Comparison of (order-independent) transparency algorithms with osgTT

Bachelor Thesis

Christoph Blümel
Degree programme: BSc Computer Science

2015-11-06

Supervisor: Jun.-Prof. Dr. Mario Hlawitschka
Scientific Visualization Group
Abstract
Christoph Blümel

Comparison of (order-independent) transparency algorithms with osgTT
This thesis documents the evaluation of several transparency techniques in aspects of quality and performance. Depth sorted alpha blending and the order-independent transparency techniques additive blending, multiplicative blending, unsorted alpha blending and depth peeling are examined. The theoretical concepts of these techniques are explained. In this work, the transparency library osgTT is extended with additive and multiplicative blending and integrated into the visualization software FAnToM. Many test cases are investigated for the quality comparison. The performance benchmarks are conducted with three different scenes.
## Contents

1 Introduction .................................................. 1  
   1.1 Motivation ............................................. 1  
   1.2 Problem description .................................... 1  
   1.3 Structure of thesis ..................................... 2  

2 Background .................................................. 3  
   2.1 The OpenGL rendering pipeline ......................... 3  
   2.2 Blending .................................................. 5  
   2.3 Alpha blending ......................................... 6  
   2.4 Order-independent transparency ......................... 8  
      2.4.1 Additive blending .................................. 8  
      2.4.2 Multiplicative blending ........................... 8  
      2.4.3 Depth peeling ..................................... 9  

3 osgTT .......................................................... 13  
   3.1 OpenSceneGraph ........................................ 13  
   3.2 Description .............................................. 14  
   3.3 Extension ............................................... 15  
   3.4 Integration into FAnToM ............................... 17  

4 Quality ...................................................... 19  
   4.1 General comparison ..................................... 19  
   4.2 Influence of alpha value ................................ 23  
   4.3 Shader ................................................... 25  
   4.4 Intersecting geometries ................................. 27  
   4.5 Cyclically overlapping geometries ..................... 30  
   4.6 Self-overlapping geometries ........................... 33  
      4.6.1 Two-sided geometries .............................. 33  
      4.6.2 Limitations of depth sorting ...................... 33  
   4.7 Depth peeling ............................................ 36  
      4.7.1 Number of passes .................................. 36  
      4.7.2 Limitations ........................................ 36  
   4.8 Further limitations of depth sorted mode .......... 40
## Contents

5 Performance .......................... 41  
   5.1 Method ............................ 41  
   5.2 Distribution of frame times .......... 43  
   5.3 Average frame times ................ 45  
   5.4 Discussion ......................... 46  

6 Conclusion ............................ 51  

7 Future work ............................ 53  

Bibliography ............................. 55  

List of Figures .......................... 59  

List of Tables ........................... 61  

Listings ................................. 63  

Acknowledgments ........................ 65
CHAPTER 1

Introduction

In everyday life transparent objects are a regular phenomenon, for example when one sees through glass. In computer graphics, transparency effects are important to create realistic scenes. Technically, a transparent object is fully translucent and thus not visible, unless refraction or reflection of light occurs. More precise terms to describe the effect are semi-transparent, partially transparent and translucent. In the following, these terms and transparent are used interchangeably.

FAnToM [31, 1] is a scientific visualization system which was initiated by Prof. Scheuermann in 1999. The name is an acronym for Field Analysis using Topological Methods. It is used for flow visualization research with applications in engineering disciplines, like fluid mechanics. In recent years, a redesign of FAnToM has been conducted. Algorithms to process data can be implemented as plugins, so-called toolboxes. There are two types of algorithms: data algorithms and visualization algorithms. Typically, data algorithms process scalar, vector or tensor fields. The output can be used as input for further data algorithms or visualized with a visualization algorithm. This pipeline structure is represented in the user interface as a data flow network graph.

1.1 Motivation

When a data set is visualized, it can happen that several surfaces overlap. In order to see the hidden surfaces and to have a better understanding of the data, it is necessary to employ a transparency effect. Currently, FAnToM is able to render transparent surfaces with alpha blending and primitive-based depth sorting. Unfortunately, this approach has problems with certain spatial arrangements and can be quite costly. Therefore, it is interesting to include other transparency techniques in FAnToM. The transparency library osgTT is a fitting start for this undertaking as it already supports three transparency techniques and uses the same rendering back end as FAnToM, namely OpenSceneGraph. Furthermore, two additional transparency algorithms shall be examined whether they are suitable: additive and multiplicative blending.

1.2 Problem description

The aim of this thesis is to integrate osgTT into FAnToM and to extend the library with additive and multiplicative blending. Moreover, all transparency algorithms need to be evaluated in terms of quality and performance. For the integration solutions have to be found how to
merge osgTT with FAnToM’s back end, how toolboxes can make use of osgTT, and how to provide transparency settings in the user interface. Additive and multiplicative blending need to be implemented in osgTT’s existing structure. For the quality comparison scenes must be created which bring out the visual differences and flaws of the transparency techniques. The performance evaluation requires a reliable way of measuring frame times and scenes which produce an appropriate amount of load.

1.3 Structure of thesis
The next chapter deals with the theoretical background of transparency techniques in OpenGL. Problems stemming from OpenGL’s design and several solutions are described. The third chapter gives attention to osgTT, its extension and the integration into FAnToM. Chapters four and five constitute the evaluation. The former focuses on quality whereas the latter concentrates on performance. In the end, a conclusion is formulated and possible topics for future work are mentioned.
CHAPTER 2

Background

In this chapter, problems which occur when drawing translucent geometries and solutions to them are described. First, an overview of the OpenGL pipeline is given. Secondly, blending is covered. Then, different transparency algorithms are presented, beginning with an order-dependent one and proceeding with order-independent algorithms.

2.1 The OpenGL rendering pipeline

OpenGL does not support rendering translucent primitives directly [14]. To illustrate this, I give an overview of the OpenGL 4 rendering pipeline as described in [23, 24, pp. 10-14]. Figure 2.1 depicts the different pipeline stages.

In the first stage, vertex specification [29], an ordered list of vertices is submitted to the graphics card. The component format of the vertices is defined; several floating-point and integer types are possible. Furthermore, it is specified how this list has to be interpreted, for example as lines or triangles.

In the second stage, the vertex shader, each individual vertex is processed by a user defined program. This program can have multiple outputs and the vertex is usually transformed from object to clip space.

The following tessellation stage [25] is optional. Here, geometry can be tessellated which increases the number of primitives, thus yielding better-looking models. Tessellation consists of three stages of which the first and the last are programmable. First, the tessellation control shader determines how much tessellation to apply. Then, the fixed-function tessellation process subdivides patches of vertices based on the tessellation control shader’s outputs. Lastly, the tessellation evaluation shader computes the vertex position for each newly-generated vertex, much like a vertex shader.

The next stage is the programmable and optional geometry shader. It can remove primitives, generate completely new ones or alter vertex values.

Vertex post-processing consists of several steps: ‘Transform Feedback is the process of capturing Primitives generated by the [v]ertex [p]rocessing step(s), recording data from those primitives into [b]uffer [o]bjects. This allows one to preserve the post-transform rendering state of an object and resubmit this data multiple times’ [27]. Another step is clipping where primitives outside of the viewing volume are discarded and primitives both inside and outside
are divided in a way that only the part inside remains. Furthermore, in vertex post-processing the vertices are transformed from clip space to window space.

