GiB RAM equipped with a single NVIDIA GeForce GTX 980 GPU with 4GiB video memory and using a rendering resolution of 1024x1024.

For the implementation of the A-buffer, we considered various PPLL techniques. In general, all methods capable of gathering incoming fragments could be used. However, the choice of algorithm affects the pipeline design, as well as the possibilities for optimizations. In particular, early-termination based on the pixel's saturation is only possible for approaches which sort the fragments on-the-fly. We refer to these approaches as *pre-sort* techniques, while we refer to approaches with a separate sorting pass as *post-sort*. Furthermore, some methods benefit from recent hardware advancements more than others. Therefore, we compared various state-of-the-art PPLL-implementations in order to find the best choice on the most recent graphics hardware. Our comparison includes three base techniques and four variations of them:

- **PreSortLF** – A lock-free insertion sort based on 64-bit atomic operations [LHL14].
- **PreSortLF*** – PreSortLF with early termination.
- **PreSortLFMerge2*** – Similar to PreSortLF*, but using two PPLLs to reduce insertion costs. The two lists are merged during compositing.
- **PreSortCS** – An insertion sort using a critical section, similar to [VF14].
- **PreSortCS*** – PreSortCS with early termination.
- **PostSort** – A PPLL-implementation as described by Yang et al. [YHGT10]. After gathering, insertion sort is performed in fixed-sized local arrays.
- **PostMerge16** – Similar to PostSort, but not limited by a fixed-sized array. Instead, it performs multi-way merge sort [KLZ13] with a chunk size of 16.

In order to evaluate the performance of each PPLL technique, the following three scenes have been used: the Dragons, the Atrium Sponza, and the Hairball. Figure 9 shows the views and the corresponding complexity of our test scenes. In the applied material description, a constant opacity value of 50 % was set for all fragments while the saturation threshold was set to 98 %. However, a direct comparison of the timings of the subtasks is not possible because, in some cases, they are inseparable. Nevertheless, we measured the timings for the two major stages. For pre-sort techniques, the 1st stage contains the gathering and sorting of fragments while the 2nd stage performs shading and blending. For post-sort techniques, the 1st stage simply gathers while the 2nd stage sorts, shades and blends the fragments. The performance results are summarized in Table 1.

The results show that pre-sorting based on lock-free insertion and early termination performs best for all our scenes. The performance of post-sorting techniques suffers from the lack of early termination. In addition, we noticed that all lock-free approaches benefit from the highly improved support for atomic operations of most recent hardware. On previous hardware generations, we found the results were mixed and different approaches performed best depending on the scene's complexity.

Nevertheless, there are some limitations in our system. The memory requirements and the rendering performance are both affected by the order of geometry submission, the window resolution and applied material descriptions.

The memory budget reserved for storing non-opaque fragments needs to be set in advance because there is no dynamic memory allocation in shader programs. This is not a specific limitation of our system, but common to all PPLL implementations. In particular, the minimum storage requirements for the lock-free insertion [LHL14] consist of the preallocation of all head pointers and the storage for the respective fragment information. This represents a space-time

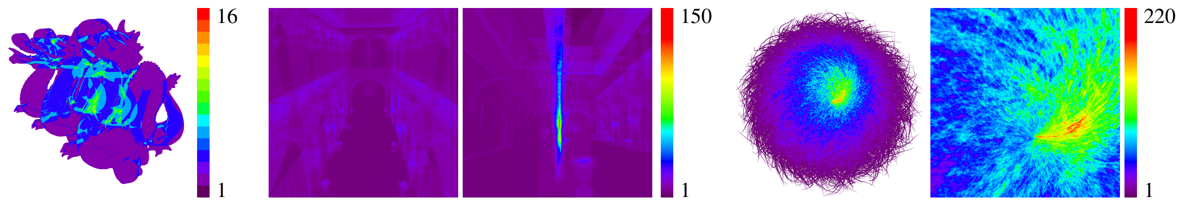


Figure 9: Depth complexity heat-maps of the three scenes: the Dragons scene, the Atrium Sponza, and the Hairball. The bar on the right shows the colors associated with the number of fragments per pixel.

