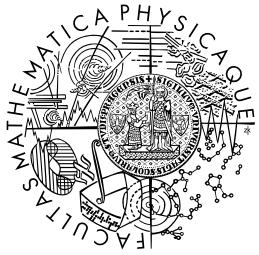

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Matúš Rožek

Order Independent Transparency

Department of Software and Computer Science Education

Supervisor of the bachelor thesis: Mgr. Martin Kahoun

Study programme: Computer Science

Prague 2024

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank my supervisor, Martin Kahoun, for his advice and feedback through this thesis and for teaching the Realtime Graphics on GPU course, which has shown me how cool rendering of triangles on a screen can be.

Title: Order Independent Transparency

Author: Matúš Rožek

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Martin Kahoun, KSVI

Abstract: Rendering transparent geometry in realtime brings a set of problems as the transparent objects need to be sorted first and rendered in order from back to front for their correct overlaying. A set of rendering algorithms called Order Independent Transparency (OIT) tries to accomplish this without sorting the geometry in advance. We create a program implementing five algorithms and compare their weaknesses, strengths, and properties. Some algorithms might excel in certain conditions and produce great results, yet fall short in slightly different environments. We aim to answer the question of which OIT algorithm is best suited for which scenarios.

Keywords: order independent transparency, transparency rendering, alpha compositing, realtime rendering

Název práce: Průhlednost nezávislá na pořadí

Autor: Matúš Rožek

Katedra: Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: Mgr. Martin Kahoun, KSVI

Abstrakt: Vykresľovanie priehľadnej geometrie v reálnom čase spôsobuje sadu problémov, pretože jednotlivé objekty potrebujú byť zoradené a vykreslené v poradí odzadu dopredu pre korektný výsledok. Existuje avšak trieda algoritmov, ktorá sa snaží o vykreslenie obrazu aj bez potreby priehľadné objekty najprv zoradiť. Tieto algoritmy sa nazývajú "priehľadnosť nezávislá na poradí" (anglicky Order Independent Transparency, skratka OIT). Vytvorili sme program, ktorý obsahuje 5 rôznych OIT algoritmov. Poukazujeme na silné a slabé stránky jednotlivých prístupov - niektoré môžu excelovať v určitých podmienkach, ale zato podávať výrazne slabšie výsledky v odlišnom prostredí. Snažíme sa odpovedať na otázku, ktorý OIT algoritmus je najlepšie využiť v akej situácii.

Klíčová slova: Průhlednost nezávislá na pořadí

Contents

Introduction

In the GPU based realtime rendering, we want to achieve a correct result no matter the order in which objects in a 3D scene are rendered. Hence, we render a new pixel only if it is closer to the camera than the pixel already present on the screen. This way, only the objects closest to the camera will be visible in the final image. For this a depth buffer[1] is used — a 2D texture in size of the window we are rendering into in which a distance from the camera for each pixel on the screen is stored.

If we try to render a new pixel onto a screen we first have to compare it's distance from the camera to the distance stored inside the depth buffer. Only if the pixel is closer than the already stored value we render it and update the distance in the depth buffer. This process is called depth testing. Thanks to the depth testing, we can render objects in a scene in any particular order. This gives us an option to, for example, render objects in such an order that expensive state changes in the GPU will be minimized, which increases the performance of rendering[2].

However, when rendering a transparent surface we want to be able to see the objects occluded by it as well. Therefore, the color of each transparent pixel has to be blended with its background.

Modern GPU hardware supports blending of transparent surfaces, yet, the resulting image happens to be dependent on the rendering order of the transparent objects, because we can always keep only the last rendered pixel. During blending of a transparent pixel, the color of the covered pixel must already be pre-computed. If we render an object closer to the camera before an overlapping object that is further away, then all sorts of visual artifacts will be present in the final image as can be seen in figure 1.

Enabling depth write during transparent object rendering would cause all farther transparent objects not to be visible. On the other hand, not writing to the depth buffer may cause those objects to overlap with foreground, producing incorrect results. As a result, it is necessary to render transparent objects from back to front.

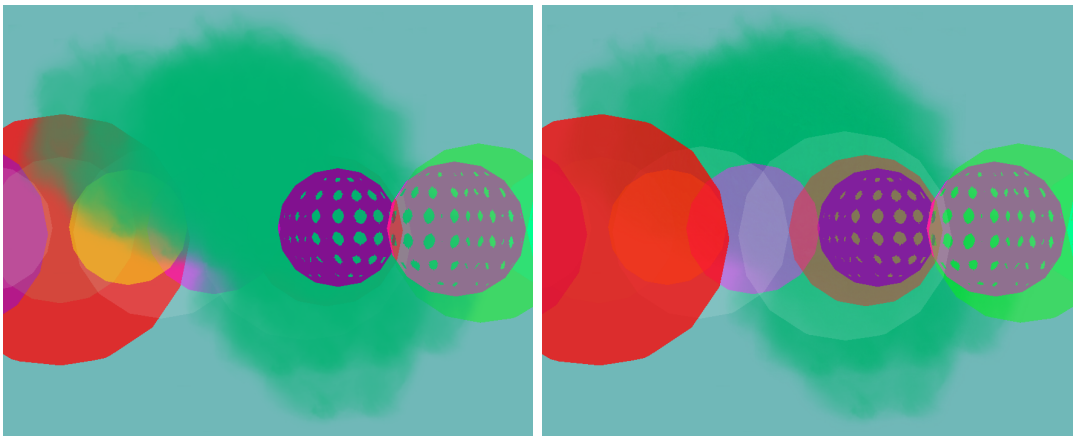

Figure 1 An example of image rendered with randomized ordering of transparent objects(left). Notice the green smoke in the background covering multiple balls that are much closer to the camera. Image rendered with correct ordering(right)

Rendering transparent objects from back to front might not be always straightforward, as multiple transparent objects might overlap, or we can render a complex non-convex transparent 3D object (such as a glass statue), or the entire 3D scene might be transparent.

We might then decide to sort individual triangles which the objects are made of and render those in the correct order. Not only would this approach be incredibly computationally inefficient (objects might be composed of thousands or even millions of triangles), but the issue would still remain present as 2 or more triangles might overlap.

Rendering of a transparent geometry is hence not a simple task. Over the years, multiple approaches to rendering without a need of sorting have been invented, many of which have found use in commercial software[3][4]. This class of rendering algorithms is called Order Independent Transparency (OIT)[5].

As a part of the thesis, we created a C++ program capable of rendering semi-transparent scenes with multiple OIT algorithms.

The rest of the thesis is organized as follows: chapter 1 contains the state of the art of OIT methods focusing more on the ones we've picked up for closer examination and comparison and also a slight introduction to the blending equation. Chapter 2 describes detailed implementation of the algorithms we chose, followed by a chapter 3 where we discuss the results.

1 Transparency Rendering

Here, we describe a several approaches used for rendering transparent geometry in order independent manner. First, let us explain a few technical terms and how alpha blending works.

1.1 Blending

Blending Equation

The blend equation introduced by Alvy Ray Smith and Ed Catmull in 1970[6] dictates the color of the final pixel in a following manner

$$C_f = C_s \cdot \alpha_s + (1 - \alpha_s) \cdot C_d \tag{1.1}$$

where $C_f = (R_f, G_f, B_f)$ denotes the red, green and blue components of the final color and α_s denotes a transparency of the given pixel. C_s and C_d are colors of the transparent pixel and covered pixel. All color components are in range $[0, 1]$.

Blending of Consecutive Surfaces

If multiple transparent objects are covered by each other then the equation 1.1 recursively unfolds into

$$C_f = C_1 \cdot \alpha_1 + (1 - \alpha_1) \cdot (C_2 \cdot \alpha_2 + (1 - \alpha_2) \cdot ... \cdot (C_n \cdot \alpha_n + (1 - \alpha_n) \cdot C_0) \tag{1.2}$$

where $C_1..C_n$ and $\alpha_1..\alpha_n$ denote colors and alphas of n transparent consecutive pixels and C_0 is a color of solid background (with $\alpha_0 = 1$).

Coverage

The equation 1.2 can be unfolded into the following form

$$C_f = C_1 \cdot \alpha_1 + (1-\alpha_1) \cdot C_2 \cdot \alpha_2 + (1-\alpha_1)(1-\alpha_2) \cdot C_3 \cdot \alpha_3 + ... + (1-\alpha_1) \cdot ... \cdot (1-\alpha_n) \cdot C_0 \tag{1.3}$$

If we rewrite it, we get

$$C_f = \sum_{i=1}^{n} C_i \cdot \alpha_i \cdot T_i(\alpha_1, ..., \alpha_n) \tag{1.4}$$

where

$$T_i(\alpha_1, ..., \alpha_n) = \prod_{j<i} (1 - \alpha_j) \tag{1.5}$$

Function $T_i(\alpha_1, ..., \alpha_n)$ is a Coverage of surface i and it is a product of reversed alphas of all surfaces closer to the viewport then surface i. Coverage is a measure of how visible the given transparent surface is on the screen.

Notice that coverage function is non-increasing and if we know it in advance for some n surfaces then we can convert blending of these surfaces into a sum as shown in equation 1.4.

Coverage of a solid background by transparent surfaces $(C_1, \alpha_1), ..., (C_n, \alpha_n)$ is simply $\prod_{i=1}^{n}(1 - \alpha_i)$.

1.2 Explanation of OpenGL Technical Terms

This section explains technical terms required to understand the implementation of algorithms.

Framebuffer is a set of multiple textures of the same resolution onto which we render using GPU. It consists of a depth buffer and at least one color attachment. We denote all framebuffers as $F = (C_1, ..., C_n, D)$, where C_1 .. C_n are different color attachments (each might be of a different format and might have different number of channels) and D is the depth buffer.

Render buffer is a type of depth buffer that can be used if we do not intend to read from it manually. It performs faster than classical depth buffers in some cases.

Color attachment is a texture into which we render during render pass. It can have up to 4 color channels and each channel can have different bit precision. Common types of color attachments include[7]:

- $RGBA8$ - color attachment with 4 channels, each composed of 8 bits. All values range in $[0, 255]$, but we usually treat them as normalized in interval $[0, 1]$
- $RGBA16F/RGBA32F$ - 4 channels, each is a signed unclamped floating point number with 16/32bit precision
- $R16F/R32F$ - one signed floating point channel with 16/32bit color precision

During one render pass the GPU can write **into only one** framebuffer, however, if the framebuffer contains multiple color attachments then we can render into all of them in parallel. This is called rendering into **multiple render targets**.

During one render pass, we can write into only one framebuffer, but can read from multiple textures (this includes color attachments of different framebuffers). For example, we can render our scene into a texture in one step and then later read from the given texture during another render pass. It is forbidden to read from texture which is bound to the framebuffer we are rendering into.

We are not able to read from a texture we are writing to at the same time, however, modern rendering APIs allow us to use **blending**. If blending is enabled, then the color currently in a texture is not overwritten by the new source color, but instead blended with it by a certain blend operation.

We show some blending operations that are possible. C_f denotes the final color of the given pixel after the blending operation ends, C_d denotes the current (destination) color of the pixel and C_s (source) is the color we are writing into the pixel during rendering. We can also use alpha of the source color $C_s.a$ and alpha of the destination color $C_d.a$. These are some of the possible blending operations[8]

- $C_f = C_d \cdot 1 + C_s \cdot 1$
- $C_f = C_d \cdot 0 + C_s \cdot C_s.a$
- $C_f = C_d \cdot (1 - C_s.a) + C_s \cdot C_s.a$
- $C_f = C_d \cdot (1 - C_s.a) + C_s \cdot (1 - C_d.a)$

A separate blending operator can be configured for a render target, or even for some channels of texture. For example, we can have one blending operation for the RGB channels of a color component and another for the A channel.

During rendering, we can **disable depth testing**. If depth testing is disabled, then pixels of all objects will be rendered into the framebuffer no matter their distance from the screen. Also, no pixels will be written into the depth buffer. It is possible to only **disable depth write**.

That way, the depth testing will be still enabled and only the pixels closer to the camera then the content of the depth buffer will be written. However, the values in depth buffer will not be updated. We can use this for rendering the solid background and keeping the depth buffer around. During transparent rendering we will not write to it, but we will compare all transparent pixels to the values stored to it and therefore no objects that are occluded will be rendered.

Shaders are a simple programs written in GLSL language that run on the GPU. OpenGL supports multiple types of shaders, but we will limit ourselves here only to **fragment shaders**. These allow us to decide how is the color of the rendered object calculated. We will denote all of our shaders as functions that output one or more color values. An example of a shader might be $S : C \rightarrow Z \times A$, which receives some value C (might be RGBA color, floating point value or other) and outputs values Z and A into first and second render target respectively.

1.3 Overview of the current OIT methods

Multiple approaches to solving a problem with rendering of a transparent geometry have been developed, like the alpha buffer used in Pixar Reyes[9] for instance, which stores all transparent pixels sorted by their depth and then blends them in a correct order. Nonetheless, we will limit ourselves only to the ones that can be used in realtime graphics, where the performance is crucial. Now, lets finally get to the OIT rendering methods and their description.

