TU Clausthal

ORDER INDEPENDENT TRANSPARENCY ACCELERATION

BACHELOR THESIS

presented by

FELIX BRÜLL

Computer Graphics Group
Department of Computer Science
Clausthal University of Technology

Felix Brüll: *Order Independent Transparency Acceleration*

STUDENT ID
464266

ASSESSORS
First assessor: Prof. Thorsten Grosch
Second assessor: Prof. Olaf Ippisch

SUBMISSION DATE
September 14. 2018

STATUTORY DECLARATION

Hiermit versichere ich, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe und dass alle Stellen dieser Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht wurden und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsstelle vorgelegt wurde.

Des Weiteren erkläre ich, dass ich mit der öffentlichen Bereitstellung meiner Abschlussarbeit in der Instituts- und/oder Universitätsbibliothek nicht einverstanden bin.

Die öffentliche Bereitstellung der PDF Datei auf Internetseiten der Universität ist gestattet.

Clausthal-Zellerfeld, September 14. 2018

Felix Brüll

ABSTRACT

Real-time transparency rendering for a high number of transparent objects on the GPU is still an open problem. Order-independent transparency algorithms provide an acceptable approximation for transparency in real-time. This work investigates Adaptive Transparency and Multi-Layer Alpha Blending, two of the leading order-independent transparency algorithms, and improves their performance by up to 30%. The techniques and data structures that were used to accelerate those algorithms can be used for similar algorithms as well.

ZUSAMMENFASSUNG

Das Darstellen von vielen transparenten Objekten in Echtzeit ist immer noch ein offenes Problem. Order-Independent Transparency Algorithmen bieten eine gute Annäherung für Transparenz in Echtzeit. Diese Arbeit untersucht Adaptive Transparency und Multi-Layer Alpha Blending, zwei der führenden Order-Independent Transparency Algorithmen, und verbessert ihre Leistung um bis zu 30%. Die Techniken und Datenstrukturen, die verwendet wurden, um diese Algorithmen zu beschleunigen, können auch für ähnliche Algorithmen verwendet werden.

CONTENTS

ACRONYMS

ALU	Arithmetic Logical Unit
AT	Adaptive Transparency
FPU	Floating Point Unit
GPC	Graphics Processing Cluster
HAT	Height Adaptive Transparency
LAAT	Linked Array Adaptive Transparency
LD/ST	Load/Store Unit
MLAB	Multi-Layer Alpha Blending
MSE	Mean Squared Error
OIT	Order-Independent Transparency
PC	Program Counter
ROP	Render Output Unit
SFU	Special Function Unit
SM	Streaming Multiprocessor
SSBO	Shader Storage Buffer Object
UAT	Unsorted Adaptive Transparency
UHAT	Unsorted Height Adaptive Transparency
UMLAB	Unsorted Multi-Layer Alpha Blending
VRAM	Video Random Access Memory

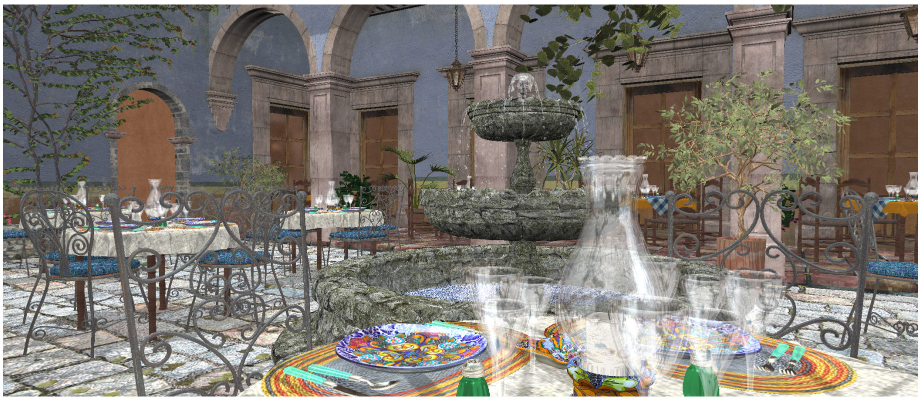


Figure 1.1: The San Miguel scene with transparent glasses. Courtesy of Guillermo M. Leal Llaguno.

1 INTRODUCTION

Real-time rendering is used in games and simulations to produce images for a 3D scenario with at least 25 frames per second. A dedicated device is required to produce high-quality images in real-time.

This device is called the graphics card, and it performs hardware accelerated rendering according to the graphics pipeline. The graphics pipeline (Section 2.1) is a widespread model which describes how to transform 3D geometry into a 2D image.

Most games and simulations are using transparent objects like glass (see Figure 1.1) or fog-particles. Unfortunately, rendering transparency with the graphics pipeline can be quite difficult. Since transparency rendering usually uses so-called alpha blending (Section 2.3), transparent geometry must be drawn in back to front order[1] or vice versa. Hence, it is called (geometry) order dependent. Figure 1.2 shows what happens if the geometry is not drawn in back to front order. In this example, objects from the background appear to be in the foreground (compare Figure 1.2 and Figure 1.3). However, sorting the geometry is too expensive for a real-time 3D application with many transparent objects. Therefore, a technique called Order-Independent Transparency (OIT) is required to draw transparent geometry without sorting the geometry beforehand. Unfortunately, a fast and correct solution for OIT is still an open problem.