Table 1: Rendering time in milliseconds for the A-buffer techniques.

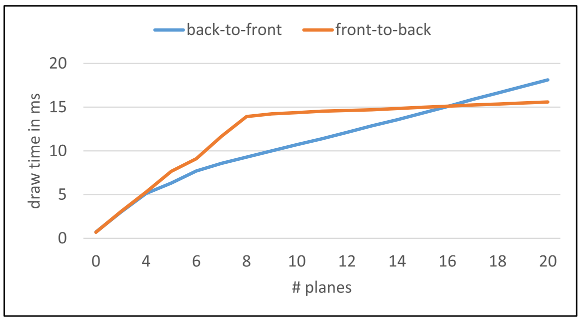
Scene	Dragons			Sponza, view 1			Sponza, view 2			Hairball, view 1			Hairball, view 2		
Fragments	1 772 682			7 940 179			7 964 342			15 398 997			70 591 231		
Stage	1st	2nd	total	1st	2nd	total	1st	2nd	total	1st	2nd	total	1st	2nd	total
PreSortLF	2.5	0.8	3.3	5.3	1.4	6.7	6.3	1.3	7.6	87.9	12.0	99.9	555.5	1.5	557.0
PreSortLF*	2.3	0.7	3.0	5.2	1.4	6.6	5.2	1.3	6.5	24.2	15.4	39.6	56.2	1.7	58.0
PreSortLFMerge2*	2.5	0.8	3.3	5.7	1.5	7.3	6.1	1.4	7.5	39.2	18.1	57.3	106.2	2.0	108.2
PreSortCS	5.6	0.9	6.6	12.6	3.4	16.0	21.9	2.8	24.7	213.8	17.2	231.0	1030.4	2.4	1032.7
PreSortCS*	5.6	0.9	6.5	12.1	3.2	15.3	12.7	2.9	15.6	29.7	12.4	42.2	71.0	2.2	73.3
PostSort	3.8	2.0	5.8	12.0	7.0	19.0	12.1	15.7	27.8	25.2	139.6	164.8	107.6	672.7	780.4
PostMerge16	3.8	1.4	5.2	12.0	5.6	17.6	12.1	7.3	19.4	25.2	65.1	90.2	107.6	312.2	419.8

tradeoff, but it also implies an overhead if there are no transparencies present. The necessary memory mainly depends on the amount of transparencies in the scene, the window resolution and the data stored for each fragment. For example, in our system, 48 bytes are stored for each fragment and 8 bytes are used for each head pointer which results in a minimum budget of about 60MB for a resolution of 1024x1024. In our tests, we set the budget to 1GB which was sufficient for all our models and allows us to store an average number of 18 transparent fragments per-pixel. However, higher resolutions and depth complexities may require more memory. If the reserved budget is not sufficient, artifacts may occur. Therefore, an adaptive memory management is highly desirable, but remains for future consideration.

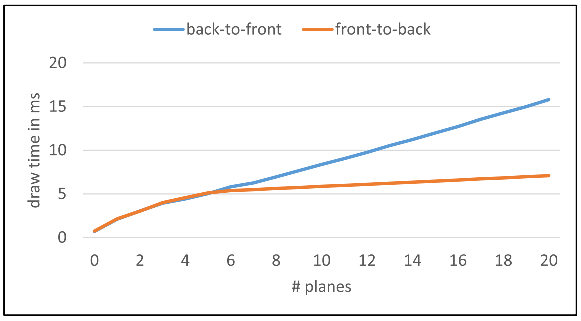
In our system, occluded fragments can be discarded based on their depth or the pixel's accumulated saturation. However, the efficiency of this optimization directly depends on the order of incoming fragments. If the scene is rendered back-to-front, all fragments are inserted at the head of the PPLL. The insertion at the head of the list is efficient, but it does not allow for an early discard of occluded fragments. In this case, the storage requirements are higher compared to front-to-back rendering.