In the following stage, primitive assembly, the vertex data from the previous stages is composed into a sequence of primitives. A limited form of primitive assembly is executed before tessellation or geometry shaders if they are active. Next, face culling is performed where triangles facing away from the viewer are usually discarded because they are occluded in closed surfaces.

In the subsequent stage, rasterization, primitives are rasterized into fragments which can be used to compute the final data for a pixel.

The pixel shader is the penultimate stage and programmable. Generally, lighting is calculated and the colour of a fragment is determined here. It is also possible to employ texture mapping or discard a fragment.

The per-fragment (or per-sample) operations compose the last stage. It consists of several tests which discard fragments if the test fails. After these tests additional operations take place. First, the pixel ownership test determines whether OpenGL is allowed to write to the pixel associated with the fragment. This is not allowed if another window overlaps the OpenGL window in that pixel. Secondly, fragments pass the scissor test only if they lie within a designated rectangular area. Thirdly, the stencil test compares the fragment with the associated location in the stencil buffer. Most typically, this is used to prevent drawing inside or outside an irregularly shaped region. Fourthly, the depth test compares the fragment’s depth with the according value in the depth buffer. If the test passes, the fragment’s depth value is written to the depth buffer. In general, this is used for hidden-surface elimination by drawing only the foremost fragments.

After these tests, blending occurs where the fragment’s colour can be combined with the colour in the framebuffer at that location via predefined equations. Instead of blending, logical operations can be performed. Here, the fragment’s colour and the colour in the framebuffer are combined via bitwise operations. With the write mask, writes to the depth and stencil buffers or to individual colour channels in the framebuffer can be disabled.

Eventually, the pixel data in the framebuffer is displayed on the screen.

As described, none of these stages provide functionality for simple and direct transparency. With naive rendering, only the fragment which was processed last at each pixel’s position is visible. These fragments correspond to the primitives which were processed last. If the depth test is enabled, the foremost fragments are visible. When translucent objects overlap, we see a combination of the objects’ colours. In the OpenGL pipeline it is possible to mix colours via blending or the fragment shader. In the next sections, I describe blending in detail and how it can be utilized for transparency effects.
2.2 Blending

Without blending [24, pp. 166-171] an incoming fragment would overwrite the pixel colour in the framebuffer. With blending it is possible to combine the fragment’s colour with the colour in the framebuffer. The result of this combination is written to the framebuffer. The way of this combination can be defined in two kinds: the blend equation and blending factors. The colour of the incoming fragment is called source colour and the pixel colour in the frame buffer is called destination colour. The blend equation is the basic mathematical operation with which the source and destination colour are combined. Both colours are scaled by one factor each before the equation is applied: the source blending factor and the destination blending factor.

The blend equation can be set with glBlendEquation. It is possible to set different equations for the RGB components and the alpha component via glBlendEquationSeparate. Table 2.1 shows the blending equations provided by OpenGL and the corresponding identifiers with which the equation can be set via the aforementioned functions. There are equations to compute the addition, difference, minimum, or maximum of the source and destination colour components. $C_s$ and $C_d$ denote the source and destination colours. $S$ and $D$ represent the source and destination blending factors.

These blending factors can be set with glBlendFunc or with glBlendFuncSeparate for the RGB components and the alpha component separately. Table 2.2 displays possible blending factors. Subscripts $s$ and $d$ denote the source and destination colour components, respectively. Some factors scale the source or destination colour components by one, zero, source alpha, destination alpha, the according source colour channels, or destination colour channels. Furthermore, there are factors which scale by the above-mentioned values subtracted from one. Moreover, it is possible to define a constant colour for blending. The appropriate function to specify this colour is glBlendColor. In the table, the constant colour is represented by the subscript $c$. Additionally, the fragment shader can output a second source colour for blending. This is called dual-source blending [24, pp. 198-200]. The usage of this second source colour is denoted by the subscript $s_1$ in the table. The next sections describe how to use blending for transparency.

<table>
<thead>
<tr>
<th>Blending equation identifier</th>
<th>Mathematical operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_FUNC_ADD</td>
<td>$C_s S + C_d D$</td>
</tr>
<tr>
<td>GL_FUNC_SUBTRACT</td>
<td>$C_s S - C_d D$</td>
</tr>
<tr>
<td>GL_FUNC_REVERSE_SUBTRACT</td>
<td>$C_d D - C_s S$</td>
</tr>
<tr>
<td>GL_MIN</td>
<td>$\min(C_s S, C_d D)$</td>
</tr>
<tr>
<td>GL_MAX</td>
<td>$\max(C_s S, C_d D)$</td>
</tr>
</tbody>
</table>
2.3 Alpha blending

The most common technique to render translucent objects is alpha blending (described in [13, pp. 199-202, 19]). When a translucent objects overlaps an opaque one, one sees a mixture of the colours of both objects. The intensity of the colour in the front depends on the opacity of the object in front. The colour of the object behind is attenuated to some degree compared to the case when we see the opaque object only. This attenuation equals the opacity of the object in front. In computer graphics the opacity of a colour is represented by the alpha value. A fully opaque colour has an alpha value of 1 whereas complete transparency is represented by 0. Values in between give an effect of translucency. Thus, we see the colour of the object behind only by the alpha value of the object in front subtracted from one. The following equation illustrates this combination and is used for alpha blending.

\[ \text{blend}_\alpha(C_s, C_d, \alpha_s) = C_s \times \alpha_s + C_d \times (1 - \alpha_s) \]  

(2.1)
2.3 Alpha blending

Equation (2.1) computes the resulting colour by adding the colour $C_s$ of the incoming fragment (the translucent object) weighted by its opacity $\alpha_s$ to the colour in the framebuffer (the object behind) weighted by $1 - \alpha_s$. This can be realized in OpenGL by using the blending equation `GL_FUNC_ADD`. Additionally, specifying `GL_SRC_ALPHA` as the source blend factor and `GL_ONE_MINUS_SRC_ALPHA` as the destination blend factor is necessary. Figure 2.2 illustrates alpha blending: A translucent sphere is blended with an image of clouds in the sky. The result is that the sky behind the sphere is attenuated realistically.

The alpha blending equation can be rewritten as eq. (2.3). The second term contains a subtraction. Since subtraction is not commutative, alpha blending is not commutative. This means that it is important to blend fragments from back to front in order to get correct results.

\[
\text{blend}_\alpha(C_s, C_d, \alpha_s) = C_s \cdot \alpha_s + C_d - C_d \cdot \alpha_s \tag{2.2}
\]

\[
= C_d + \alpha_s \cdot (C_s - C_d) \tag{2.3}
\]

In order to blend fragments in the right order, primitives have to be submitted to the graphics card in back-to-front order. This so-called depth sorting [13, pp. 204-205, 28] has to be done on the CPU and can be quite costly. If opaque primitives are drawn first and translucent ones second, it suffices to sort translucent geometry only. Problems can occur with sorting: Intersecting primitives have to be split at their intersections (fig. 2.3(a)). Cyclic overlaps can not be sorted as the polygons are in the front and in the back at the same time (fig. 2.3(b)). They would have to be split for perfect sorting, too. But splitting of primitives increases cost. Even if no problematic arrangements like the above mentioned exist, it can be difficult to sort correctly: In fig. 2.3(c) A represents the viewer. B and C are translucent polygons. The correct order is to draw C after B. If depth sorting were done by the distance of the centre of each polygon to the viewer, B would be mistakenly drawn last as its centre is closer to the viewer. This incorrect order persists if sorting were done by the nearest or farthest vertex instead of the centre. Often, a simplified depth sorting is done by sorting objects via their bounding boxes only. This is faster, but overlapping primitives of the object can be blended in the wrong order or only the foremost fragments are drawn if the depth test is enabled, thus omitting fragments behind.