Depth Peeling

We will start with the *depth peeling*[10], which is an OIT rendering approach proposed in 2001. It renders the transparent geometry in multiple iterations and produces sorted like results (as if the geometry has been rendered in a sorted order).

In each iteration, the algorithm renders all of transparent geometry and collects ("peels") the front most rendered layer of transparent pixels, which are then blended into a separate buffer. For this 2 depth buffers are used - one which stores the distance to the furthest peeled layer and the second one which denotes distance from closest solid surface. *Depth peeling* renders all objects in range between these 2 depth buffers and then moves the first depth buffer forward before the next iteration.

In each pass, only the closest non-rendered layer of transparent pixels is rendered and therefore we achieve an ordering. The number of passes is proportional to the number of transparent layers and in each pass all transparent objects have to be rendered, which puts the upper complexity bound of the algorithm in $O(n^2)$ region, where n is the number of transparent layers.

Depth peeling produces sorted like result(if we run it for a sufficient number of passes) for a cost of more rendering passes and a lower performance.

It was further improved by *dual depth peeling* algorithm [11], which peels 2 layers of transparent surfaces in each iteration and *multi layer depth peeling*[12], which can peel 4 layers per iteration.

We decided to implement this method ourselves to compare image results of other methods against it.

Stochastic Transparency

This technique[13] introduced in 2011 treats all transparent pixels as opaque and discards some of them in the fragment shader. The number of discarded pixels is proportional to transparency of a surface. Follow up technique, *depth-based stochastic transparency*, introduced in the same paper takes more rendering passes and in turn produces more realistic image for a cost of a higher runtime.

Stochastic transparency and *depth-based stochastic transparency* turn transparency problem into rendering of solid geometry. Similar approach was used in the video game GTA5[14] for rendering of a distant vegetation.

It overall produces much more convincing results then *weighted sum* technique and has better runtime than *depth peeling*.

Adaptive Transparency and Multi Layer Alpha Blending

Adaptive transparency[15] and *multi layer alpha blending*[16] are approaches which store fixed amount of samples per pixel (in the first case only the transparency and the depth of a fragment are stored, in second one we store also the color).

If the number of transparency layers exceeds the fixed amount then two samples are merged into one. The first approach uses 2 transparent geometry passes, collecting and merging the samples in the first one and then later approximating coverage function for individual transparent pixels in the second pass. The second technique requires only one transparency rendering pass.

These techniques produce a sorted-like result for a scenes with a small amount of transparent surfaces, yet require more memory than other approaches listed.

Weighted Sum

Weighted sum introduced by Meshkin in 2007[17] takes a rather different approach at rendering transparency. Instead of trying to achieve a ground truth result it strives for simplicity and only estimates the final image by summing up the non-order-dependent members of the blending equation 1.2.

The produced results are rarely correct if multiple layers of transparent geometry are present and *weighted sum* often produces brighter colors because colors of the transparent surfaces are summed instead of interpolated.

However, it makes up for this disadvantage by a great run time, as it requires only one transparent geometry pass and produces convenient results for a geometry with low transparency values. It is the first Blended OIT[18] algorithm and it lay the ground for further improvements in the field over the years.

Weighted Average

Weighted average[11], which was introduced together with the *dual depth peeling*[11] in 2008 is another Blended OIT algorithm and it builds upon *weighted sum* in multiple ways.

It averages the color and alpha values of all transparent objects into a separate buffer. Then, in a full screen pass the coverage of background is approximated from the averaged alpha channel and the averaged transparent colors are blended with solid background.

Similar to *weighted sum*, the resulting image is correct if there is only one layer of transparent surfaces. For any higher number of surfaces the algorithm does not produce a correct result.

Also, due to averaging colors of all transparent surfaces without putting any significance to their distance from viewpoint, it produces visible artifacts if objects with a low transparency are covered by a high-transparency geometry. This problem is further explored and improved by the *weighted blended OIT*[18] algorithm.

Nonetheless, it is particularly accurate at rendering transparent geometry with similar transparency across the whole scene and the final image looks much more convincing then the *weighted sum*.

Weighted Blended OIT - WBOIT

Weighted Blended OIT[18] algorithm builds on the ideas of *weighted average* approach mentioned above, but improves its blending functionality in multiple ways. The background coverage is calculated precisely in all cases by multiplying alphas of all consecutive transparent pixels. Similar to the *weighted average*, it averages all transparent surfaces, but this time multiplies them with weights, assigning more importance to certain surfaces over others.

The weight function tries to approximate the coverage function 1.4. The paper itself proposes multiple ways to calculate weights, often based upon transparency levels and distance of individual surfaces.

The resulting image looks very appealing and it is a significant improvement over *weighted average* for similar runtime and memory consumption. It requires less memory or rendering passes then *adaptive transparency* and *depth-based stochastic transparency*, yet falls short in certain corner cases.

Moment Transparency - MBOIT / MOMENT OIT

Moment based OIT[19] or *moment OIT*[20], introduced in 2018 independently by different authors, improves upon *WBOIT* by introducing a new, more precise weights function.

The new weight function is computed during the runtime of the algorithm in a separate transparent geometry pass using *HamburgerMSM* algorithm used in *moment shadow mapping*[21]. During this pass the optical depth of the transparent geometry with depth moments is stored into a separate framebuffer. In the next rendering pass the coverage function is approximated from the stored data and used as a weight function for individual surfaces. This approach found its use in the game Alan Wake 2 released in 2023[3].

For brevity, we will use the name *MBOIT* to denote both of these methods.

2 Implemented Transparency Rendering Algorithms in Detail

We decided to implement and investigate the following OIT algorithms in detail. We picked all Blended OIT methods (*WBOIT*, *weighted average*, *weighted sum*, *moment transparency*) and we also implemented the *depth peeling* algorithm for the ground truth(sorted like) comparison.

In this chapter we intent to describe both the implementation and theoretical details of the chosen algorithms and also the implementation of the application capable of rendering the scene using these algorithms. For rendering one frame of a scene, the currently used algorithm receives:

- a list of opaque and transparent objects. Each object is represented as a set of vertices in 3D space, indices, which connect individual vertices into triangles and a texture.
- Position and a rotation of the camera in the scene for the current frame and its perspective matrix. Using these two, a world-to-view matrix can be calculated, which can transform positions of individual vertices in a world space onto a screen space.
- A set of stable data that persist between the frames. These can be compiled GPU programs used for rendering, additional framebuffers or other type of data that is required by the algorithm but does not necessary have to change between frames.

It would be possible to use one algorithm for rendering of the solid background, as each of the algorithms has to do this step, and then using a second algorithm to only add the transparent objects to the result. We decided not to use this approach and instead let each of the algorithms render opaque objects by itself, as this solution gives us more flexibility in implementation of individual algorithms.

2.1 Depth Peeling

Depth peeling algorithm produces an accurate result for a cost of higher run time. In practice, it must render transparent geometry in multiple passes, collecting ("peeling") some layers of transparent geometry away and storing them in an accumulation buffer.

The algorithm uses two depth buffers to find transparent geometry closest to the camera which was not yet rendered. In each pass we collect only the geometry closest to the front buffer. Hence, the number of passes is proportional to number of transparent layers present. Depth testing using multiple depth buffers is usually not supported by the hardware and therefore we have to emulate one depth test inside the fragment shader, where we discard pixels manually. This can be seen in step 7 of the algorithm pseudocode 1.

Depth peeling algorithm has seen a further improvement since 2001 in a form of *dual depth peeling*[11] and *multi layer depth peeling*[12], which introduce a way to peel more than one layer of transparent surfaces per render pass.

Our implementation "peels" only one layer of transparent surfaces each pass and runs in predetermined number of iterations. This is the basic version of the algorithm as stated in the source paper [10] and it does not use any optimizations from later proposals [12][11].

See algorithm 1 for overview of the basic *depth peeling* implementation.

Algorithm 1 Depth Peeling pseudocode

Require: Framebuffers $F_1 = (C_1, D_1)$, $F_2 = (C_2, D_2)$ and $F_3 = (C_3, D_3)$ with *RGBA8* color components C_1, C_2, C_3 and depth buffers D_1, D_2, D_3. F_1 stores the color and depth of the opaque background, F_2 keeps around the depth and color of the last peeled layer, which is then accumulated at the end of each iteration into F_3

Output: Final scene rendered in F_1 framebuffer

// Cleanup the framebuffers after the previous frame

$C_3 \leftarrow (0, 0, 0, 1)$, $D_3 \leftarrow 0$

$C_1 \leftarrow$background color, $D_1 \leftarrow 1$

Render all solid geometry and depth into C_1 and D_1

for number of iterations **do**

// Copy the depth of the opaque objects

$D_2 \leftarrow D_1$

// Cleanup after previous iteration

$C_2 \leftarrow (0, 0, 0, 0)$

// The closest layer of geometry that is further from camera than D_3 and closer to the camera then D_2 will be rendered

Render all transparent geometry into C_2 and D_2 without blending as if it was solid. Read and write depth using D_2 with hardware support. Simultaneously use D_3 as a texture and discard all pixels that are closer to the camera than distance in D_3

// Now, we need to blend the closest layer into an accumulator

Enable blending

Set separate blending operator for RGB channels as $C_f = C_s \cdot \alpha_d + C_d \cdot 1$

Set separate blending operator for alpha channel as $C_f = C_s \cdot 0 + C_d \cdot (1 - \alpha_s)$

Do a full screen pass and blend C_2 into C_3

Disable blending

// Save the distance to the closest layer of transparent geometry so that we do not render it again in next iteration

$D_3 \leftarrow D_2$ (might be omitted in the last iteration)

end for

// Merge the color of transparent objects from accumulator with the opaque background

Enable blending and set blending operator to $C_f = C_s \cdot 1 + C_d \cdot \alpha_s$

Do a full screen pass and blend accumulated color in C_3 into C_1

2.2 Weighted Sum

The *weighted sum* algorithm calculates the final color C_f in the following way

$$C_f = \sum_{i=1}^{n} C_i \alpha_i + C_0 \left(1 - \sum_{i=1}^{n} \alpha_i\right) \tag{2.1}$$

Where C_f is the final color, $C_1 ... C_n$ are colors of individual transparent surfaces and $\alpha_i ... \alpha_n$ their alphas. C_0 is the color of the opaque background.

The algorithm requires one solid pass, one transparent pass and one full screen pass at platforms where blending of negative values is not possible.

Notice that the right part of the equation might produce values that are higher than 1 or negative. In OpenGL, clamping of values into an interval $[0, 1]$ or $[-1, 1]$ occurs[8] for 8bit unsigned and signed colors respectively. However, the floating point colors of accuracy *F16* or *F32* are not clamped. Using these, it would be possible to implement *weighted sum* algorithm using only one framebuffer with color accuracy $F16$.

Meshkin algorithm produces plausible results for objects with low opacity and ground truth results if there is only one layer of transparent objects . However, in scenes with higher alpha values the sum of the alphas and colors produces large values that are displayed in the final image as oversaturated towards the color of transparent pixels as can be observed in figure 2.1.

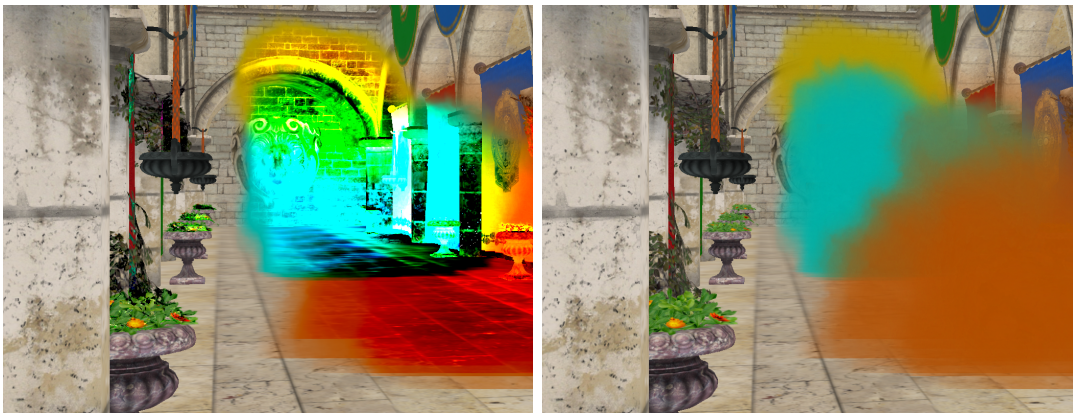

Figure 2.1 Smoke particles in Sponza scene. *Weighted sum*(left) and *depth peeling*(right)

An implementation of *weighted sum* might look like algorithm 2. This implementation does not use the optimization with only one framebuffer of $F16$ type mentioned above. Instead, we use one *RGBA8* framebuffer for the solid background and one *RGBA16F* for transparent geometry.

Algorithm 2 Possible implementation of weighted sum

Require: Framebuffers $F_1 = (C_1, D_1)$, $F_2 = (C_2, D_2)$ and $F_3 = (C_3, D_3)$. F_1 and F_3 have *RGBA8* colors, F_2 has *RGBA16F* color. F_1 is used for storing the opaque background, F_2 as an transparency accumulator. We merge these two results in the final step into the F_3 framebuffer.