In addition, the processing of occluded fragments decreases rendering performance. Figure 10 shows our test results for analyzing this effect. In this example, semi-transparent full-screen planes are rendered in ascending depth order and vice versa. For a low depth complexity of less than 5 fragments per pixel, both approaches perform almost equally. For higher depth complexity, the draw times for front-to-back rendering increase almost quadratically because the insertion of a fragment requires traversing the list of all fragments gathered for this pixel. However, once the pixel's opacity is saturated, the depth is written to the G-buffer and all further fragments can be discarded. In contrast, the draw times for rendering back-to-front increase linearly, but no discarding is possible. Consequently, the required memory budget and the performance of our current implementation does not only depend on the current view and the object's materials, but also on the order of geometry submission. As a remedy, insertion costs for front-to-back rendering could be reduced by adapting the lock-free insertion sort.

Furthermore, we measured the performance overhead if no transparent objects were in the scene. In a first test, we analyzed the material descriptions to disable the A-buffer initialization and to simplify the shade-compositing pass if all materials were opaque. In addition, the A-buffer insertion code was automatically removed by the shader optimization. As a result, there was no performance overhead at all. However, for some materials, the transparency calculations may also result in opaque fragments. Therefore, we performed a second test in which the shader optimization was avoided by explicitly setting the opacity to 100 % via input parameter. The measured draw times indicate that the performance drops about by 4 %. This overhead is caused by the buffer initialization and the check for transparent fragments during



(a) Opacity = 50%.



(b) Opacity = 80%.

Figure 10: This Figure shows the draw times for rendering semi-transparent full-screen planes in different order of submission. For both tests, the pixel's saturation threshold was set to 99%. In back-to-front order (blue), draw times increase almost linearly because the fragments are inserted efficiently at the front of the A-buffer. In contrast, insertion costs for rendering front-to-back (orange) first increase almost quadratically but, if saturation is reached, all further fragments can be discarded. For higher opacity (b), this threshold is reached earlier than for lower opacity (a).

compositing. This constant overhead only depends on the viewport resolution, not the geometry or depth complexity.

In our system, anti-aliasing is achieved as a post-process using FXAA [CR12] which has no additional memory requirements and a very low performance overhead. If a higher visual quality is desired, multisampling with a corresponding G-buffer could also be considered. In addition, the A-buffer would need to be extended with a 4-byte sample mask for each transparent fragment, which would increase the memory requirements by about 10%. During compositing, all fragments would need to be blended based on their transparency and corresponding sample mask.

6. Conclusion and Future Work

In this paper, we presented an efficient integration of order-independent transparency into a programmable deferred shading pipeline. Our pipeline concept is easily extensible in terms of additional passes, geometry representations and user-defined materials. Transparency is a property of our material description which is programmable at vertex and fragment level, thereby, giving the user maximum flexibility to manipulate the opacity values on a per-fragment basis. The material description and the designated shader programs for different geometry representations are merged using meta-programming techniques. During rendering, only the transparent fragments are routed into the A-buffer, which is based on lock-free insertion. All opaque fragments are submitted to the G-buffer. The light information is gathered in a multi-texture bit-field. Gathering this information in a separate pass allows for an efficient shading and blending of all transparent and opaque fragments.

In addition, we evaluated and compared various state-of-the-art A-buffer implementations to find the most efficient on latest graphics hardware. Based on the results, we redesigned the deferred shading pipeline of the open-source rendering framework guacamole to add support for order-independent transparency. The flexibility of programmable opacity increases the realism and also improves usability and spatial perception by enabling adaptive fade-outs or see-through lenses. The overhead of our system is output-sensitive and minimal if only opaque objects are present.