**Figure 2.2:** Alpha blending (chequered areas indicate translucency). Source: adapted from [3]
2.4 Order-independent transparency

As alpha blending with depth sorting is costly and problematic under certain conditions, order-independent transparency (OIT) techniques have been an area of active research. ‘Order-independent transparency is a technique where blending operations are carried out in a manner such that rasterization order is not important’ [24, p. 609]. This research has yielded approximative blending formulas which are commutative. Other approaches sort fragments via the GPU in contrast to primitives in depth sorting. In the next sections I describe two commutative blending formulas and one OIT-technique which sorts fragments.

2.4.1 Additive blending

Additive blending [19] simply adds the source and destination colours component-wise without any scaling. As eq. (2.4) shows, this can be implemented in OpenGL by setting the blend equation GL_FUNC_ADD and specifying GL_ONE for both blend factors. GL_ONE leaves all components unchanged by multiplying them simply with one. The alpha value which is the real indicator of the translucency of a fragment is ignored. Furthermore, as the maximum value of a colour component is one, blending multiple fragments reaches this limit quickly. Then, additional fragments cannot influence this colour channel any more. Consequently, additive blending produces only an approximative result.

\[
blend_{add}(C_s, C_d) = C_s \times 1 + C_d \times 1 = C_s + C_d
\]  (2.4)

Figure 2.4 displays an example of additive blending. As no scaling is applied, there is no attenuation in colour. The colours of the sky and the sphere are simply added which produces a bright combination. Although the output in this example is not as realistic as with alpha blending, this ‘type of blending is often used for particle effects, where each particle might be a spark or other, small lighted point. It can also be used to simulate flames’ [19].

2.4.2 Multiplicative blending

With multiplicative blending [19] source and destination colour are multiplied component-wise. This can be achieved with OpenGL in the following way: GL_ZERO is set as the source blend factor. This eliminates the first term of the blending equation GL_FUNC_ADD. Specifying GL_SRC_COLOR which represents the source colour as the destination blend factor results in
2.4 Order-independent transparency

The multiplication of source and destination colour as depicted in eq. (2.5). Again, the alpha value has no influence which leads to an approximation of realistic blending.

\[
\text{blend}_{\text{mult}}(C_s, C_d) = C_s \times 0 + C_d \times C_s = C_d \times C_s
\]  

(2.5)

Figure 2.5 gives an example of multiplicative blending. Here, the sky behind the sphere is attenuated. As the values in the colour components are fractions, multiplying them with another colour results in even smaller fractions. This mimics the natural attenuation of objects behind translucent materials. Moreover, the area around the sphere is completely black in the blended image. The reason for this is that the area around the sphere is filled with fully transparent black in the original image of the sphere. If the area around the sphere is blended with the sky, the colour components of the sky are multiplied with zero (as the colour black has the value zero in all of its colour channels), producing zero again and thus black.

2.4.3 Depth peeling

Depth peeling is a multi-pass fragment-level depth sorting technique for order-independent transparency described in [8]. In depth peeling the scene is separated into consecutive layers of surfaces one behind the other. One pass over the scene is necessary to peel a single layer. Compositing these layers produces the final image.

Figure 2.6 gives a diagrammatic view of depth peeling. The images depict the peeling of successive layers. As the images show, each layer consists of multiple depths; more precisely,
one depth per fragment. Evidently, the peeling process happens at the fragment level. The first layer is the nearest surface of the scene. The second layer constitutes the second nearest surface, and so on. The idea of a second or nth nearest surface is rather unintuitive. Figure 2.7 illustrates this notion with a red teapot on a blue ground plane. The interior of the teapot is green. This colouring helps to distinguish the different layers. Layer 0 represents the nearest surface. Layer 1 is mostly in green which marks the teapot’s inside. Layer 2 consists mostly of a shape of a teapot in blue. This means that the fragments of this layer are from the ground plane. Layer 3 only consists of few and small parts.

The standard depth test gives us the nearest fragments, and thus the nearest surface. Unfortunately, it does not provide a way to determine the nth nearest surface. Depth peeling solves this problem. With n passes, we can get n layers deep into the scene. Each layer consists of depth and colour information. Depth peeling is a straightforward multi-pass algorithm. In the first pass we render regularly which gives the nearest surface with the standard depth test. In the second pass the depth buffer from the first pass is used to ‘peel away’ fragments with depths that are less than or equal to the nearest depths from the first pass. Then, only the fragments behind the nearest surface remain. Now, we can generate the second nearest surface and use its depth buffer to peel away the first and second nearest surfaces in the third pass. There is one obstacle with this algorithm: It needs two depth tests per pass, but OpenGL provides only one.

To better explain the process of depth peeling, the pseudocode in listing 2.1 uses two depth units. \( A \) and \( B \) represent depth buffers. They are switched with every pass since the depth buffer from the previous pass is used in the current. Depth unit 0 is used to discard previously nearest fragments, that is previously nearest surfaces from former passes. This is not necessary in the first pass. In this case depth unit 0 is disabled. Depth unit 1 employs regular depth testing. After removing fragments processed in previous passes via depth unit 0, depth unit 1 computes the depth buffer for the surface of the current pass. The depth buffer of unit 1 in pass \( i \) is used as ‘peeling’ depth buffer for unit 0 in pass \( i + 1 \) with which unit 0 can discard the nearest fragments of pass \( i \) and all previous passes. It is important to disable depth writes on unit 0. Otherwise, the buffer would be updated with depth values of fragments passing the test of unit 0. This destroys the depth values of the nearest surface of the previous pass in the depth buffer which are needed for depth unit 0 to function properly. Eventually, the rendered scene is saved as layer \( i \). To compute the final image, the layers are composited via alpha blending (section 2.3) in back-to-front order (e.g. via viewport-sized textured quads).

As OpenGL provides only one depth unit, a different implementation for one of the units is required. Depth unit 0 can be implemented easily via a fragment shader which discards fragments with greater depth value than in the depth buffer of unit 1 in the previous pass. The contents of unit 1’s depth buffer can be passed to the shader via a depth texture. Alternatively, it is possible to realize depth unit 0 by clever usage of shadow mapping [8]. With this approach, depth peeling can be implemented with the fixed-function OpenGL pipeline. The standard depth test is used for depth unit 1.

For completely correct results, it is necessary to compute additional passes until there are no transparent fragments. In reality, as fragments farther back have diminished effect on the final image, it suffices to truncate the number of passes for a reasonable and efficient approximation.
2.4 Order-independent transparency

Figure 2.6: Depth peeling strips away depth layers with each successive pass. The frames show the frontmost (leftmost) surfaces as bold black lines, hidden surfaces as thin black lines, and ‘peeled away’ surfaces as light grey lines. Source: [8]

Figure 2.7: These images illustrate the layers of depth peeling from the nearest surface to the fourth nearest surface. Source: [8]
for (i=0; i<num_passes; i++)
{
    clear color buffer
    A = i % 2
    B = (i+1) % 2
    depth unit 0:
    if(i == 0)
        disable depth test
    else
        enable depth test
    bind buffer A
    disable depth writes
    set depth func to GREATER
    depth unit 1:
    bind buffer B
    clear depth buffer
    enable depth writes
    enable depth test
    set depth func to LESS
    render scene
    save color buffer RGBA as layer i
}
This chapter deals with the transparency library osgTT. First, I give an overview of OpenSceneGraph which is used by both FAnToM and osgTT. This is followed by a description of osgTT. After that, I illustrate how I extended osgTT with additive and multiplicative blending. Finally, the integration of osgTT into FAnToM is depicted.