Output: Final scene rendered in F_3 framebuffer

// Cleanup after the previous frame

1: $C_1 \leftarrow$background color
2: $D_1 \leftarrow 1$
3: Render all opaque geometry into C_1 and D_1

// Cleanup the framebuffer into which we sum up the colors

4: $C_2 \leftarrow (0, 0, 0, 0)$
5: $D_2 \leftarrow D_1$
6: *// We want to turn depth writing so that we render all layers of transparent geometry, not just the closest one*
7: Turn off depth writing, but still use D_2 for depth testing
8: Enable blending with blend operator $C_f = C_s \cdot 1 + C_d \cdot 1$
9: Render all transparent geometry into C_2
10: Turn off blending
11: Do a full screen pass into C_3. Use C_1 and C_2 as textures and calculate the final color $C_f = C_2.rgb + C_1.rgb \cdot (1 - C_2.a)$

2.3 Weighted Average

At GDC 2008 Bavoid and Myers[11] presented (together with *dual depth peeling* mentioned above) an OIT method of their own. Their *weighted average* algorithm computes the resulting color of transparent scene in a following manner

$$C_f = \frac{\sum_{i=1}^{n} C_i \cdot \alpha_i}{\sum_{i=1}^{n} \alpha_i} \left(1 - \left[\frac{1}{n}\sum_{i=1}^{n} \alpha_i\right]^n\right) + C_0 \cdot \left[\frac{1}{n}\sum_{i=1}^{n} \alpha_i\right]^n \tag{2.2}$$

Individual members of the equation have the same meaning as in the section 2.2.

Weighted average renders the transparent geometry in 2 passes and uses 2 render targets for the first pass. In the first pass it accumulates the color and alpha values of transparent layers into one render target and uses the second render target as an integral counter for the number of transparent layers per pixel.

The second pass is a full screen pass during which the accumulated colors and alpha values are averaged and blended with the solid background rendered previously. We effectively sum up the color and alpha of all consecutive transparent pixels and therefore a higher precision color buffer type is required (one that is not clamped, so usually $F16$ or $F32$).

The ideal condition for this algorithm is a fully transparent scene with mid to low range alpha values. In such a case the *weighted average* produces images that are believable to a human eye, yet not always quite realistic as can be seen in figure 2.2.

Figure 2.3 Overbleeding of transparent plants from behind the smoke particles in *weighted average* algorithm(left). *Depth peeling*(right)

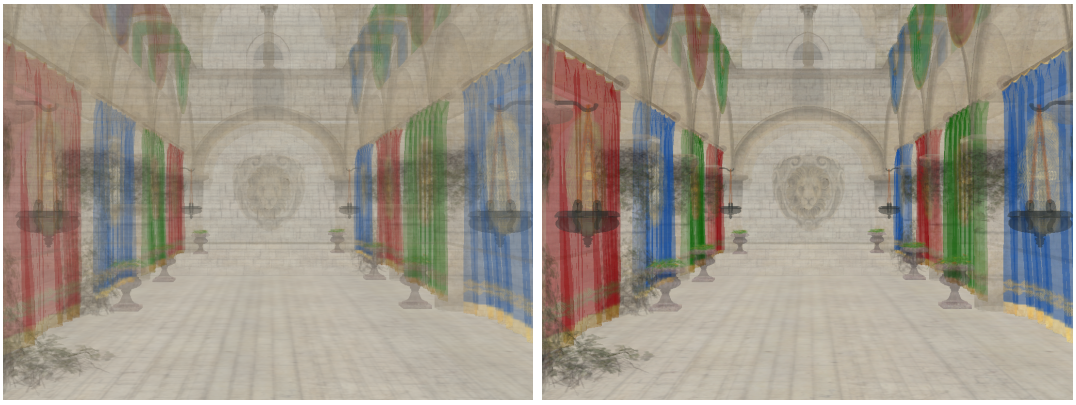

Figure 2.2 *Weighted average*(left) produces less colorful, yet still believable result in the Sponza scene with 25% transparency. *Depth peeling*(right)

The averaging of consecutive surfaces might cause visible artefacts in the final image if the transparent objects of multiple colors are present (semi-transparent bushes in Sponza scene for example). The closest transparent pixels have the same importance in the final result as the ones behind them covered by multiple layers of transparent geometry. This can produce artifacts visible to the human eye as can be observed in figures 2.3 and 2.4.

In an environment where all alpha values α_i are the same the background coverage is computed exactly. However, this might not apply for scenes with varying alpha values as can be seen in figure 2.5.

Other issue is that the number n might be composed of objects with transparency $\alpha_i = 0$. This might cause artifacts in scenes where the opacity of objects varies.

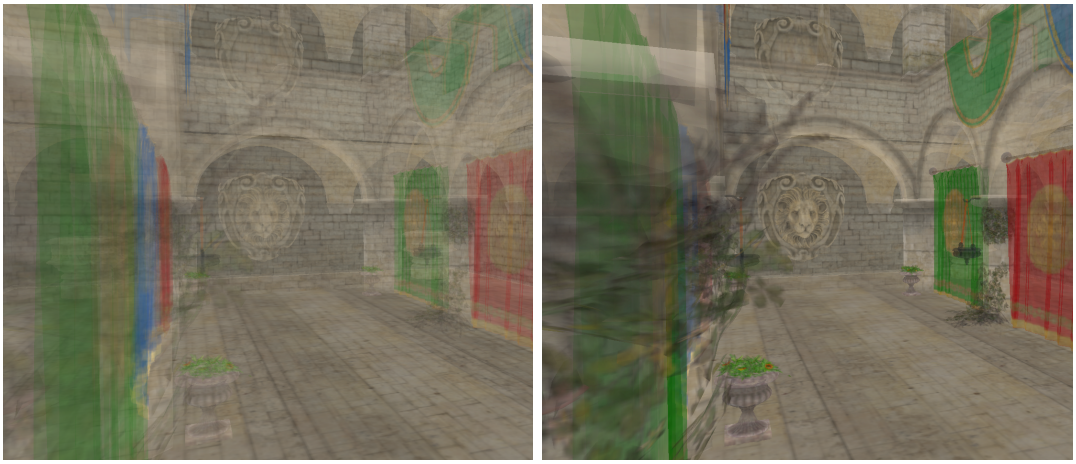

Figure 2.4 Sponza scene with decreased transparency. Color of the leaves at the front gets averaged with background in *weighted average*(left). *Depth peeling*(right)

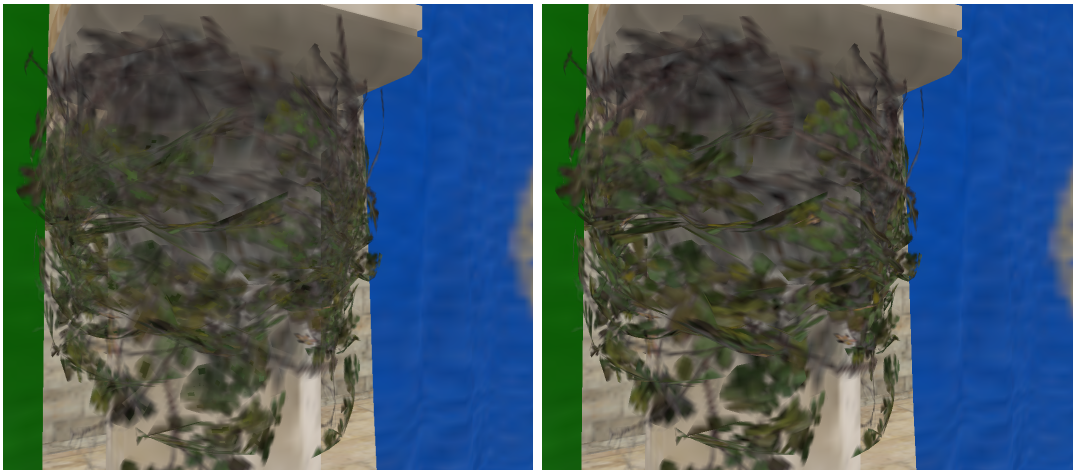

Figure 2.5 Sponza scene with low resolution leaves textures)In *weighted average*(left), parts of the leaves geometry are fully transparent, however, these will still count into the number of consecutive surfaces and will introduce slight artifacts. *Depth peeling*(right)

We implemented *weighted average* method using framebuffer components of both 32bit floating colors (*R32F* and *RGBA32F*) and 16bit colors (*R16F* and *RGBA16F*). Instead of attaching a depth buffer we used a render buffer as we do not need to read the depth during the runtime of the algorithm.

For *weighted sum*, we will describe the calculations of some shaders it uses, because these are not as trivial as in the previous algorithms.

We will use shader $S_{oit} : A \times N \to C$, where A is the accumulated $RGBA$ color, N is a floating point number greater than 0 and C is the resulting $RGBA$ color. The shader calculates the resulting color in the following way.

$$S_{oit}(A, N) = (\frac{A.rgb}{A.a}, \left[\frac{A.a}{N}\right]^N) \tag{2.3}$$

It will be used for blending of accumulated colors into the final image.

We will also need a shader $S_{accum} : C \to A \times N$, which receives a color of the

pixel C and outputs $RGBA$ color A and a float value N in a following way

$$S_{accum}(C) = (C.rgb \cdot C.a, C.a) \times (1) \tag{2.4}$$

This one will be used for accumulating the color values into multiple render targets.

The implementation of *weighted average* can be seen in algorithm 3.

Algorithm 3 Weighted Average pseudo code

Require: 2 framebuffers $F_1 = (C_1, D_1)$ and $F_2 = (C_{2s}, C_{2q}, D_2)$, where D_1 and D_2 are render buffers and C_1 is color component of type $RGBA8$. C_{2s} and C_{2a} are color components of types either $R32F$ and $RGBA32F$ or $R16F$ and $RGBA16F$. F_1 is used as a storage of the opaque background and F_2 as an accumulator of color and alpha values.

Output: Final scene rendered in F_1 framebuffer

1: $C_1 \leftarrow$background color
2: $D_1 \leftarrow 1$
3: Render all solid geometry into C_1 and D_1
4: $C_{2q} \leftarrow (0, 0, 0, 0)$
5: $C_{2s} \leftarrow 0$
// Use the solid depth so that we only render transparent geometry that is not occluded by the solid one
6: $D_2 \leftarrow D_1$
7: Enable blending with blend operator $C_f = C_d \cdot 1 + C_s \cdot 1$
8: Turn off depth writing
// Using blending, we store the number of consecutive surfaces into C_{2s} and accumulate their colors into C_{2q} for each pixel on the screen
9: Render all transparent geometry into C_2. During rendering, write 1 into C_{2s} for each rendered pixel and write it's color and alpha into C_{2q}. We do this using an S_{accum} shader
10: Change blending operator to $C_f = C_s \cdot (1 - \alpha_s) + C_d \cdot \alpha_s$
11: Do a full screen pass into F_1 during which we sample values from C_{2s} and C_{2q} and blend them with color which is already in C_1 using the $S_{oit}(C_{2q}, C_{2s})$ shader

---

2.4 Weighted Blended Order Independent Transparency

In 2013 McGuire and Bavoil introduced new blending algorithm[18] in their paper *weighted blended order independent transparency* (we use *WBOIT* for brevity). The *WBOIT* calculates the following color

$$C_f = \frac{\sum_{i=1}^{n} C_i \cdot \alpha_i \cdot w(z_i, \alpha_i)}{\sum_{i=1}^{n} a_i \cdot w(z_i, \alpha_i)} \left(1 - \prod_{i=1}^{n}(1 - \alpha_i)\right) + C_0 \cdot \prod_{i=1}^{n}(1 - \alpha_i) \tag{2.5}$$

Individual members of the equation have the same meaning as in section 2.2. In addition, $z_1 ... z_n$ are depths of the individual transparent pixels and the $w(z_i, \alpha_i)$ is a weight function, which we describe later.

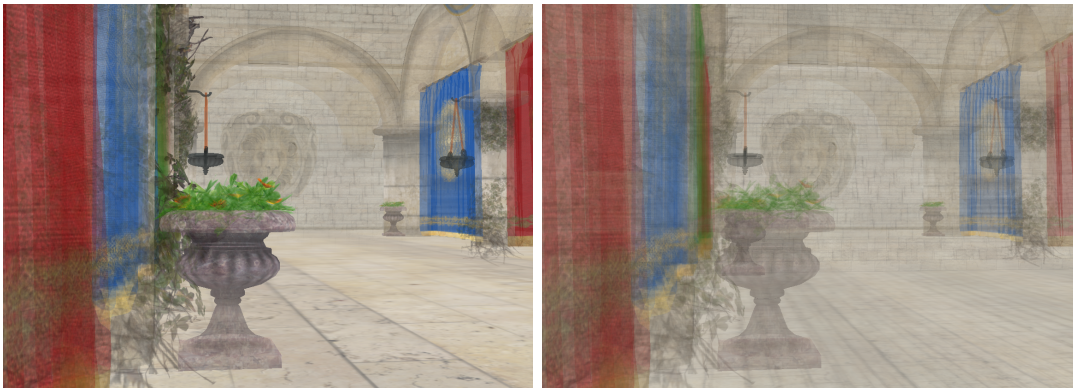

Figure 2.6 *WBOIT*(left) assigns more importance to closer surfaces in the Sponza scene with 25% transparency. *Weighted average*(right)

We should note that this equation differs from the one stated in the original paper by the member α_i on the left side in numerator. Bavoil and McGuire assumed an alpha-premultiplied color in their paper, which we do not and therefore we explicitly state the α_i member in the equation.