For future work, we plan to improve the dependency between rendering performance and order of geometry submission. By extending the pre-sorting implementation with back-pointer semantics, we could greatly reduce insertion costs for front-to-back rendering without losing the benefit of early termination. Furthermore, the proposed light culling approach has some limitations concerning how lights are assigned to a tile. The tile frustum intersection test is achieved implicitly by rasterization, which might create a bottleneck in the scenes with very large number of lights. Therefore, we would like to investigate depth-aware methods suggested in [Har12,OBA12] for use in conjunction with order-independent transparency.

7. Acknowledgement

This work was supported by the German Federal Ministry of Education and Research (BMBF) under grant 03IPT704X (project Big Data Analytics).

References

[AKK*11] ARGELAGUET F., KULIK A., KUNERT A., ANDUJAR C., FROEHLICH B.: See-through techniques for referential awareness in collaborative virtual reality. *International Journal of Human Computer Studies 69* (2011), 387–400. 5

[AMHH08] AKENINE-MÖLLER T., HAINES E., HOFFMAN N.: *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008. 2

[BGO09] BAYRAKTAR S., GÜDÜKBAY U., ÖZGÜÇ B.: GPU-Based Neighbor-Search Algorithm for Particle Simulations. *Journal of Graphics, GPU, and Game Tools 14*, 1 (Jan. 2009), 31–42. 2

[BKB13] BAUER F., KNUTH M., BENDER J.: Screen-Space Ambient Occlusion Using A-Buffer Techniques. In *2013 International Conference on Computer-Aided Design and Computer Graphics* (Nov. 2013), IEEE, pp. 140–147. 1

[BKKF13] BECK S., KUNERT A., KULIK A., FROEHLICH B.: Immersive group-to-group telepresence. *Visualization and Computer Graphics, IEEE Transactions on 19*, 4 (April 2013), 616–625. 4

[BM08] BAVOIL L., MYERS K.: *Order Independent Transparency with Dual Depth Peeling*. Tech. rep., NVIDIA Corporation, 2008. 3

[Car84] CARPENTER L.: The A-buffer, an antialiased hidden surface method. *ACM SIGGRAPH Computer Graphics 18*, 3 (July 1984), 103–108. 1

[Cat74] CATMULL E. E.: *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, The University of Utah, 1974. 1

[CFM*12] CARNECKY R., FUCHS R., MEHL S., JANG Y., PEIKERT R.: Smart transparency for illustrative visualization of complex flow surfaces. *IEEE transactions on visualization and computer graphics 19*, 5 (May 2012), 838–51. 2

[CR12] COZZI P., RICCIO C.: *OpenGL Insights*. CRC Press, July 2012. `http://www.openglinsights.com/`. 9

[ESSL11] ENDERTON E., SINTORN E., SHIRLEY P., LUEBKE D.: Stochastic transparency. *Visualization and Computer Graphics, IEEE Transactions on 17*, 8 (2011), 1036–1047. 3

[Eve01] EVERITT C.: *Interactive Order-Independent Transparency*. Tech. Rep. 6, NVIDIA Corporation, 2001. 3

[HAMO05] HASSELGREN J., AKENINE-MÖLLER T., OHLSSON L.: Conservative rasterization. *GPU Gems 2* (2005), 677–690. 6

[Har12] HARADA T.: A 2.5D culling for Forward+. In *SIGGRAPH Asia 2012 Technical Briefs on - SA '12* (New York, USA, 2012), ACM Press, pp. 18:1–18:4. 9

[JÏ4] JÄHNE S.: *Using per-pixel linked lists for transparency effects in remote-rendering*. PhD thesis, Universität Stuttgart, 2014. 3

[JH08] JANG H., HAN J.: Fast collision detection using the A-buffer. *The Visual Computer 24*, 7-9 (May 2008), 659–667. 2

[KG13] KARIS B., GAMES E.: Siggraph '13: Acm siggraph 2013 courses: Real shading in unreal engine 4, 2013. 3, 7

[KKB*11] KULIK A., KUNERT A., BECK S., REICHEL R., BLACH R., ZINK A., FROEHLICH B.: C1x6: A Sterepscopic Six-User Display for Co-located Collaboration in Shared Virtual Environments. *Proceedings of the 2011 SIGGRAPH Asia Conference on - SA '11 30* (2011), 1. 5