3.1 OpenSceneGraph

OpenSceneGraph (OSG) is a rendering middleware written in C++ and based on OpenGL. It reduces the complexity of the OpenGL low-level API by providing a higher-level abstraction. OSG provides modularity and object-orientation for OpenGL concepts like graphics primitives and materials. The basis for OSG’s retained rendering system is the theory of scene graphs. Such graphs collect rendering commands and data for executing them later. This is in contrast to immediate rendering systems where function calls have direct and instantaneous effect. By collecting commands, OSG is able to perform optimizations, e.g. reordering of commands. In general a scene graph is a hierarchical graph which encapsulates the spatial and logical relationships of a graphical scene. Usually, a tree is built, but nodes can also have multiple parents in OSG. At the top, a root-level node is located. At the bottom, leaf nodes represent the bottom layer of the tree. In between, there are group nodes which can have an arbitrary number of children. With group nodes, children can be treated as one. Furthermore, group nodes propagate their information and effects of operations to their children.

In 1998, Don Burns initiated the development of OpenSceneGraph. In the following year, Robert Osfield took over the project and made it open source. OSG is licensed under the OpenSceneGraph Public License (OSGPL) which is based on the GNU Lesser General Public License (LGPL). The year 2007 saw the release of OSG 2.0 with improved multi-core and multi-GPU support. OpenGL 3, OpenGL 4 and OpenGL ES support was introduced with OSG 3.0 in 2011. OSG is available for many platforms including Microsoft Windows, Linux, FreeBSD, Android, and Mac OS X. Class documentation can be found at [15]. Unfortunately, descriptions are not very extensive or lack completely.

OpenSceneGraph consists of four core libraries and several additional libraries known as NodeKits. Moreover, it is extensible via plugins. The core comprises the OpenThreads, osg, osgDB and osgUtil libraries. OpenThreads provides an object-oriented interface for threads. The osg library includes basic elements, such as nodes, geometries or textures. It
also contains mathematics classes for matrix operations. Reading and writing files as well as stream input/output operations are handled by the osgDB library. The osgUtil library helps in building the rendering backend, e.g. tree traversal and culling. An important NodeKit library is osgViewer. It provides viewer-related classes which integrate the scene graph with a wide variety of windowing systems. The osgFX NodeKit provides special effects and helps in implementing new ones. There are NodeKits for shadow techniques, particle effects, text, and volume rendering, among other things.

3.2 Description

osgTT (OSG Transparency Toolkit or Tool) [20, 12] is a transparency library for OpenSceneGraph developed by the American company AlphaPixel, LLC. It is open source via the MIT License and supports three transparency techniques: alpha blending with depth sorting and two unsorted methods, namely, depth peeling and ‘delayed blend’. These modes are implemented using only the fixed-function OpenGL pipeline; no shaders are used. AlphaPixel’s website indicates that the toolkit is to be extended with two single-pass techniques, A-buffer and weighted average, in the future, but there have not been any changes to the repository since January 2014. osgTT is comprised of two main classes: TransparencyGroup and DepthPeeling. A few other classes exist which provide demo applications.

The class TransparencyGroup inherits from OSG’s group node and mimics its behaviour. Its children are applied with the currently set transparency mode. In this class all modes, except depth peeling, are implemented. Children can be added with the function addChild( osg::Node* child, bool transparent, bool twoSided ) whose last two arguments are not existing with a regular group node. The first argument determines the subgraph which should be added. The second argument specifies whether transparency should be applied to the subgraph. If a subgraph with translucent geometry is added and this argument is set to false, the geometry is rendered as opaque. Except with depth peeling, here, this setting is ignored. With the last argument it is possible to indicate that back faces should be drawn. The active transparency technique can be defined with setTransparencyMode( TransparencyMode mode ). Its argument is an element of the enumeration TransparencyMode which provides constants for the supported transparency techniques and one to disable transparency. There are also methods to get the current transparency mode and to set a different instance of the DepthPeeling class.

In general, the transparency modes are implemented by setting an osg::StateSet when a child is added. As the two boolean parameters of osgTT’s addChild method give four different choices, there are four state sets: A pair of state sets for transparent children and a pair for opaque ones. The state sets in each pair differ in whether back faces are culled. The two state sets for two-sided children have face culling disabled.

The depth sorted mode employs regular depth testing and uses alpha blending. Depth sorting is realized by using OSG’s transparent render bin. Geometries in this bin are sorted by the centre of their bounding boxes and are rendered back-to-front [16].

[12, TransparencyGroup.h] describes delayed blend as follows: ‘Transparent objects are rendered using multiplicative alpha blending’. The word ‘multiplicative’ is misleading as regular alpha blending is applied but without depth sorting. Depth buffer updates are disabled in this mode.

Depth peeling is handled by a DepthPeeling object. According to a comment in [12,
DepthPeeling.h], the code was adapted from an example available at [21]. It is important to set the number of passes, the texture unit to be used, and the size of the viewport. The viewport size determines the size of the texture for the peeling layers. The multi-pass algorithm is implemented by using osg::Camera objects which render to textures. For each pass, one camera is created which renders one peeling layer. This implementation utilizes shadow mapping as depth test. In the end, another camera composites the layers as textured quads via alpha blending.

### 3.3 Extension

The aim is to extend osgTT with additive and multiplicative blending. The source code of osgTT with the new modes is located at [9, branch devs/mam09btk/osgtt] (integrated into FAnToM). The two new modes are similar to delayed blend which is already supported. I could use the implementation of delayed blend as a basis for the new modes. The biggest difference among these modes is the blending operation. Blending factors can be defined in OSG with the osg::BlendFunc class. I added two new BlendFunc objects to the TransparencyGroup:

```cpp
_blendFuncAdd = new osg::BlendFunc( GL_ONE, GL_ONE );
_blendFuncMult = new osg::BlendFunc( GL_ZERO, GL_SRC_COLOR );
```

The BlendFunc constructor takes two arguments. The first one is the source blending factor and the second is the destination blending factor. The additive blending object _blendFuncAdd uses GL_ONE for both arguments as described in section 2.4.1. The multiplicative blending object _blendFuncMult is constructed with GL_ZERO as source factor and GL_SRC_COLOR as destination factor (see section 2.4.2). Both modes use the blending equation GL_FUNC_ADD. This equation is the default, thus it need not be specified explicitly.

The transparency modes are implemented in the function setTransparencyMode( TransparencyMode mode ). Originally, there were several if-else clauses which tested which mode is specified via the argument `mode`. I changed this to a more elegant switch statement. The enumeration TransparencyMode was extended with two constants for the new modes: ADDITIVE_BLEND and MULTIPLICATIVE_BLEND.