During the transparent geometry pass the colors and alphas of all transparent objects are summed up into two render targets. Later, during the full screen pass these colors are averaged out and blended with solid background. Weight function assigns weights to individual pixels of transparent geometry based on their distance and alpha. This way, the objects closer to the camera should have greater weight in the average than the objects further from it. This improves upon the problem of averaging multiple surfaces which *weighted average* algorithm has as can be seen in figure 2.6.

To compute a coverage of the background *WBOIT* uses a clever trick during the blending stage, thanks to which the individual alpha channels (1 - α_i) of the transparent objects are multiplied (instead of summed up). Note that this product $\prod_{i=1}^{n}(1-\alpha_i)$ is the actual coverage of the background. Unlike the previous method, this one computes the correct coverage in all circumstances.

One possible implementation of the weight function is to use none at all, e.g. $w(z_i, \alpha_i) = 1$, which simply averages all transparent objects and blends them with solid background. This version of the algorithm inherits some weaknesses of the *weighted average* algorithm, but calculates the coverage of solid background more accurately than it's predecessor. The difference between some weight function and no weight function might be observed in figure 2.7

Notice that for $w(z_i, \alpha_i) = 1$ and $a_i = a_j$ for all pairs i and j the *WBOIT* equation produces the same result as *weighted average* equation. This means that the *WBOIT* algorithm without a weight function calculates the same color as *weighted average* in environments where all alpha values of transparent objects are the same.

As the source paper states, one disadvantage of the weight functions is that simply rescaling all scene geometry around the camera moves it into different depth range (distance from all objects changes) and will result in different image.

Our implementation of *WBOIT* uses an accumulation shader $S_{accum} : C \times Z \rightarrow A \times T$, where C is an *RGBA* color, Z is a depth of a pixel in $[0, 1]$ interval and T and A are two render targets with float and *RGBA* types. The shader calculates

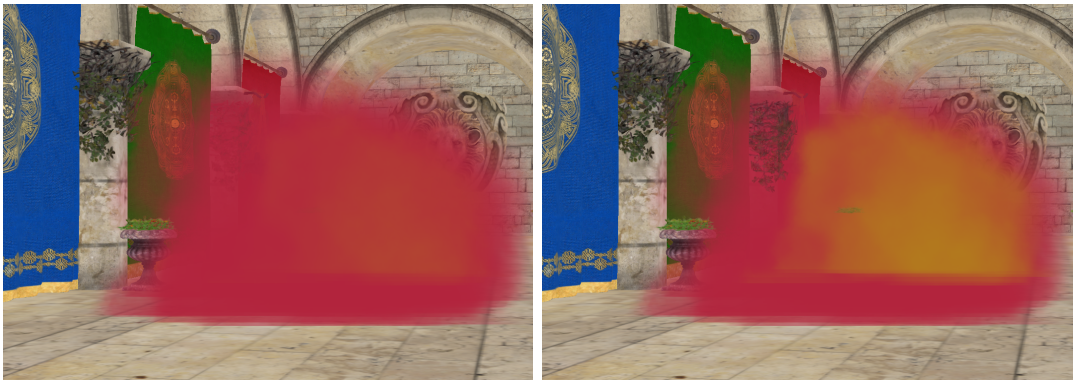

Figure 2.7 *WBOIT* with weight function $w = 1/z^5 \cdot \alpha$ on the left. *WBOIT* with no weight function on the right. Notice the overbleeding of transparent vegetation and yellow smoke from behind the red one

output color in the following way

$$S_{accum}(C, Z) = (C.rgb \cdot C.a \cdot W(Z, C.a), C.a \cdot W(Z, C.a)) \times (C.a) \tag{2.6}$$

This shader requires a weight function $W(Z, A)$, which assigns an importance to individual samples based on their alpha value A and distance from camera Z.

We implemented the following weight functions

$$W_1(Z, \alpha) = 1 \qquad W_2(Z, \alpha) = \frac{1}{Z^3} \qquad W_3(Z, \alpha) = \frac{1}{Z^5}$$
$$W_4(Z, \alpha) = \frac{1}{Z^3} \cdot \alpha \qquad W_5(Z, \alpha) = \frac{1}{Z^5} \cdot \alpha$$

Each of the weight function can be made *adaptive* by dividing the distance Z by Z_s before feeding it into the W function. Here, Z_s is the distance to the closest opaque surface (we can read it from the depth buffer). No transparent surface will be further from the camera than Z_s and therefore $Z/Z_s \in [0, 1]$.

Another shader we need is a merging shader $S_{oit} : A \times T \to C$, where A and T are accumulated *RGBA* colors and multiplied alphas. It renders color in the following way

$$S_{oit}(A, T) = \left(\frac{A.rgb}{A.a}, T \right) \tag{2.7}$$

An implementation of *WBOIT* might look like the algorithm 4.

Algorithm 4 WBOIT Algorithm pseudocode

Require: Framebuffers $F_1 = (C_1, D_1)$ and $F_2 = (C_{2s}, C_{2q}, D_2)$. C_1 is of type $RGBA8$, C_{2s} is of type $R32F$ and C_{2q} has type $RGBA32F$. F_1 stores the color and depth of the opaque geometry and F_2 accumulates colors and coverage of the background.

Output: Final scene rendered in F_1

// Cleanup after the previous frame

1: $C_1 \leftarrow$background color
2: $D_1 \leftarrow 1$
3: Render all solid geometry into C_1 and it's depth into D_1 with depth testing and writing on and blending off
4: $D_2 \leftarrow D_1$
// This value is the background coverage. We start with 1 and then we lower it by multiplying it with (1-α) of each transparent surface
5: $C_{2s} \leftarrow 1$
// These is the color accumulator. We start with no colors and then add some during the rendering
6: $C_{2q} \leftarrow (0, 0, 0, 0)$
7: Disable depth writing, but keep depth testing on
8: Use 2 render targets C_{2s} and C_{2q}
9: Enable blending
10: Set the blending operator of C_{2s} to $C_f = C_s \cdot 0 + C_d \cdot (1 - \alpha_s)$
11: Set the blending operator of C_{2q} to $C_f = C_s \cdot 1 + C_d \cdot 1$
12: Render all transparent geometry into C_{2s} and C_{2q}. The final color is calculated as $C_{2q}, C_{2s} = S_{accum}(C)$, where C is an $RGBA$ color of the currently rendered transparent pixel
13: Use blend operator $C_f = C_s \cdot (1 - \alpha_s) + C_d \cdot \alpha_s$
14: Do a full screen pass into F_1. C_1 already contained color of opaque geometry and in this step we only add transparent objects

2.5 Moment Transparency

The last OIT approach we implemented is the moment transparency. For the sake of brevity we will use name *"MBOIT"*. *MBOIT* is quite similar to its predecessor 2.4. The main difference is the modified weight function

$$w(z, \alpha) = exp(-Hamburger4MSM(\frac{d}{\hat{d}}, z)) \tag{2.8}$$

The definition of *Hamburger4MSM* can be seen in algorithm 5. The rest of the equation remains the same

$$C_f = \frac{\sum_{i=1}^{n} C_i \cdot \alpha_i \cdot w(z_i, \alpha_i)}{\sum_{i=1}^{n} a_i \cdot w(z_i, \alpha_i)} \left(1 - \prod_{i=1}^{n}(1 - \alpha_i)\right) + C_0 \cdot \prod_{i=1}^{n}(1 - \alpha_i) \tag{2.9}$$

Instead of leaving a decision to choose a weight function upon a programmer, *MBOIT* calculates it by itself in one extra render pass. During this additional

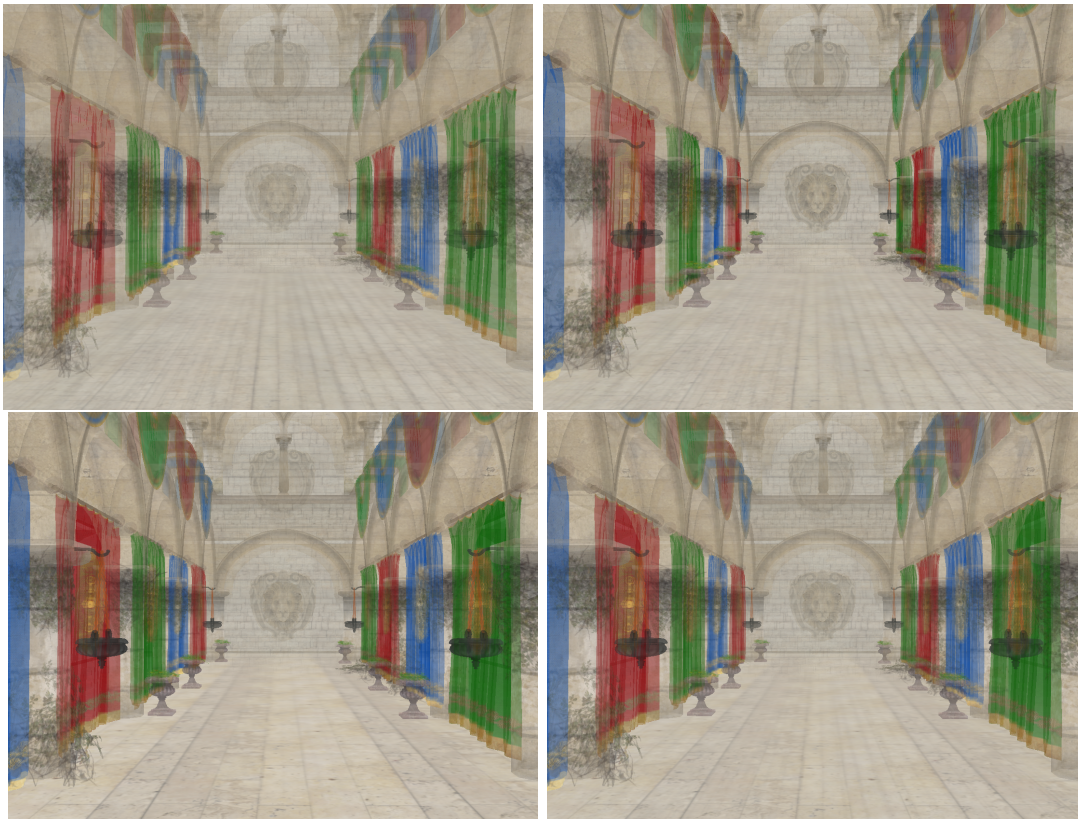

Figure 2.8 Sponza scene with 25% transparency.*MBOIT*(top left) produces almost the same result as *depth peeling*(top right). *WBOIT* with weight functions $w = 1/z^5 \cdot \alpha$ and $w = 1/z^3 \cdot \alpha$ on bottom left and bottom right respectively produce both slightly brighter image.

render pass moments of transparent surfaces are collected and summed up. In later render pass these moments are retrieved and used to calculate the appropriate weights.

As the source paper states, the new weight function calculates the coverage of transparent pixels correctly only for 2 or less surfaces covered by each other. If there are more than 2 surfaces overlapping then the resulting coverage is only approximated, but usually quite precisely as can be seen in figure 2.8.

The source papers suggest multiple versions of this algorithm. Either by using more moment passes, using trigenometric or quantitized moments instead of power moments or storing 8 moments in 2 color buffers.

We implemented the basic version of *MBOIT* algorithm with one transparency render pass for collecting 4 power moments and one render pass for accumulating the transparency.

A description of the *Hamburger4MSM* algorithm from the *moment shadow mapping* paper[21] can be seen in algorithm 5. We will need this later in the S_{weight} shader.