[KLZ13] KNOWLES P., LEACH G., ZAMBETTA F.: Backwards Memory Allocation and Improved OIT. *Pacific Conference on Computer Graphics and Applications - Short Papers* (2013), 59–64. 7

[LHL14] LEFEBVRE S., HORNUS S., LASRAM A.: Per-Pixel Lists for Single Pass A-Buffer. In *GPU Pro 5: Advanced Rendering Techniques*, Engel W., (Ed.). CRC Press, 2014, pp. 3–23. 3, 5, 7

[MB13] MCGUIRE M., BAVOIL L.: Weighted Blended Order-Independent Transparency. *Journal of Computer Graphics Techniques (JCGT) 2*, 2 (Dec. 2013), 122–141. 3

[MCTB11] MAULE M., COMBA J. A. L. D., TORCHELSEN R., BASTOS R.: A survey of raster-based transparency techniques. *Computers & Graphics 35*, 6 (Dec. 2011), 1023–1034. 3

[MCTB13] MAULE M., COMBA J. A., TORCHELSEN R., BASTOS R.: Hybrid transparency. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2013), I3D '13, ACM, pp. 103–118. 3

[OA11] OLSSON O., ASSARSSON U.: Tiled Shading. *Journal of Graphics, GPU, and Game Tools 15*, 4 (Nov. 2011), 235–251. 4, 6

[OBA12] OLSSON O., BILLETER M., ASSARSSON U.: Clustered Deferred and Forward Shading. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics* (Aire-la-Ville, Switzerland, Switzerland, 2012), EGGH-HPG'12, Eurographics Association, pp. 87–96. 9

[Oli12] OLIVER P.: Unreal engine 4 elemental. In *ACM SIGGRAPH 2012 Computer Animation Festival* (New York, NY, USA, 2012), SIGGRAPH '12, ACM, pp. 86–86. 2

[PD84] PORTER T., DUFF T.: Compositing digital images. *ACM SIGGRAPH Computer Graphics 18*, 3 (July 1984), 253–259. 2

[RBA08] ROŻEN T., BORYCZKO K., ALDA W.: GPU bucket sort algorithm with applications to nearest-neighbour search. *Journal of the 16th Int. Conf. in Central Europe on Computer Graphics, Visualization and Computer Vision* (2008), 161–168. 2

[RFV13] ROSSIGNAC J., FUDOS I., VASILAKIS A.: Direct rendering of Boolean combinations of self-trimmed surfaces. In *CAD Computer Aided Design* (2013), vol. 45, pp. 288–300. 2

[SF09] SCHOLLMEYER A., FRÖHLICH B.: Direct trimming of nurbs surfaces on the gpu. *ACM Trans. Graph. 28*, 3 (July 2009), 47:1–47:9. 3

[SLB*14] SCHNEEGANS S., LAUER F., BERNSTEIN A.-C., SCHOLLMEYER A., FROEHLICH B.: guacamole - An Extensible Scene Graph and Rendering Framework Based on Deferred Shading. In *Proceedings of the 7th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS '14) in conjunction with IEEE Virtual Reality* (Minneapolis, Minnesota, USA, 2014). 2, 3

[ST90] SAITO T., TAKAHASHI T.: Comprehensible rendering of 3-D shapes, 1990. 2

[VF14] VASILAKIS A. A., FUDOS I.: k + -buffer. In *Proceedings of the 18th meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games - I3D '14* (New York, USA, 2014), ACM Press, pp. 143–150. 7

[YHGT10] YANG J. C., HENSLEY J., GRÜN H., THIBIEROZ N.: Real-Time Concurrent Linked List Construction on the GPU. *Computer Graphics Forum 29*, 4 (Aug. 2010), 1297–1304. 3, 7

[YWY10] YU X., WANG R., YU J.: Real-time Depth of Field Rendering via Dynamic Light Field Generation and Filtering. *Computer Graphics Forum* (2010). 1