Listings 3.1 and 3.2 display the implementation of additive and multiplicative blending. Depth writes are disabled so that not only the foremost fragments are blended. The appropriate blending factors are set in lines 12 and 18.

```cpp
switch( mode )
{
[...]

case ADDITIVE_BLEND:
{
    osg::ref_ptr<osg::Depth> depth = new osg::Depth;
    depth->setWriteMask( false );
    _transparentState->setAttributeAndModes( depth.get(), osg::StateAttribute::ON );
    
    _transparentState->setRenderingHint( osg::StateSet::DEFAULT_BIN );
    _transparentState->setRenderBinDetails( 12, "RenderBin" );
```
Listing 3.1: Source code for additive blending

switch( mode )
{
    [...]
    case MULTIPLICATIVE_BLEND:
    {
        osg::ref_ptr<osg::Depth> depth = new osg::Depth;
        depth->setWriteMask( false );
        _transparentState->setAttributeAndModes( depth.get( ), osg::StateAttribute::ON );
        _transparentState->setRenderingHint( osg::StateSet::DEFAULT_BIN );
        _transparentState->setRenderBinDetails( 12, "RenderBin" );
        _transparentState->setAttributeAndModes( _blendFuncMult.get( ), osg::StateAttribute::ON );
        Group::addChild( _scene.get( ) );
    }
    break;
    [...]
3.4 Integration into FAnToM

In order to integrate osgTT into FAnToM, changes to the back end had to be made and new GUI elements had to be added to control the transparency mode and the number of passes. The source code of FAnToM with integrated osgTT can be found at [9, branch devs/mam09btk/osgtt]. Two approaches were explored while designing the GUI elements, as the first turned out to be flawed. At first, I created a submenu for transparency in the preferences top menu. Here, it was possible to set the transparency mode. Unfortunately, one could not see which mode is currently active. The same problem would have arisen for the number of passes in depth peeling. Therefore, I began a second approach where I created a separate view for transparency, the Transparency View. This view contains a combo box to choose the transparency mode and a slider to specify the number of passes. With this second approach it is always possible to see the currently active transparency settings. Figure 3.1 displays the new GUI elements.

In the back end, osgTT’s TransparencyGroup had to be integrated in FAnToM’s scene graph. Originally, there was a scene node below the clipping node. Nodes of algorithms were added as children to this scene node. Now, a TransparencyGroup is in place of this scene node. This TransparencyGroup has four child nodes. These four nodes correspond to the parameter combinations of the addChild method. Therefore, there is a transparent, an opaque, a transparent/two-sided, and an opaque/two-sided node. Nodes of algorithms are supposed to be added to one of these nodes depending on their transparency settings.

If the transparency settings of a node are changed, it has to be moved to the correct node of the four nodes. I realized this by adding a new class OsgTransparentGraphics which is derived from OsgGraphics. OsgTransparentGraphics has two methods to set whether it is transparent and whether it is two-sided. An OsgGraphics object adds or removes its nodes from the scene graph whether it is set to be visible. The node under which it should insert its nodes is supplied at construction time. Originally, this was the scene node which was mentioned earlier. The constructor of OsgTransparentGraphics has arguments for the four nodes which represent the transparency settings. As an OsgTransparentGraphics object knows these four nodes, it can add its nodes to the right node when its transparency settings are changed. To handle this I implemented the method OsgTransparentGraphics::updateParent() which determines which of the four nodes is the correct new parent node. To set the new parent, the method setParent( osg::ref_ptr< osg::Group > newParent ) was added to OsgGraphics. It moves the nodes of this OsgGraphics object from the old parent to the new parent. Now, algorithms can use OsgTransparentGraphics to alter transparency settings.

I adapted the show surface algorithm of the grid toolbox to use the two transparency options of OsgTransparentGraphics. This results in two new checkboxes, Enable transparency and Set two-sided, in the options view of this algorithm. Now, it is possible to enable transparency and two-sidedness for the rendered surface. The altered show surface source code is available at [10, branch devs/mam09btk/osgtt].

The interface GraphicsEngine was extended with methods to specify and retrieve the transparency mode options for osgTT. These methods are implemented in OsgGraphicsEngine. With setTransparency( Transparency mode ) it is possible to set the transparency mode. The type Transparency is an abstraction of the TransparencyMode enumeration of osgTT. The method
setTransparencyPasses( unsigned int numPasses ) specifies the number of passes for multi-pass modes, here depth peeling. Both methods are used by the transparency view.

osgTT’s DepthPeeling class needs the size of the viewport to work properly. Therefore, the DepthPeeling instance has to be informed when FAnToM’s main graphics window is resized. This was implemented with a resize callback which sets the viewport size in the DepthPeeling object. The GraphicsPainter class was extended with functionality to register callbacks for resize events. Now, a resize callback can be added with the addResizeCallback method.
In this chapter the transparency algorithms are evaluated by their difference in image quality and realism. The number of passes used for depth peeling is given in brackets. Fixed-function lighting was used for the comparisons except in section 4.3. A light was inserted above the camera. Its position is updated to stay above the camera when the camera perspective is changed. The source code for this mechanism is available at [9, branch devs/mam09btk/ffpLight].

4.1 General comparison

Figure 4.1 gives a general overview of the different transparency modes. In the front, the scene uses the Stanford dragon [26] coloured in cyan. Behind, there is a Utah teapot [6] in magenta. In the back, a yellow Stanford bunny [26] is placed. Both the dragon and the teapot have an alpha value of 0.5 whereas the bunny is completely opaque.

Figure 4.1(a) displays the scene with transparency disabled. Thus, only the dragon and a small part of the teapot are visible.

Figures 4.1(b) and 4.1(c) compare additive blending with a black and a white background. With a white background most of the image is white because the white of the background is added to the colour of transparent geometries in front of it. Since the bunny is not transparent, it is not blended with the background. Thus, its yellow colour is visible with additively blended colours of the models in front of it. A black background does not influence the colour of transparent objects with additive blending. Consequently, a lot more of the scene is recognizable. Here, the colour of overlapping, transparent objects becomes brighter. The middle of the image where all three models overlap is almost completely white. In general, the more transparent, overlapping objects exist, the brighter the image becomes as the colours are added and get closer to the maximum of white.

Figures 4.1(h) and 4.1(i) compare multiplicative blending with the background in white or black, respectively. This mode has some similarities to additive blending. With a black background most of the image is black because the black of the background is multiplied by the colours of transparent geometries in front of it. Since black has the value zero in all colour channels the result of the multiplication is zero in all channels and thus black overall. As the bunny is not transparent, it is not blended with the background and we can see it blended with the dragon and the teapot. With a white background this flaw of multiplicative blending does not occur as white with the value one in all channels does not influence multiplicative blending.
Where transparent objects overlap, the colour becomes darker because the values in the colour channels are fractions and give even smaller fractions if multiplied with other colours. As a consequence, if in the blending series of a pixel at least one fragment has a colour channel with value zero, the channel is zero for the result colour of the pixel. This happens with the bunny as it becomes green because it ‘loses’ its red channel since the red channel of the cyan dragon equals zero. Other examples are the teapot which becomes blue and the middle of the image which appears in black.

As seen, for both additive and multiplicative blending it is important to choose the colour of the background judiciously since it can influence the quality of the image greatly.

Depth peeling (fig. 4.1(f)) and depth sorting (fig. 4.1(g)) produce the most realistic images with depth peeling being even better. In the depth sorted image some artefacts are visible with the teapot: First, the edge where the lid meets the body and the handle should be more opaque and magenta. Second, a bit of the sprout where it joins the body is cut off. These cases are due to the enabled depth test which causes discardment of important fragments if not handled in the right order (see section 4.6.2 for a detailed test case).