Algorithm 5 Hamburger4MSM Algorithm

Input: $b \in \mathbb{R}^4$, fragment depth $z_f \in \mathbb{R}$, bias α
Output: Coverage amount

1: $b' \coloneqq (1-\alpha) \cdot b + \alpha \cdot (0.5, 0.5, 0.5, 0.5)^T$
2: Use a Cholesky decomposition to solve for $c \in \mathbb{R}^3$:
$$\begin{pmatrix} 1 & b'_1 & b'_2 \\ b'_1 & b'_2 & b'_3 \\ b'_2 & b'_3 & b'_4 \end{pmatrix} \cdot c = \begin{pmatrix} 1 \\ z_f \\ z_f^2 \end{pmatrix}$$
3: Solve $c_3 \cdot z^2 + c_2 \cdot z + c_1 = 0$ for z using quadratic formula and let $z_2, z_3 \in \mathbb{R}$ denote the solutions
4: If $z_f \leq z_2$: Return 0
5: Else if $z_f \leq z_3$: Return $\frac{z_f \cdot z_3 - b'_1 \cdot (z_f + z_3) + b'_2}{(z_3 - z_2) \cdot (z_f - z_2)}$
6: Else: Return $1 - \frac{z_2 \cdot z_3 - b'_1 \cdot (z_2 + z_3) + b'_2}{(z_f - z_2) \cdot (z_f - z_3)}$

We will use additional shader $S_{moment} : Z \times \alpha \to M \times S$, which receives depth Z of the pixel, its transparency α and outputs 4 moments in M in $RGBA$ form and their sum S as a float.

$$S_{moment}(z, \alpha) = ((z, z^2, z^3, z^4) \cdot -log(1-\alpha)) \times (-log(1-\alpha)) \tag{2.10}$$

Later, we use the shader $S_{weight} : M \times S \times Z \to T$, which receives previously calculated moments M in $RGBA$ form, their sum S as a float and the depth of a pixel Z. This shader approximates the coverage of a pixel in depth Z.

$$S_{weight}(M, S, Z) = exp(-Hamburger4MSM(M/S, Z, 0) \cdot S) \tag{2.11}$$

MBOIT can be implemented in a similar way to the pseudocode 6.

Algorithm 6 MBOIT Algorithm pseudocode

Require: The same resources as *WBOIT* algorithm with addition of framebuffer $F_M = (C_{Ms}, C_{Mq}, D_M)$ where C_{Ms} and C_{Mq} have types *R32F* and *RGBA32F* respectively

Output: Final scene rendered in F_1

// We are essentially doing the same algorithm, just with one additional render pass and different weight function

1: Do steps 1-7 of the *WBOIT* algorithm
2: $D_M \leftarrow D_1$
3: $C_{Ms} \leftarrow 0$
4: $C_{Mq} \leftarrow (0, 0, 0, 0)$
5: Enable blending and set blending operator to $C_f = C_s \cdot 1 + C_d \cdot 1$
// Capture moments of transparent geometry for each pixel on the screen
6: Render all transparent geometry into F_M and capture moments z, z^2, z^3, z^4 into C_{Mq} and their sum into C_{Ms} as $C_{Mq}, C_{Ms} = S_{moment}(Z)$ where Z is the depth of the rendered transparent pixel
7: Do steps 8-11 of the *WBOIT* algorithm
8: Do step 12 of *WBOIT*, but calculate the weight function by S_{weight} shader instead of $W(z, \alpha)$. The resulting weight is $S_{weight}(C_{Mq}, C_{Ms}, Z)$, where Z is the depth of the currently rendered transparent pixel
9: Do rest of the steps just like *WBOIT*

3 Results

In this chapter, we discuss both the visual results and performance of implemented algorithms. We decided to compare all algorithms across multiple categories of rating to show their weaknesses and strengths in several situations. We will compare the results of all algorithms to the *depth peeling* algorithm with 32 passes (or sometimes 48 passes in cases of a complex scene), because it produces the same result as if all surfaces were rendered from back to front.

3.1 Final color

Let's discuss first the accuracy of the final color calculation and and how it differs from the result we could have achieved if individual triangles of the scene were rendered from back to front (or rendered by *depth peeling*).

We compared the visual look of algorithms across 2 different scenes with slight variations (for instance, placing smoke clouds around the scene). The scenes used were

- *sponza scene* - a famous architectural building used for testing of rendering algorithms
- *ball park* - our own scene composed of multiple balls with varying colors and opacities

We should note that we rendered leaves in the *sponza scene* as if they were partially transparent object, which is not that common in practical cases where the vegetation is drawn by different techniques. We did this to show artifacts produced by the algorithms if several contrasting transparent objects would be occluded by each other. If the vegetation was rendered properly, the problem would still remain present for objects like pieces of glass for example.

The most notable weakness of the *weighted sum* algorithm is the underflowing of colors. This happens mostly in environments where multiple transparent surfaces with higher alpha values are covered with each other as can be seen in figure3.1. The resulting image looks overburned as the *weighted average* subtracts the color of background from the sum of transparent surfaces instead of properly blending them using alpha values. The greater the alpha of the transparent surfaces, the more will *weighted sum* subtract. This might in some cases even result in black colors.

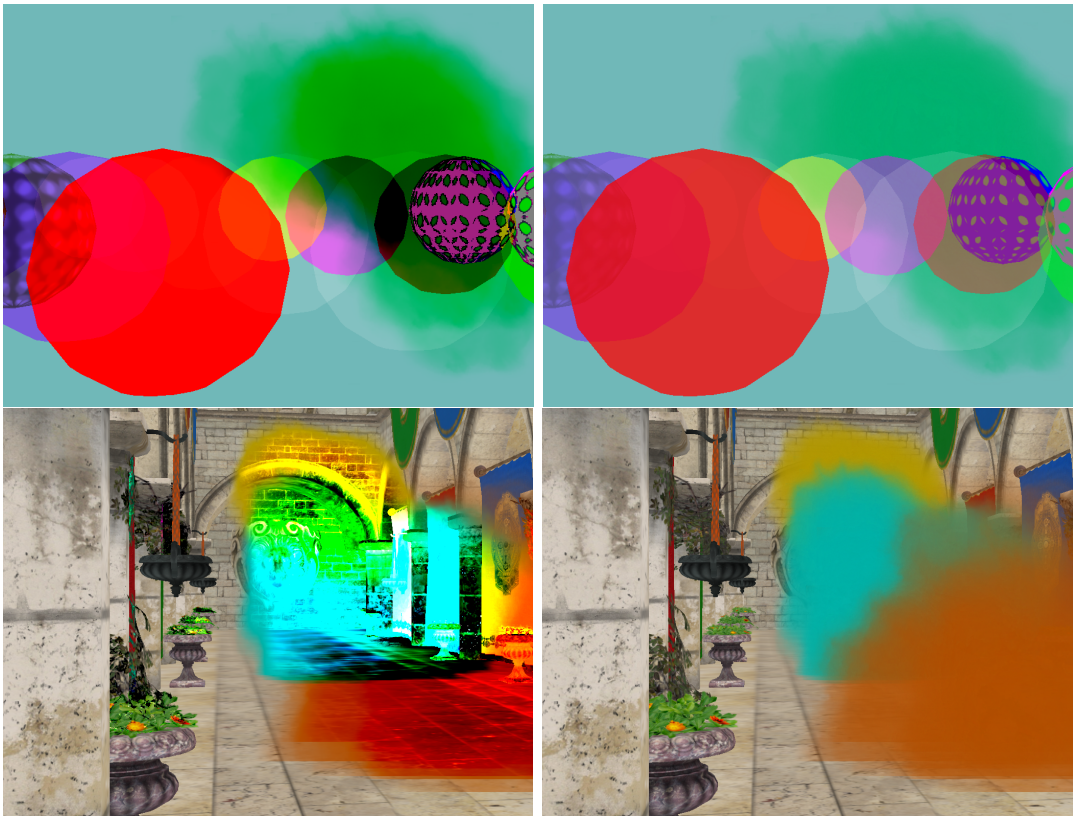

Figure 3.1 If multiple transparent surfaces overlap each other, the *weighted sum*(left) algorithm produces burned or even black colors. Sorted-like image result achieved using *depth peeling*(right).

A certain divergence from the final color can be also spotted in images produced by the *weighted average* algorithm. It is not so apparent in cases when transparent surfaces have low alpha, because the average is weighted towards pixels with greater opacity. This still produces a small error, however, it is not usually that apparent as can be seen in figure 3.2.

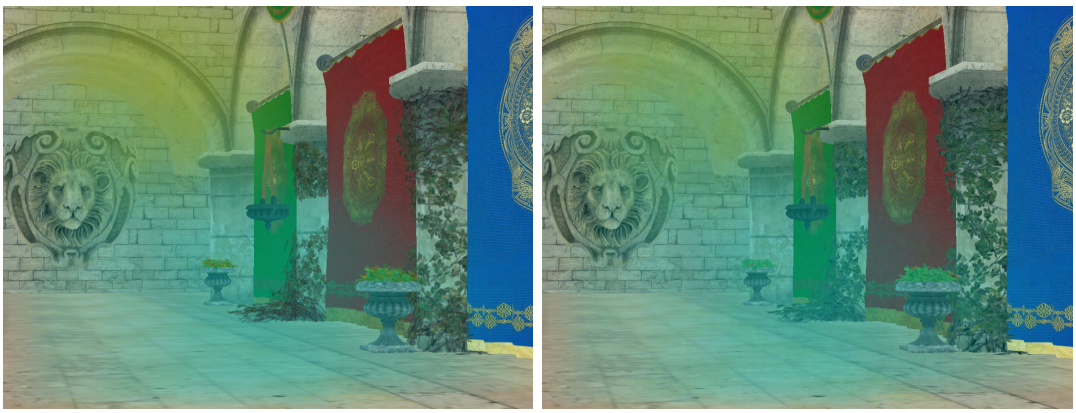

Figure 3.2 Sponza scene with two low opacity smoke clouds rendered by *weighted average*(left) and *depth peeling*(right). The difference is not very notable.

However, due to averaging of alpha and color values of all overlapped surfaces instead of blending them from back to front the algorithm produces incorrect results in cases where multiple surfaces with similar alphas cover each other as can

be observed in figure 3.3. This is most notable in form of overbleeding artifacts which we discuss more in the section 3.2.

Figure 3.3 The color of the background red smoke is clearly visible even though it should be fully occluded by the gray smoke in foreground. This is a common case of overbleeding. *Weighted average* on left, *depth peeling* with 32 passes on right

The averaging of transparent surfaces happens to be a problem for the *WBOIT* algorithm as well. Nevertheless, it is not so visible in most cases as *WBOIT* tries to combat this by utilizing a weight function which assigns more importance to closer surfaces. A notable improvement from *weighted average* can be observed in figure 3.4.

In ideal conditions this completely solves the problem, although the weight function might in some cases overestimate or underestimate the coverage, resulting in an image that is slightly off. Most notable case of this is when the weight function assigns too much weight to close surface, resulting in poor visibility of background surfaces as can be seen in figure 3.7.

Figure 3.4 *WBOIT* gives much more weight on the smoke in foreground, yet, the result is still not perfect and overbleeding of vegetation can be spotted(left). *Depth peeling*(right)

Even though the weight function is quite helpful, it is sometimes difficult to control. Tweaking it to work well for a certain environments might cause it to underperform in different conditions as can be seen in figures 3.6 and 3.5. It might also take an non-trivial time to find a fitting weight function for a given environment.

Figure 3.5 In the right side of the image, both the hanging vines and leaves in the pot are not properly occluded by the transparent blue smoke. This is due to them getting assigned a small importance from the used weight function ($W = 1/z^5 \cdot \alpha$) - *WBOIT*(left). *Depth peeling* (right).

The weight function might in some cases underperform, but overall it is better having a slightly underperforming weight function that having none at all.

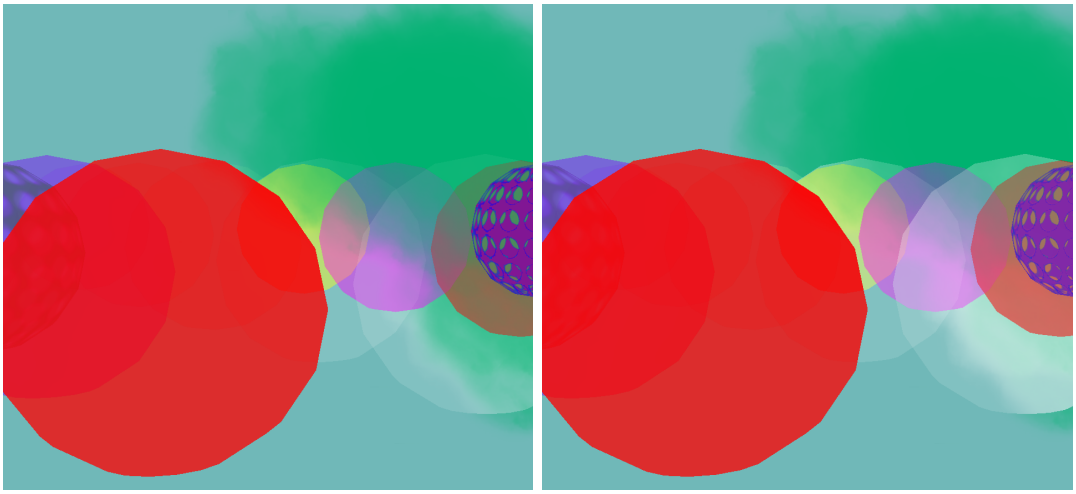

Figure 3.6 WBOIT with 2 different weight functions, each one producing different results. $W = 1/z^3 \cdot \alpha$ on the left, $W = 1/z^5 \cdot \alpha$ on the right. Note the visible difference in the almost completely transparent white ball on the right side of the images.

MBOIT solves the problem of carefully choosing a weight function for a certain scene by computing it itself. This results in a very believable images in a lots of situations as can be seen in figure 3.8. On average its results are closest to the *depth peeling* as can be seen in figure 3.7.

Having said that, the *MBOIT* has it's limits as well. It estimates the weight function correctly only in cases where 2 or less transparent surfaces overlap each other (as stated in the source paper[19]). For greater number of surfaces the weight function is only approximated and might sometimes perform even worse than *WBOIT* as can be seen in figure 3.9.

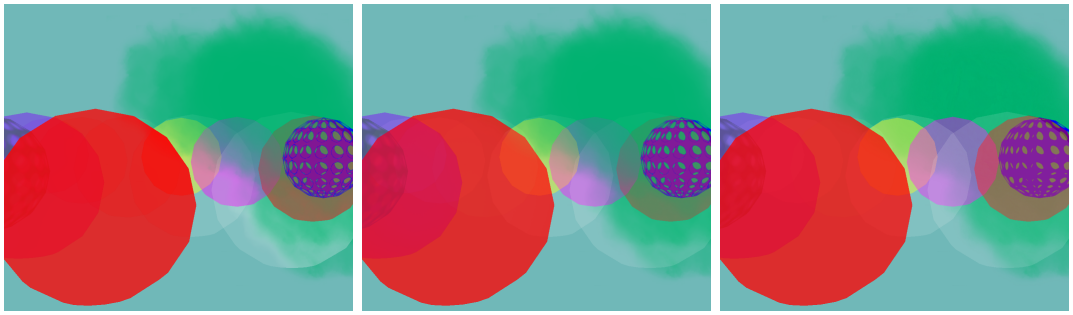

Figure 3.7 Ball Park scene. Notice how WBOIT(left) (with $W = 1/z^3 \cdot \alpha$) almost completely covers the balls behind the red one on the left. MBOIT(middle) computes coverage more accurately and covered balls have almost the same visibility as in case of depth peeling(right)

Figure 3.8 *MBOIT*(left) produces pretty convincing result even in environments with multiple transparent objects. Note the imperfection in form of vegetation overbleeding to the foreground from behind the yellow smoke. *Depth peeling*(right)

Figure 3.9 Sponza scene with smokes of multiple colors. A slight overbleeding of background vegetation can be seen. MBOIT on left, WBOIT with a weight function $1/z^5 \cdot \alpha$ on the right

In it's best with weight function precisely chosen for a given scene, the *WBOIT* will outperform *MBOIT*. In spite of that, the *MBOIT* will usually come on top in a common scene with varying alpha values, colors and distances between objects without a need to do any preparation or precalculation of it's weight function.

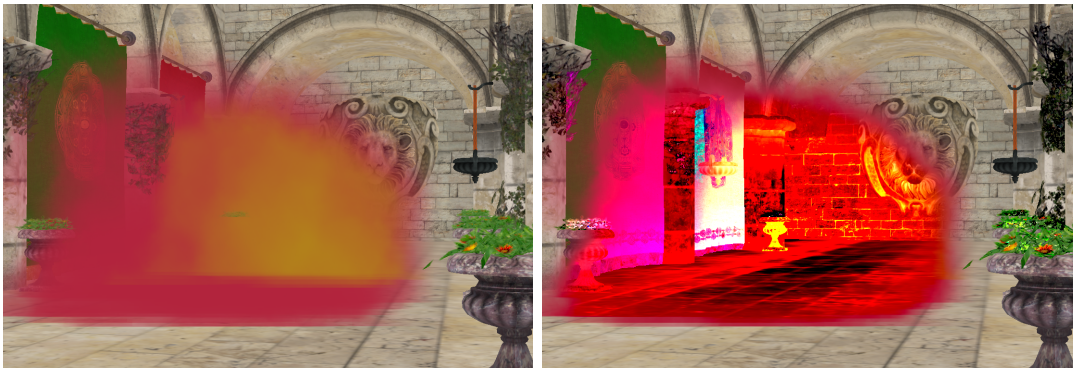

Figure 3.10 Overbleeding of vegetation and a yellow smoke from behind the red one in*weighted average*(left). *Weighted sum* overbleeds much more and reveals even the solid background.

3.2 Overbleeding and Image Stability

Overbleeding is a rendering artifact produced by some of the transparency algorithms. It is caused by blending the transparent foreground differently with transparent background then the solid background.

Since overbleeding is a frequent occurrence in all of the blended OIT algorithms and is usually the source of the most notable rendering artifacts we decided to dedicate an entire subsection to it.

By image stability we mean how much the resulting image changes for individual methods if we move the camera around or rescale the objects in the scene.

We decided to include this metric in our results because an algorithm might perform very well in a very specific conditions, but moving the camera or objects around might reveal significant rendering artifacts. This case might be the most notable in video games where the player might move around quickly.

Overbleeding usually comes hand in hand with image instability, therefore we decided to discuss these two phenomenon together in one subsection.

Both *weighted sum* and *weighted average* algorithms are not very resistant against overbleeding since they do not allocate more importance to surfaces closer to the camera. This might cause significant overbleeding artifacts as seen in figure 3.10. Both of them are, however, very stable and produce similar results when moving around the scene.

On the other hand, a lack of a weight function also results in almost 100% image stability. No matter the distance from the object, it's scale or rendering order, the result remains the same.

WBOIT does not suffer that much from overbleeding as previous 2 algorithms. Thanks to the weight function, it is able to produce an image without any overbleeding in ideal conditions. However, conditions might not always be ideal.

One problem of the *WBOIT* algorithm is its instability with moving around the scene or simply rescaling it. The weight function is usually quite sensitive to changing distances between objects or their scale as can be seen in figure 3.11.

Figure 3.11 Moving just a few meters back from the smoke reveals overbleading on distant plants on the right.

For this reason we experimented with *adaptive distance* in some of our weight functions. In certain scenarios, this improves the quality of the resulting image as seen in figure 3.12.

On the other hand, adaptive distance causes problems in other scenarios. Most notably in low alpha environments. Removing the weight function is also an option how to make the algorithm more stable. Moving around the scene will no longer cause drastic changes in result, although the overbleeding will become much more visible.

MBOIT algorithm outperforms *WBOIT* in most cases with both stability and a lack of overbleeding as seen in figure 3.13. Both of these problems are still present, but in much less cases as the *MBOIT* calculates the weight function on the runtime and adapts it to the current environment.

Our statement from the previous subsection applies here as well. For a given scene, there always exists a perfect weight function using which the WBOIT will outperform the *MBOIT*. However, searching for such a function might be difficult or straight up impossible in dynamic scenes where the environment changes from frame to frame (moving particle systems for example).

3.3 Performance

Performance of individual algorithms is important for real time rendering. In certain cases, we might prefer algorithm with inferior image quality but a better runtime performance. For this reason, we also discuss how each algorithm performs in certain scenarios.

For benchmarking, we picked 3 different scenes

- *transparent sponza*: a fully transparent sponza scene. No objects in this scene are opaque
- *sponza with smokes*: default opaque sponza scene with multiple transparent smoke clouds
- *ball park*: scene with several balls of different colors and alphas and one particle cloud

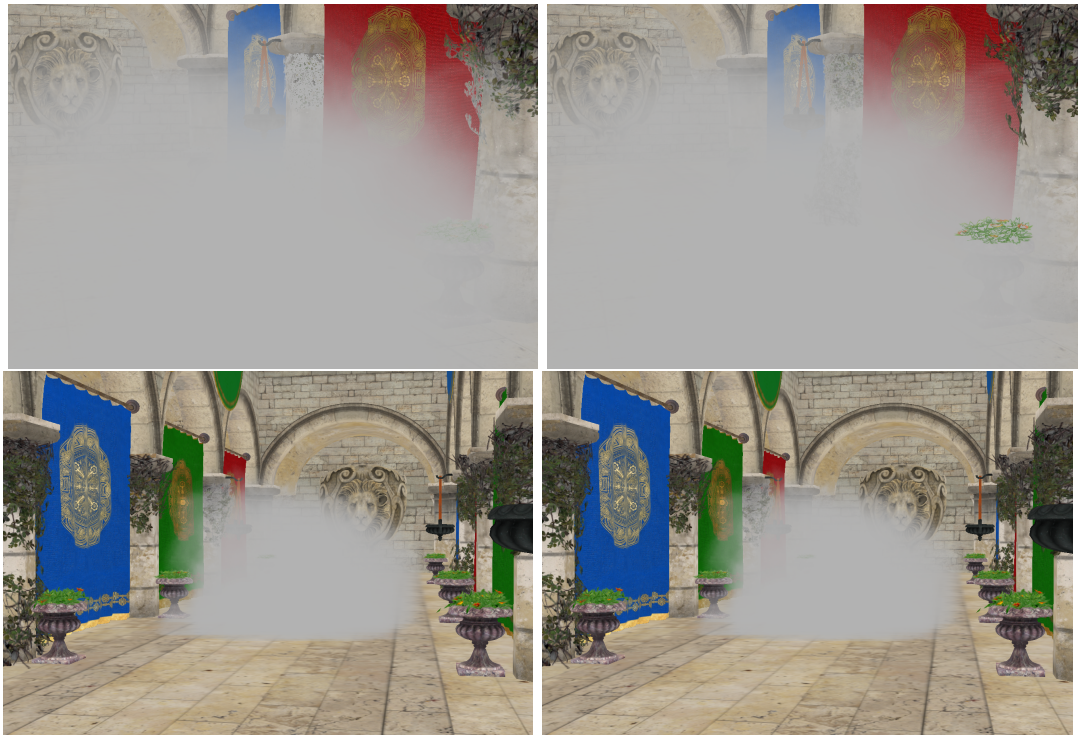

Figure 3.12 Adaptive Z improves the algorithm in the short distance without making it worse for further distances. Adaptive weight function $w = 1/adapt(z)^5 \cdot \alpha$ on the left and a default weight function $w = 1/z^5 \cdot \alpha$ on the right. Notice the overbleeding of vegetation on short distances with the default weight function

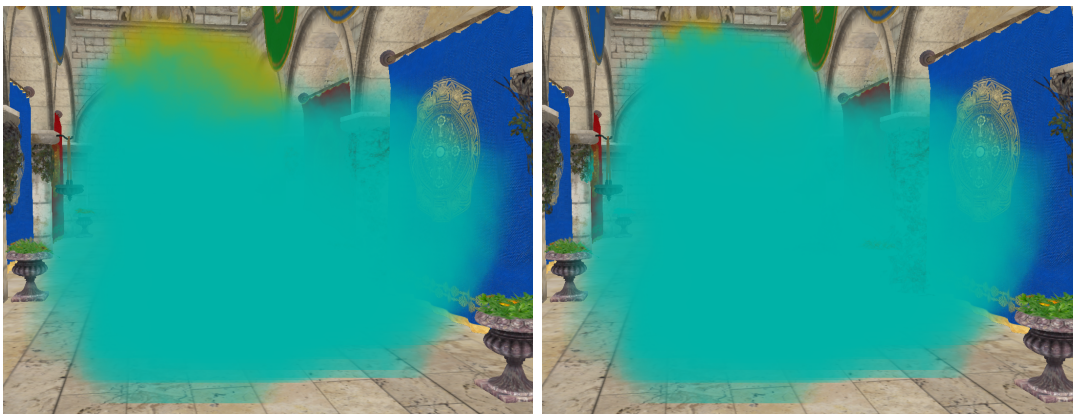

Figure 3.13 *MBOIT*(left) produces much less visible overbleeding on the plants in the right part of the image. *WBOIT*(right)

Performance Measure			
	Scene		
OIT Algorithm	Sponza Transparent	Sponza Smokes	Ball Park
Everything Opaque	1.07ms	1.05ms	0.51ms
Weighted Sum	1.85ms	1.22ms	0.65ms
Weighted Avg.	3.03ms	1.53ms	0.83ms
WBOIT 16F	3.49ms	1.56ms	0.90ms
WBOIT 32F	9.34ms	2.6ms	1.75ms
MBOIT	18.18ms	4.21ms	3.25ms
Depth Peeling (32)	31.25ms	14.92ms	9.43ms

Table 3.1 Benchmarked Performance of Transparency Rendering Algorithms

Figure 3.14 Benchmarked scenes. Transparent sponza, sponza with smokes and ballpark from left to right

We also measured performance for rendering the given scene as fully solid and opaque with no transparency and rendering it with *depth peeling* algorithm, both for comparison reasons.

All benchmarks were executed on Acer Nitro AN515-54 laptop with 16GB RAM, Intel Core i5-9300H CPU @ 2.4GHz and NVIDIA RTX 2060 GPU with Windows 10 operating system and 1920x1080 screen.

From the benchmarks 3.1 we can infer that the *weighted sum* algorithm outperforms all other. It renders the scene somewhat slower compared to rendering it fully opaque, which makes sense because the geometry for the whole scene has to be rendered and blended instead of only rendering the pixels closest to the camera.

Notice *WBOIT 16F* and *WBOIT 32F*, which are 2 versions of the *WBOIT* algorithm. The first one uses no weight function and is therefore able to use 16bit floating point color buffer instead of the 32bit one. Run times of both *weighted average* and *WBOIT 16F* closely match each other as their number of rendering passes is essentially the same and both of them use the same number of framebuffers and have color components of the same size.