Figure 4.1(d) shows delayed blending. The order in which the geometries are drawn depends on their (coincidental) order in the scene graph. In this case the teapot is rendered erroneously last as if it were in front of the dragon. In fig. 4.1(e) the objects were manually ordered, but this should be regarded as an exception. Here, the rendered image looks as good as with depth peeling. In order to create comparable images of the unordered delayed mode, the order with which the teapot is rendered last was reproduced manually for the following test cases.
4.1 General comparison

(a) None

(b) Additive (white background)

(c) Additive (black background)

(d) Delayed
(e) Delayed (manually ordered)  
(f) Depth peeling (6)  
(g) Depth sorted  
(h) Multiplicative (white background)
4.2 Influence of alpha value

This section examines how the transparency algorithms handle a different alpha value. The scene is the same as in the general comparison (section 4.1) except that the alpha value is with 0.3 lower.

Delayed blending, depth peeling and depth sorted transparency (figs. 4.2(c) to 4.2(e)) compute the alpha value correctly. That is, the dragon and teapot are more translucent. The images of additive and multiplicative blending (figs. 4.2(b) and 4.2(f)) look exactly the same as the ones with higher alpha value in figs. 4.1(c) and 4.1(h). This is not surprising as both algorithms only use the colour channels in their blending functions and thus ignore the alpha value.
(a) None
(b) Additive
(c) Delayed
(d) Depth peeling (6)
4.3 Shader

In this comparison it is determined whether shaders work with the transparency modes. The scene of the general comparison (section 4.1) is used again, but this time lighting is calculated by a simple Blinn-Phong shader provided by FAnToM. Figure 4.3 displays the scene rendered with the shader. One difference between the shader and fixed-function lighting is that the bunny is brighter.

All algorithms work correctly with the shader except depth peeling. In fig. 4.3(d) one can see that there is no transparency at all. osgTT’s depth peeling does not work with shaders. As depth peeling is implemented via the fixed-function pipeline in osgTT, there seems to be a problem with the programmable pipeline.
(a) None

(b) Additive

(c) Delayed

(d) Depth peeling (6)
4.4 Intersecting geometries

This section describes how the algorithms cope with intersecting objects. The scene contains one Stanford dragon and a box [7] which intersects it in its middle. Both objects are translucent.

In this test case depth sorted transparency is interesting because OSG’s depth sorting via bounding boxes cannot sort the objects correctly as none is completely in front of or behind the other. Depending on the camera angle the image is rendered as if the box were behind (fig. 4.4(e)) or in front of (fig. 4.4(f)) the dragon. In the first case, the middle part of the dragon is missing due to the enabled depth test. In the second case, the part of the box overlapped by the dragon is missing for the same reason.

Delayed mode with its coincidental order renders the whole box as if it were in front of the dragon (fig. 4.4(c)). Depth peeling renders the scene perfectly (fig. 4.4(d)). Additive (fig. 4.4(b)) and multiplicative (fig. 4.4(g)) modes are not affected by intersecting objects.
(a) None

(b) Additive

(c) Delayed

(d) Depth peeling (4)
4.4 Intersecting geometries

Figure 4.4: The dragon and the box intersect

(e) Depth sorted

(f) Depth sorted with a different camera angle

(g) Multiplicative
4.5 Cyclically overlapping geometries

The scene for this test case contains four triangles which cyclically overlap. This means that the yellow triangle overlaps the blue one, the blue one overlaps the red triangle which overlaps the green one and the green triangle overlaps the yellow one reaching a cycle in this way.

This arrangement is interesting for depth sorted transparency for the same reason as described in section 4.4. But in this case the problem occurs without intersecting objects. As pictured in figs. 4.5(e) and 4.5(f) it depends on the camera angle again which overlapped corners are visible. In the first case the green and the red corner are visible and rendered in the correct order, but the blue and yellow ones are not drawn because of the depth test. In the second case it is the other way round. With further camera angles other combinations of corners are visible.

Delayed blending (fig. 4.5(c)) draws only the blue corner correctly in this case of coincidence. The other corners are drawn as if they were in the front but, in reality, they are in the back. Depth peeling (fig. 4.5(d)) has no problems with this scene and renders it flawlessly. Additive (fig. 4.5(b)) and multiplicative (fig. 4.5(g)) blending produce typical results without problems stemming from the cyclic overlaps.
4.5 Cyclically overlapping geometries

(a) None

(b) Additive

(c) Delayed

(d) Depth peeling (2)
Figure 4.5: The triangles overlap cyclically
4.6 Self-overlapping geometries

In this section the transparency modes are tested with self-overlapping geometries. First, osgTT’s two-sided option is examined which can create self-overlapping geometries. Second, a limitation of the depth sorted mode is described in detail.

4.6.1 Two-sided geometries

Here, osgTT’s two-sided option is tested with the transparency modes. Enabling two-sidedness effects that back faces are drawn, too. Consequently, geometries are self-overlapping as they overlap their back faces. The scene from the general comparison (section 4.1) is used but with two-sidedness enabled for the dragon and the teapot. These two models are partly self-overlapping without two-sidedness. With the option enabled, they are completely self-overlapping and have twice as many layers.

Comparing two-sided geometries in fig. 4.6 with one-sided ones in fig. 4.1, the following differences are apparent. Additive mode is brighter and less details are recognizable as the additional layers add more colour (fig. 4.6(a)). Multiplicative mode is darker as the additional layers reduce the final colour of a pixel (fig. 4.6(e)). Furthermore, a part of the dragon’s crest (spikes) on his back of his middle body part is more noticeable now. This makes the dragon’s middle body look as if it were curved differently. Delayed, depth peeling and depth sorted modes (figs. 4.6(b) to 4.6(d)) are more opaque because of the additional layers’ colours. They are not as bright since the white of the background is much less significant with the additional layers. Depth peeling is rendered with eight passes which is not enough to incorporate all layers, but additional passes do not yield recognizable differences. In depth sorted mode, the teapot has more artefacts than without two-sidedness (fig. 4.1(g)). The reason is the same as stated in section 4.1: The depth test discards important fragments.

4.6.2 Limitations of depth sorting

In fig. 4.7 the camera is being tilted over a transparent dragon in depth sorted mode. At first, the self-overlapping body parts are visible (fig. 4.7(a)). With progressing tilt, they gradually disappear (figs. 4.7(b) and 4.7(c)). Eventually, only the foremost parts of the dragon are visible as if it were not translucent.

This behaviour occurs with self-overlapping geometries if the polygons of the object are not sorted by depth which does not happen with OSG’s depth sorting. As a consequence self-overlapped parts of a geometry are only visible in osgTT’s depth sorted mode if the polygons are processed back to front. If they are not processed in the right order, the depth test discards fragments of self-overlapped parts if a fragment in front of them has already been processed. Depending on the camera position the order of polygons is coincidentally adequate or yields flawed images. The orientation of the dragon in the general comparison (fig. 4.1(g)) is very fitting for osgTT’s depth sorted mode. But if the dragon were rotated by 180 degrees around its y-axis or with the camera angles in fig. 4.7, only pieces of its self-overlapped body parts are drawn correctly.
(a) Additive

(b) Delayed

(c) Depth peeling (8)

(d) Depth sorted
4.6 Self-overlapping geometries

Figure 4.6: Two-sided geometries (back faces visible)

(a) Multiplicative

Figure 4.7: Tilting the camera in depth sorted mode

(b)
4.7 Depth peeling

In this section characteristics unique to depth peeling are described. In contrast to the other modes, depth peeling is a multi-pass technique. Hence, the effect of the number of passes on the quality of the rendered image is examined. Furthermore, a bug in osgTT with depth peeling is described.