MBOIT algorithm introduces a significant drop in the performance due to increased amount of render passes required and more complex calculations. We

Memory Consumption		
Algorithm	Per 1920x1080	Per Pixel
Depth peeling	49,766MB	24B
Weighted sum	58,06MB	28B
Weighted avg. 16F	62,208MB	30B
WBOIT 16F	62,208MB	30B
WBOIT 32F	82,944MB	40B
MBOIT 32F	116,121MB	56B

Table 3.2 Memory Consumption of Individual Transparency Algorithms

should note that the source papers[19][20] are aware of the performance drops and offer solutions in the form of decreased resolution or quantized moments. These optimizations increase the performance of the algorithm, but on the other hand decreases the image quality. We have not experimented with any of these optimizations.

The *depth peeling* algorithm comes out as the slowest one, which is expected as it has to render the transparent geometry in 32 passes. As we mentioned before, there are multiple proposals [11][12] that increase the performance of this algorithm, but we have not implemented any of these as it is outside bounds of this thesis.

3.4 Memory Consumption

We decided to compare memory footprint of all implemented algorithms as can be seen in table 3.2. We only state the amount of consumed GPU video memory as the RAM memory consumption is small and might depend on the software architecture of the rendering program.

We state the amount of memory our implementations of the algorithms consume. It might be possible to implement each algorithm using smaller amounts of memory. We simply did not attempt to optimize for the smallest possible memory consumption as the amount of memory on video cards is usually quite large compared to the amount required by individual implementations.

We should also note that the default framebuffer of a window usually consumes 24-32bits per pixel (depending on the presence of the alpha channel). Our numbers include the size of the default framebuffer as well (we assume the size of default framebuffer to be 32bits per pixel).

Notice that for certain algorithms in table we also stated the precision of floating point numbers we used - *16F* for 16bit floating point precision and *32F* for 32bit colors.

The *weighted average* algorithm uses floating point color component. We experimented both with using both 16bit and 32bit floating point colors and observed no difference in the result.

Converting *MBOIT* to lower precision format is not so straightforward, because the number precision might cause rounding of numbers close to 0 down, resulting

in possible division by 0. Source papers for *MBOIT*[19][20] offer certain solutions to decreasing a memory footprint, which are however out of bounds of this thesis and were not implemented.

Conclusion

After presenting the results we arrive at the conclusion chapter. We implemented a demo application capable of loading and modifying various scenes and rendering them with 5 different transparency rendering algorithms. The application is capable of switching the OIT algorithms at runtime, rendering of a simple interface and manipulation of the objects in the scene. We provided multiple examples of images rendered using various OIT algorithms, compared their quality and analyzed performance in different scenarios and their memory consumption.

Now, we will sum up the results and try to provide recommendation which of the tested order independent transparency algorithms to use and when.

We will begin with *weighted average* and *WBOIT* algorithms.

The performance of both algorithms is almost the same(if both of them use the same type of color format), yet *WBOIT* excels in the quality of the result as it does not produce as much overbleeding as *weighted average*. Even if we use *WBOIT* with no weight function (e.g. weight function $W(z, \alpha) = 1$) it still calculates the background coverage more precisely.

In section 3.2 we have shown that the *weighted average* is much more stable. However, using *WBOIT* without any weight function will give it the same stability as *weighted sum* has. This applies to the memory footprint as well - the version of *WBOIT* with no weight function can be implemented using the same amount of memory as the *weighted average*.

From a programmer's viewpoint, the implementation of both of these algorithms is practically the same - both require the same amount of transparency passes from the rendering engine, same amount of framebuffers and the same amount of render targets. Therefore, we arrive at conclusion that *WBOIT* should be preferred over *weighted average* as it is better in all aspects.

The *weighted sum* produces the worst image results from all of the blended OIT algorithms we tested. The exception to this are environments with low amount of overlapping transparent surfaces which have low alpha values. Nonetheless, it beats all of the remaining algorithms in the runtime speed and also has quite low memory requirements. It is very simple to implement and is the only blended OIT algorithm that does not require support for multiple render targets and can be therefore implemented on an older hardware.

We, therefore, recommend using *weighted sum* algorithm only in cases where the performance is critical, the amount transparent surfaces is limited and their alpha is low and sorting the transparent geometry or using any other blended OIT algorithm is not an option. In other cases we recommend using some other algorithm with results of higher quality.

On the other hand, the *MBOIT* algorithm is the exact opposite of the *weighted sum*. In most cases it produces results of high quality that are often unrecognizable from the sorted rendering by a human eye, although for a cost of a lower performance. It is more stable than *WBOIT* and produces less overbleeding artifacts in most scenarios. Its implementation is slightly more difficult, because it requires

one extra render pass for the transparent geometry than some other algorithms. For this reason, implementing it into an already established rendering pipeline might not be the most trivial task.

We recommend using *MBOIT* for applications where performance is not critical (on a current generation hardware for example), the rendered scene is not purely static and there is enough free space in the video memory. For a practical use it is recommended to implement some of the proposed optimizations[19][20] that increase performance.

Future Work

Investigate Optimizations Of MBOIT Algorithm

The source papers [19] and [20] offer a few optimizations that claim to increase the performance of the *MBOIT* algorithm. One of the proposed optimizations is downscaling the moment framebuffer to 1/4 of the original size. One possible area of investigation would be to analyze how much impact on the visual result does this one optimization have and if it is worth it.

Bibliography

1. Akenine-Möller, Tomas; Haines, Eric; Hoffman, Naty; Pesce, Angelo; Iwanicki, Michał; Hillaire, Sébastien. *Real Time Rendering: Fourth Edition*. CRC Press, 2018. Page 24.
2. Akenine-Möller, Tomas; Haines, Eric; Hoffman, Naty; Pesce, Angelo; Iwanicki, Michał; Hillaire, Sébastien. *Real Time Rendering: Fourth Edition*. CRC Press, 2018. Page 795.
3. *How Northlight makes Alan Wake 2 shine* [`https://www.remedygames.com/article/how-northlight-makes-alan-wake-2-shine`]. [N.d.]. Accessed online 8.5.2024.
4. Kohler, Johan. Practical Order Independent Transparency. *Activision Research*. 2016.
5. Wyman, Chris. Exploring and Expanding the Continuum of OIT Algorithms. *High Performance Graphics*. 2016.
6. Smith, Alvy Ray. Alpha and the History of Digital Compositing. *Independent*. 1995.
7. *The OpenGL® Graphics System: A Specification (Version 4.6 (Core Profile) - May 5, 2022)*. 2022. Accessed online 8.5.2024, page 210.
8. *The OpenGL® Graphics System: A Specification (Version 4.6 (Core Profile) - May 5, 2022)*. 2022. Accessed online 8.5.2024, page 511.
9. Cook, Robert L.; Carpenter, Loren; Catmull, Edwin. The Reyes image rendering architecture. *ACM SIGGRAPH Computer Graphics*. 1987.
10. Everitt, Cass. Interactive Order-Independent Transparency. *Technical report NVidia, 2001*. 2001.
11. Bavoil, Louis; Myers, Kevin. Order Independent Transparency with Dual Depth Peeling. *Technical report NVidia, 2008*. 2008.
12. Liu, Baoquan; Wei, Li-Yi; Xu, Ying-Qing. Multi-Layer Depth Peeling via Fragment Sort. *IEEE*. 2009.
13. Enderton, Eric; Sintorn, Erik; Shirley, Peter; Luebke, David. Stochastic Transparency. *IEEE*. 2011.
14. *GTA V - Graphics Study* [`https://www.adriancourreges.com/blog/2015/11/02/gta-v-graphics-study/`]. [N.d.]. Accessed online 8.5.2024.
15. Adaptive Transparency. *ACM Symposium on Interactive 3D Graphics and Games*. 2011.
16. Multi-Layer Alpha Blending. *ACM Symposium on Interactive 3D Graphics and Games*. 2014.
17. Meshkin, Houman. Sort-Independent Alpha Blending. *GDC Session, 2007*. 2007.
18. McGuire, Morgan; Bavoil, Louis. Weighted Blended Order-Independent Transparency. *The Journal of Computer Graphics Techniques*. 2013.

19. Sharpe, Brian. Moment Transparency. *Proceedings of the Conference on High-Performance Graphics*. 2018.

20. Münstermann, Cedrick; Krumpen, Stefan; Klein, Reinhard; Peters, Christoph. Moment-Based Order-Independent Transparency. *In Proceedings of the ACM on Computer Graphics and Interactive Techniques 1:1 (Issue for the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games 2018)*. 2018.

21. Christoph Peters, Reinhard Klein. Moment shadow mapping. *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games*. 2015.

A User Documentation

In this attachment we briefly describe how to use the demonstration application used in this thesis.

How To Run

The demo application can be started by running the `OIT.exe` executable included as an electronic attachment to this thesis, see attachment **??** for a description of its contents.

Recommended Requirements

The application can be run on a Windows 10 operating system with a GPU supporting OpenGL 4.3.

Usage

In the following subsection we describe briefly how to use the demo application.

- Use W/S/A/D keys for horizontal movement
- Use E/Q/C/Space keys for vertical movement
- Hold right mouse button simultaneously with moving mouse to rotate the camera around

Notice a menu in the top of the screen with a *Windows* button. Clicking on the button will reveal a popup menu consisting of 3 items just as the one in figure A.1.

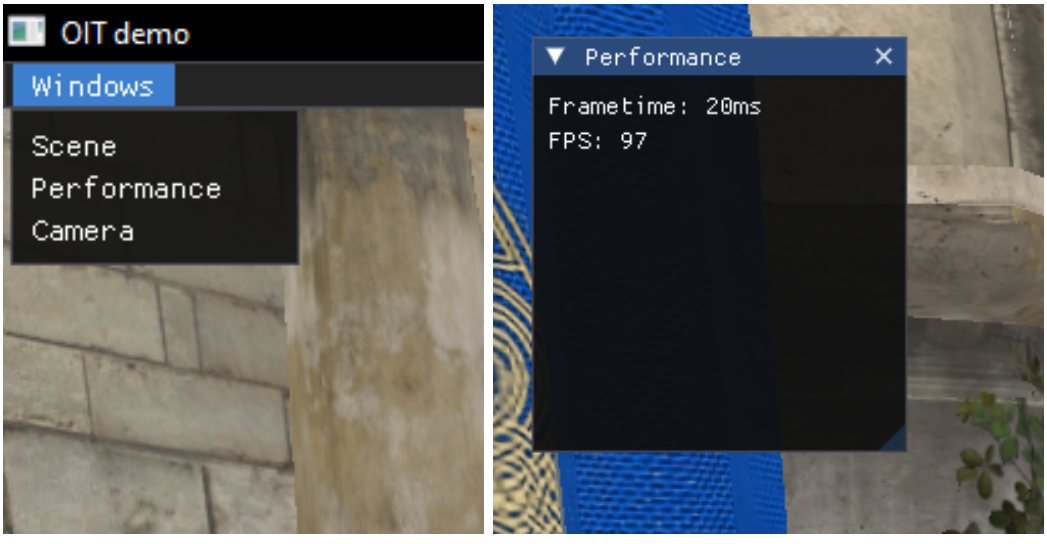

Figure A.1 Window Popup Menu and Performance window

Clicking on any of the popups will create a separate window somewhere on the screen. Each window can be closed by clicking on the popup item again or on the cross on the top right side of the window.

Performance window as seen in figure A.1 shows the FPS averaged over the last second and a current frametime in milliseconds rounded up. It can be used for benchmarking.

Camera Window Settings allows the user to tweak the camera sensitivity, movement speed and field of view. See it in figure A.2.

Figure A.2 Camera Window

Scene Window shown in figure A.3, is the most complicated UI window we offer. It enables the user to change between implemented transparency rendering algorithms using dropdown menu `Transparency rendering method`. These can be also changed between using F1 to F6 keys, where

- `F1` selects *solid only* rendering, which renders all objects as fully opaque
- `F2` selects *depth peeling* algorithm
- `F3` selects *weighted sum* algorithm
- `F4` selects *weighted average* algorithm
- `F5` selects *WBOIT* algorithm
- `F6` selects *MBOIT* algorithm

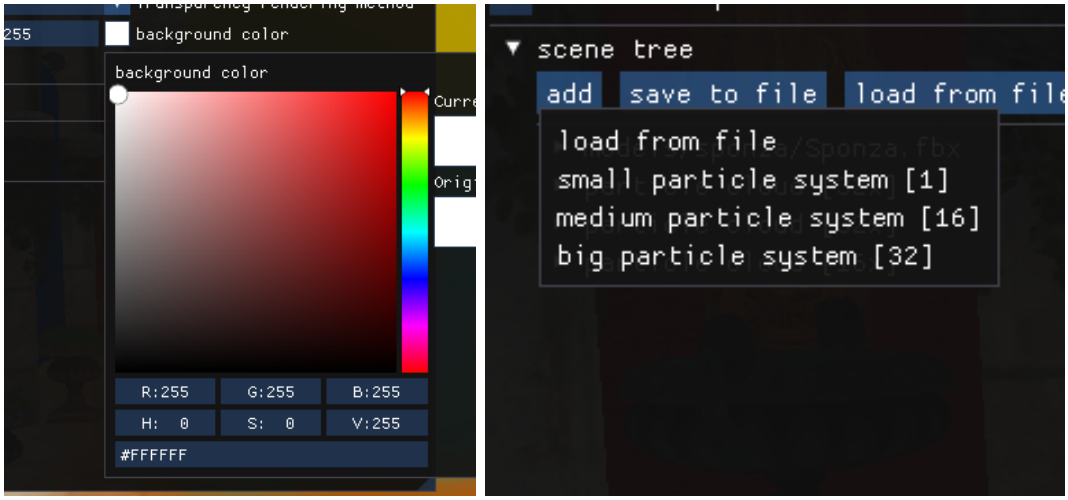

Figure A.4 Changing background color (left) and adding object to the scene (right)

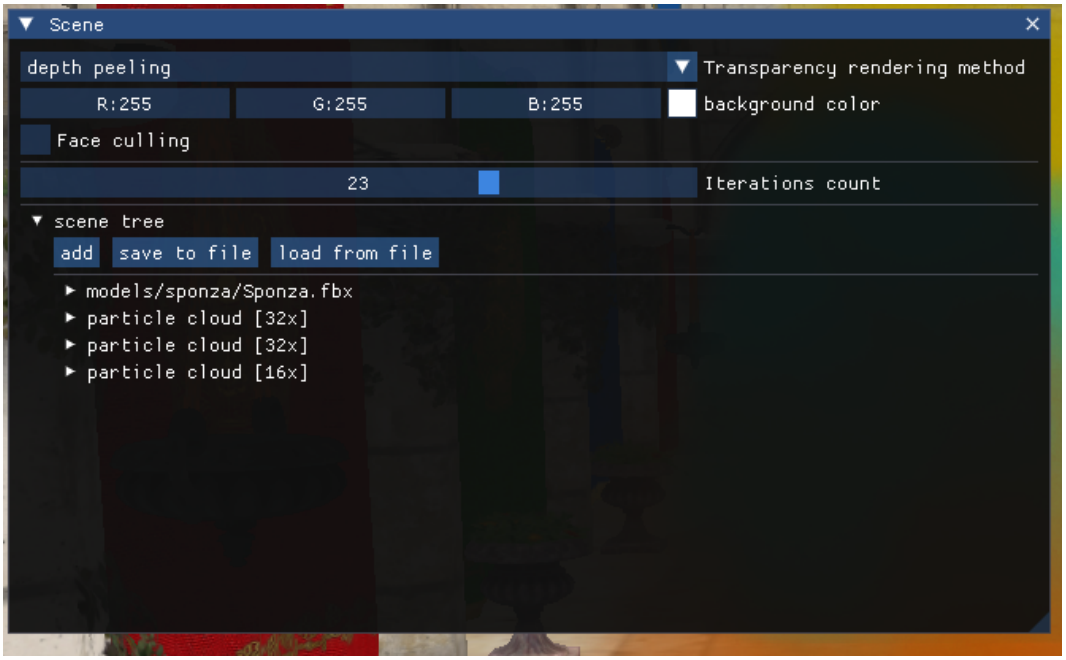

Figure A.3 Scene Window

It is also possible to change background color of the scene by tinkering with `Background color` color picker or directly editing RGB values with sliders next to it.

Face culling of the geometry in the scene can be turned on/off with checkbox `Face culling` under which a few UI items might be available depending on the type of selected rendering algorithm. Each algorithm offers various UI elements that offer a certain control over the rendering. For instance, the *depth peeling* algorithm gives user a slider through which the number of passes of the algorithm can be specified.

In the bottom part of the `Scene` window, a `scene tree` menu can be seen. The menu consists of 3 buttons - `add`, `save to file` and `load from file`.

Clicking on the first button gives the user an option to add a certain object

to the scene. It can be either a model saved on the disc or a particle system of a certain size. We support *.fbx* model format. The textures for the given model should be stored next to it in the same directory.

Next two buttons give the user option to either save the current scene to a *.json* file or load it from one. This way a certain scene consisting of multiple objects can be prepared in advance, saved into a file and loaded during the next runtime of the application.

If user decides to load a scene during runtime, the current scene is discarded and instead replaced by a new one. The application loads a scene named "*default_scene.json*" on each startup.

Below the three buttons, a list of objects in the scene is drawn. After clicking on a triangle of an appropriate object a set of UI elements giving control over the specified object is shown as seen in image A.5. Here, the user might change the position or scale of the object. The color or alpha of the object can be changed as well. This way objects that are opaque by default can be made transparent. Clicking on a `delete` button will destroy the given object.

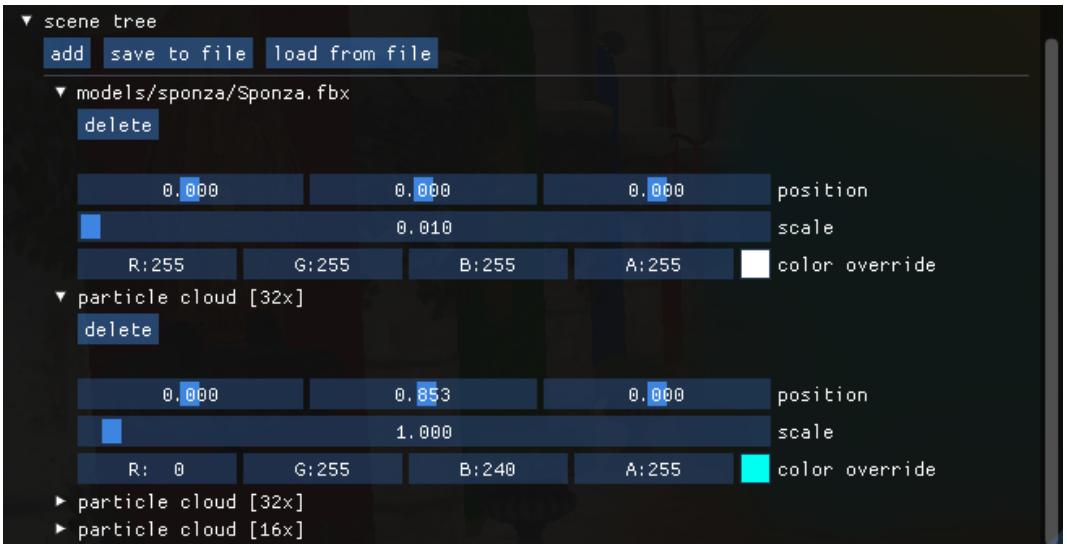

Figure A.5 Each object can be modified or even deleted

B Developer Documentation

In this section we will explain the structure of our code, point out the most interesting files and how is the whole codebase structured.

This section assumes basic knowledge of programming in C++ and understanding of how realtime rendering works.

How To Compile And Run

The code was compiled using C++20 version of the language and MSVC compiler.

To compile and run use the attached Visual Studio 2022 Solution `/OIT.sln`.

Dependencies and Libraries

For development of the demo application we used the following C++ libraries

- `SDL` for creation of the window and OpenGL context
- `OpenGL` rendering API for rendering on GPU
- `GLAD` for OpenGL function bindings
- `ImGui` for creating a simple user interface
- `Nlohmann Json` for parsing JSON files used for scenes
- `STB image` for loading *.png* files used as textures
- `GLM` for 3D linear algebra
- `Assimp` for loading 3D models in various file formats
- `Tiny File Dialogs` for cross platform implementations of open/save file popups

Source code for all of the above mentioned libraries is included in the `/dependencies` directory. No additional dependencies have to be installed in order to compile the project on Windows using Visual Studio.

Most Notable Files

All header files are located in directory `/source/include/` and their implementation can be found inside `/source/source/`.

List of the most interesting files follows. All files are relative to the directory `/source/`.

- `/source/main.cpp` contains the entry point of the program
- `/source/demo_app.cpp` and `/include/demo_app.h` contain source code of the demo application

- `/include/common.h` contains definitions for macros and types used across the whole program
- `/include/camera.h` and `/source/camera.cpp` contain functionality for the 3D world camera
- `/include/shaders.h` and `/source/shaders.cpp` contain helper functions for compiling and linking GPU programs used for rendering
- `/include/geometry.h` and `/source/geometry.cpp` contain helper functions for loading of 3D scenes from files
- `/include/render_object.h` and `/source/render_object.cpp` contain functionality for representing the objects inside the world

Implementation Of Transparency Rendering Algorithms

Source code for individual algorithms is located inside `/Source/Include/-Algorithms/` and `/Source/Source/Algorithms/`. Each of the implemented algorithms is located inside of it's own file.

`/Source/Include/Algorithms/` and `/Source/Source/Algorithms/` contain the following files:

- `common.h` and `common.cpp` contain the implementations of functionality shared across all algorithms
- `depth_peeling.h` and `depth_peeling.cpp` implement the *depth peeling* algorithm
- `meshkin.h` and `meshkin.cpp` contain implementation of *weighted sum* algorithm
- `moment.h` and `moment.cpp` contain implementation of the *MBOIT* algorithm
- `solid_only.h` and `solid_only.cpp` do not contain any transparency rendering algorithm. Instead, functions inside these files are used for rendering of the opaque only geometry in the scene
- `wboit.h` and `wboit.cpp` contain implementation of *WBOIT* algorithm
- and `weighted_average.h` with `weighted_average.cpp` contain implementation of the *weighted average* algorithm

All algorithms are wrapped inside the namespace `oit::algorithms` and each one has an individual namespace reserved for itself. For example, the whole functionality of *depth peeling* algorithm can be found inside the namespace `oit::algorithms::depth_peeling`.

Even though each algorithm works in a slightly different manner, they all share a similar interface. Each of our algorithms implements the following functions, which are then called from the main application in a certain circumstances.

- `init` is called during the intialization of the application. This function should initialize all data required for the runtime of the algorithm and return them to the application
- `terminate` is a function that should cleanup all resources of the given algorithm and prepare for application shutdown. It is called just before the application turns off
- `on_screen_resize` is called if the size of the application window changes. Each algorithm should resize it's framebuffers (if it uses any) to match the size of the main window
- `render_world` is called when the given algorithm is requested to render one frame of a scene
- `draw_interface` is called during drawing of *ImGUI* interface. This function allows each algorithm to define a set of adjustable parameters that can be visible inside the UI

Scene Representation

For rendering, we use a structure of data named *render object* which resides inside `/source/include/render_object.h` and `/source/source/render_object.cpp`. It consists of OpenGL handles for vertex buffer, index buffer and texture and size, position and rotation.

Since each *render object* can have only one texture, for rendering of a scene with multiple textures we have *render scenes*, which are each a set of *render objects* with additional position, rotation and scaling. The rendered world then consists of multiple *render scenes*.

We group objects into *render scenes* so that positioning a part of a scene or removing it is not difficult for the user. For instance, if the user wants to remove a particle cloud they created before then it is not necessary to remove each particle individually. Instead, only the *render scene* which contains all the particles is removed.

Render scenes are defined inside the same files as *render objects*.

Shaders

Shaders written in the GLSL language are located inside the `/shaders/` folder.

Each of the transparency rendering algorithms defines which shaders it uses inside of its header file.

For instance, the *depth peeling* algorithm requires following shaders (`/source/include/depth_peeling.h`):

Listing B.1 Example of shaders used by algorithm

```
constexpr cstr SOLID_SHADERS[] = {"shaders/solid.vert", "shaders/solid.frag"};
constexpr cstr PEEL_SHADERS [] = {"shaders/solid.vert", "shaders/peel.frag"};
constexpr cstr MERGE_SHADERS[] = {"shaders/full_screen_quad.vert",
    "shaders/peel_merge.frag"};
```

A helper functionality for compiling and linking shaders can be found inside the `/source/include/shaders.h` and `/source/source/shaders.h`.

Serialization

Some premade scenes are serialized to *.json* format in directory `/scenes/`.

(De)serialized scenes have the following *json* structure. The comments following after "//" are not a valid JSON and serve only as a description of the format.

Listing B.2 Deserialized scene example

```
{
  "back": [
    0.002739322604611516, // color of the background (R, G, B), normalized
    0.019094884395599365, // in interval [0, 1]
    0.5588235259056091
  ],
  // list of objects in the scene
  "objects": [
    {
      // color of the object
      "color_override": [
        0.0,
        1.0,
        0.0,
        1.0
      ],
      "file_path": "models/sponza/Sponza.fbx", // source file
      // position of the object
      "position": [
        0.0,
        0.0,
        0.0
      ],
      // rotation of the object
      "rotation": [
        0.0,
        -0.0,
        0.0
      ],
      // scale of the object
      "scale": 0.009999999776482582,
      // mesh is an object loaded from file
      "type": "mesh"
    },
    {
      "color_override": [
        0.9177013039588928,
        0.9411764740943909,
        0.14302192628383636,
        1.0
      ],
      // number of particles
      "count": 16,
      "position": [
        0.0,
        0.0,
        0.0
```

```
      ],
      "rotation": [
        0.0,
        -0.0,
        0.0
      ],
      "scale": 1.0,
      // particle is a type of an object that is not loaded,
      // but instead generated
      "type": "particle"
    }
  ]
}
```