4.7.1 Number of passes

In this section the influence of the number of passes with depth peeling is described. Figure 4.8 displays the scene from the general comparison (section 4.1) from zero to six passes with depth peeling. Six passes are enough to incorporate all layers of this scene.

Zero passes only produce an image of the background because the geometries are not processed at all (fig. 4.8(a)). The first pass shows only the dragon and a part of the teapot as if transparency was disabled, but both models are blended with the white of the background (fig. 4.8(b)). With two passes the dragon is transparent and the bunny as well as the remaining part of the teapot appear, but the teapot itself is not translucent, though (fig. 4.8(c)). Not until the scene is rendered with three passes does the teapot become transparent (fig. 4.8(d)). The next three passes give improvements where all three models overlap and self-overlapping of the dragon and the teapot occurs (figs. 4.8(e) to 4.8(g)). The differences of the last two passes are virtually not recognizable without magnification.

4.7.2 Limitations

There is a bug with depth peeling which manifests itself when zooming or changing the camera angle. Figure 4.9 shows this bug when zooming. If you zoom in or out the scene is eventually cut off in the back or the front, respectively. In fig. 4.9(a) the bunny is not completely rendered and in fig. 4.9(c) parts of the dragon are cut off. It is possible to work around this by changing to a different transparency mode and then changing back to depth peeling. This was done in figs. 4.9(b) and 4.9(d). Via changing the camera angle, one can trigger this bug, too.

The authors of osgTT possibly hint at this bug in the source code:

I believe there is currently a bug which manifests itself when a user switches dynamically between the various TransparencyModes that causes OSG to miscalculate the near/far values for the topmost Camera. [12, see TransparencyGroup.h]
4.7 Depth peeling

(a) 0 passes

(b) 1 pass

(c) 2 passes

(d) 3 passes
Figure 4.8: Effect of the number of passes with depth peeling
Figure 4.9: Bug when zooming with depth peeling
4.8 Further limitations of depth sorted mode

Due to depth sorting via bounding boxes, it is possible that the depth sorted mode creates flawed images even if there are no problematic arrangements of geometries like intersections or cyclic overlaps. Figure 4.10 illustrates this: Depending on the camera position the half-moon is fully drawn or a part is missing.

Figure 4.10: Problems with depth sorting via bounding boxes
CHAPTER 5

Performance

In this chapter the performance of the transparency algorithms is evaluated. For multi-pass modes (depth peeling), the number of passes is given in brackets.

5.1 Method

For measuring the performance of the algorithms, three different scenes were created. They are called **wide scene**, **deep scene** and **deep, occluded scene**. Each scene contains the Stanford Dragon model [26] ten times. In the wide scene the models are arranged in a wide and horizontal way with a bit of overlap. In the deep scene the ten dragons overlap very much and minimal horizontal shift is used. The deep, occluded scene is virtually identical to the deep scene, except that a completely opaque, white triangle is placed in front of the models. Figures 5.4 to 5.6 show these three scenes.

The Stanford Dragon consists of 1,132,830 triangles and 566,098 vertices [26]. So, each scene comprises about 11 million triangles. The colour of the models changes from $RGB = (0,1,1)$ (cyan) to $RGB = (1,0,1)$ (magenta) from front to back. This colour gradient is realized by increasing the $R$ component and decreasing the $G$ component by 0.1 with every model further.

```cpp
void OsgView::paint()
{
    [...] // ... draw code
    timing.start();
    osgViewer->frame();
    glFinish();
    timing.pause();
    [...] // ... draw code
}
```

**Listing 5.1:** Part of the paint method with frame time measurement
behind. The alpha channel of every dragon equals 0.5.

The deep scene is intended to surface performance differences with many overlapping triangles compared to the wide scene. The idea in placing an opaque triangle in front of the scene in the deep, occluded scene is the following: In the rendered image only the white triangle is visible. Therefore, the ten dragon models behind it are irrelevant for the final image and could be discarded from the rendering process. With the deep, occluded scene we can measure whether a transparency algorithm makes use of this saving.

The test system uses an Intel Core i7-980 processor, which has six cores and a clock rate of 3.33 GHz [4]. Moreover, it is equipped with 24 GiB of main memory and an Nvidia GeForce GTX 580 [11] with 3072 MiB of video memory.

Listing 5.1 displays the code which was used to measure the frame times. At first, the time measurement is started with a timing object (5). Then, the actual OSG rendering method is called (7) [17]. Calling glFinish() in line 8 guarantees that rendering is complete [24, p. 31]. Finally, the time measurement is stopped (10).

In order to create a representative set of frame times, 100 values were measured. These values were obtained by calling the paint method 100 times consecutively. The source code used for the benchmarks can be found at [9, branch devs/mam09btk/bench].

![Figure 5.1: Distribution of frame times of the wide scene](image-url)
5.2 Distribution of frame times

Figures 5.1 to 5.3 display the distribution of all frame times via box plots. In the whole series of benchmarks there are only two outliers, one with the wide scene (16.048 ms) and one with the deep scene (15.640 ms), both with transparency disabled. All other values differ by less than 1 ms, referring to one set of frame times. This shows that the measurement method is reliable and the transparency algorithms provide consistent frame times. As a consequence the box plots are degenerate and resemble mere strokes. Moreover, one can see a linear increase with depth peeling in the figures.

Figure 5.2: Distribution of frame times of the deep scene
Figure 5.3: Distribution of frame times of the deep, occluded scene
5.3 Average frame times

Table 5.1 displays the average frame times. The mean values of single pass modes are very similar and vary in a range of 10 to 11 ms. No transparency is virtually as fast as single-pass transparency. Depth peeling with one pass is about 0.2 ms slower than the other single pass modes. The frame time of depth peeling increases almost linearly with the number of passes: With multi-pass depth peeling the increase per pass is a bit more than the frame time of single-pass depth peeling. It is more than 12 ms with the wide scene and more than 11 ms with the other scenes, respectively. In general, the wide scene is a bit slower than the other scenes: Here, single pass modes are approximately 0.4 ms slower. There is practically no performance difference between the deep and the deep, occluded scene.

<table>
<thead>
<tr>
<th>Transparency mode</th>
<th>Frame time [ms] of scene</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>wide</td>
</tr>
<tr>
<td>None</td>
<td>10.613</td>
</tr>
<tr>
<td>Additive</td>
<td>10.597</td>
</tr>
<tr>
<td>Delayed</td>
<td>10.597</td>
</tr>
<tr>
<td>Depth sorted</td>
<td>10.648</td>
</tr>
<tr>
<td>Multiplicative</td>
<td>10.538</td>
</tr>
<tr>
<td>Depth peeling (1)</td>
<td>10.776</td>
</tr>
<tr>
<td>Depth peeling (2)</td>
<td>23.223</td>
</tr>
<tr>
<td>Depth peeling (3)</td>
<td>35.655</td>
</tr>
<tr>
<td>Depth peeling (4)</td>
<td>48.068</td>
</tr>
<tr>
<td>Depth peeling (5)</td>
<td>60.451</td>
</tr>
<tr>
<td>Depth peeling (6)</td>
<td>72.792</td>
</tr>
<tr>
<td>Depth peeling (7)</td>
<td>85.388</td>
</tr>
<tr>
<td>Depth peeling (8)</td>
<td>97.821</td>
</tr>
<tr>
<td>Depth peeling (9)</td>
<td>110.195</td>
</tr>
<tr>
<td>Depth peeling (10)</td>
<td>122.238</td>
</tr>
</tbody>
</table>
5.4 Discussion

Only depth sorting and depth peeling include additional steps compared to the other modes. Moreover, in each mode the complete geometry has to be processed at least once. Thus, very similar frame times among single-pass modes, even disabled transparency, are logical. The minute differences among them are explained by coincidental load on the test system.

Depth sorting doesn’t impact performance in the test cases. According to [16], OSG uses bounding boxes for depth sorting and sorts by the middle of the box. Furthermore, it doesn’t sort inside a geometry. Sorting the ten dragons by their bounding boxes only seems to be too fast to impact performance significantly.

Single-pass depth peeling tends to be a little bit slower than the other single-pass modes. Here, a slight overhead seems to occur. Plus, this mode does not provide any transparency. Depth peeling performance decreases linearly with the number of passes as the algorithm does not reduce the amount of geometry per pass. Consequently, the whole geometry has to be processed in every pass and the linear increase results. The increase in frame time in multi-pass depth peeling is more than the frame time of single-pass depth peeling. Here, additional steps of the algorithm seem to impact performance. One of these steps is to composite the images of the passes into the final image.

The performance of the deep and deep, occluded scene is virtually identical. The slower values of the wide scene are result of coincidental load, rather than the scene itself because the amount of geometry is the same as in the deep scene and no mode is sophisticated enough to take advantage of the arrangement of the scene. Consequently, the triangle in the deep, occluded scene has no effect. One could think that the depth test could save work in this case by discarding fragments behind the triangle. But the depth test is positioned at the very end of the graphics pipeline in the per-sample operations [24, p. 13] after the expensive steps of vertex processing and shading.
5.4 Discussion

(a) None

(b) Additive

(c) Delayed
Figure 5.4: Wide scene in different transparency modes

- (d) Depth peeling (4)
- (e) Depth sorted
- (f) Multiplicative
5.4 Discussion

(a) None

(b) Additive

(c) Delayed

(d) Depth peeling (5)
Figure 5.5: Deep scene in different transparency modes

Figure 5.6: Deep occluded scene: only a white triangle is visible
CHAPTER 6

Conclusion

The purpose of this thesis was to bring new transparency techniques to FAnToM and evaluate them. This was done with the integration of osgTT. A suitable implementation in FAnToM’s back end was found with the alteration of the scene graph and the addition of OsgTransparentGraphics. The ShowSurface algorithm was successfully adapted. Moreover, I created a fitting way to expose the transparency settings in the user interface with the Transparency View. Additive and multiplicative blending were added to osgTT and work as expected.

Several test cases for the quality comparison were conducted. Depth peeling, depth sorted mode and multiplicative mode provide satisfactory quality. Depth peeling produces flawless results, unfortunately osgTT’s implementation does not work with shaders. The depth sorted mode has problems with certain spatial arrangements and sometimes self-overlapping parts of a geometry are not drawn. Its sorting with bounding boxes is a good compromise between speed and quality. The multiplicative mode gives a sufficient approximation in most cases and does not require sorting. But it may not be used if a black background is required. As delayed blend’s quality depends mainly on coincidence in form of the structure of the scene graph, it is not reliable and not recommended to use. Manual sorting is possible, but not user-friendly at all. Additive blending’s way of adding colour with every translucent surface is contrary to natural transparency, yielding unrealistic outputs.

The performance evaluation revealed that all algorithms, except depth peeling, are identical in speed. Moreover, they are as fast as rendering without transparency. Depth peeling’s performance decreases linearly with the number of passes. This can lead to problems with scenes which require lots of passes to be rendered satisfactorily. Rotating and zooming may be jerky and especially animations may not be smooth.
CHAPTER 7

Future work

Different opportunities for future works can be pursued:

A major improvement of the depth peeling implementation would be the support of shaders. The code it is based on has been updated and possibly supports shaders now [21]. This could be used as a basis. Another bug in depth peeling is that the near and far values are not updated properly (see 4.7.2). This issue could be looked into more closely.

Moreover, other transparency techniques could be included and evaluated. Another commutative blending technique to be examined could be additive blending with influence of the alpha value. Described in [13, pp. 201-202], this technique uses source alpha as source blending factor. [5] introduces dual depth peeling which needs only half as many passes as regular depth peeling. This eases depth peeling’s downside of poor performance with many passes.

ShowSurface is the only visualization algorithm which has been adapted. Currently, specifying whether a surface is transparent has to be done manually. This can be automatized by observing the alpha value. FAnToM includes many more visualization toolboxes which could be updated. A spinner could be a more fitting tool than a slider to set the number of passes in the Transparency View. A spinner widget has yet to be abstracted from the underlying Qt framework.
Bibliography


List of Figures

2.1 OpenGL 4 rendering pipeline. Blue boxes represent programmable stages. Source: [23] ................................................................. 4
2.2 Alpha blending (chequered areas indicate translucency). Source: adapted from [3] ............................................................. 7
2.3 Problems with depth sorting ................................................... 8
2.4 Additive blending (chequered areas indicate translucency). Source: adapted from [3] ............................................................. 9
2.5 Multiplicative blending (chequered areas indicate translucency). Source: adapted from [3] ......................................................... 9
2.6 Depth peeling strips away depth layers with each successive pass. The frames show the frontmost (leftmost) surfaces as bold black lines, hidden surfaces as thin black lines, and ‘peeled away’ surfaces as light grey lines. Source: [8] . . . 11
2.7 These images illustrate the layers of depth peeling from the nearest surface to the fourth nearest surface. Source: [8] ................................ 11
3.1 New GUI elements in FAnToM ............................................. 18
4.1 General comparison (alpha equals 0.5) ......................................... 23
4.2 Alpha value lowered to 0.3 ..................................................... 25
4.3 Lighting via shader .................................................................. 27
4.4 The dragon and the box intersect .............................................. 29
4.5 The triangles overlap cyclically .................................................. 32
4.6 Two-sided geometries (back faces visible) .................................. 35
4.7 Tilting the camera in depth sorted mode .................................... 35
4.8 Effect of the number of passes with depth peeling ....................... 38
4.9 Bug when zooming with depth peeling ....................................... 39
4.10 Problems with depth sorting via bounding boxes ..................... 40
5.1 Distribution of frame times of the wide scene ............................ 42
5.2 Distribution of frame times of the deep scene ............................. 43
5.3 Distribution of frame times of the deep, occluded scene ............... 44
5.4 Wide scene in different transparency modes ............................ 48
5.5 Deep scene in different transparency modes ............................. 50
5.6 Deep occluded scene: only a white triangle is visible ................ 50

59
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Blending equations. Source: [24, p. 171]</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Blending factors. Source: [24, p. 169]</td>
<td>6</td>
</tr>
<tr>
<td>5.1</td>
<td>Mean frame times from 100 measurements</td>
<td>45</td>
</tr>
</tbody>
</table>
Listings

2.1 Pseudocode for layer extraction in depth peeling using two depth buffers. Source: [8] ................................. 12

3.1 Source code for additive blending ........................................ 15

3.2 Source code for multiplicative blending ............................. 16

5.1 Part of the paint method with frame time measurement .............. 41
Acknowledgments

I thank Mario Hlawitschka for giving me the opportunity to write this bachelor thesis and for his professional advice.

I would also like to thank Jakob, Svø, Jenny, Gabriele and Wolfgang for existing.
Eigenständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann.


Christoph Blümel