[1] back to front from the camera's perspective

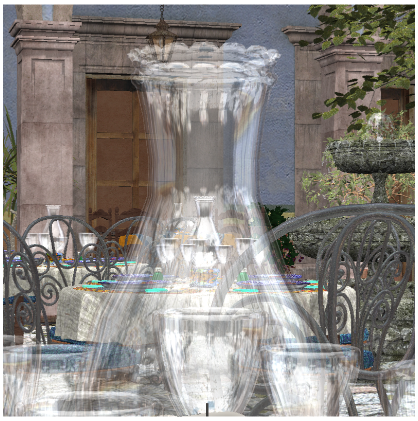


Figure 1.2: Alpha blending with unsorted geometry.

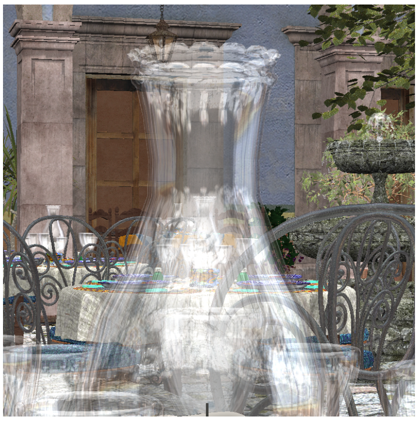


Figure 1.3: Alpha blending with sorted geometry.

This work investigates and improves Adaptive Transparency (AT) and Multi-Layer Alpha Blending (MLAB) which are two state of the art OIT algorithms. The next chapter introduces the fundamentals of real time transparency rendering on the GPU. Chapter three gives a brief overview about the evolution of OIT algorithms and explains AT and MLAB in detail. The fourth chapter presents several techniques to accelerate OIT algorithms in general, including storage optimizations, register usage and a comparison of sorting algorithms for lists with a fixed size. In chapter five, I modified MLAB by storing the per-pixel list nodes in an unsorted manner which accelerates the algorithm by up to 30%. Chapter six discusses a new and better heuristic for the AT node compression. Thereafter, a modified version of AT with the new compression heuristic is developed that stores the per-pixel list nodes in an unsorted manner and is up to 24% faster than AT.

2 FUNDAMENTALS

2.1 GRAPHICS PIPELINE

This section provides a simplified overview of the graphics pipeline on the GPU which is usually used to create real-time 3D applications.

In short: The application submits a 3D model to the GPU. On the graphics card, the geometry is first transformed and then projected onto a pixel grid (rasterization) for the 2D image. The GPU allows the application to customize several steps of this procedure. OpenGL provides an API for the graphics pipeline on the graphics card. The simplified OpenGL graphics pipeline is shown in Figure 2.1.

In the following, every step of this procedure will be explained in detail.

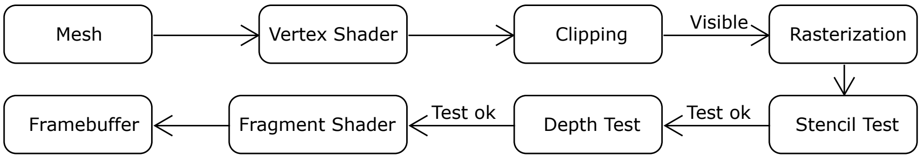


Figure 2.1: Simplified OpenGL pipeline.

Mesh

3D models are usually described with a list of triangles called mesh. The triangles describe the surface of an object (see Figure 2.2). The corner position with associated attributes is called vertex. Vertices usually hold information about the 3D position, surface normal and color of the triangle. An example for vertices is shown in Figure 2.3. V0, V1, V2 and V3 are four different vertices with a 3D position attribute. A triangle mesh describes how to connect vertices to form a 3D model. As shown in Figure 2.3, the vertex list (V0, V1, V2, V2, V1, V3) with V0-V1-V2 and V2-V1-V3 describes two separate triangles where some triangle vertices (e.g. V1 and V2) appear multiple times causing great data redundancy for large geometry.

A so called indexed representation is better for large meshes. The indexed version requires two buffers. The first buffer contains all vertices: (V0, V1, V2, V3). The second one consists of indices referring to vertices from the first buffer: (0, 1, 2, 2, 1, 3).

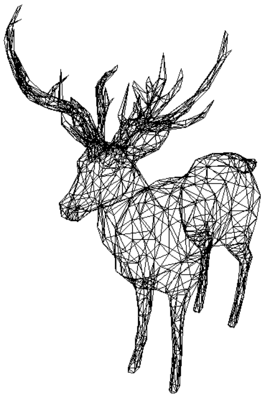


Figure 2.2: Deer represented by a triangle mesh.

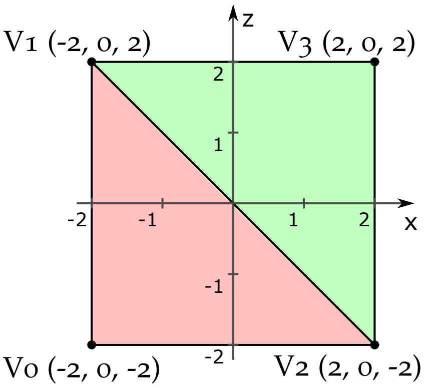


Figure 2.3: Triangle mesh that describes a quad.

Vertex Shader

After submitting the mesh to the GPU, the vertex shader is executed for each triangle vertex. Shaders are programs for the GPU written by the application programmer. The vertex shader typically transforms the vertices so that they can be projected onto a 2D image plane later.

OpenGL only draws geometry that appears in the canonical volume which is defined as $V := [-1, 1]^3$. The x-and y-position within the canonical volume determines the pixel coordinate of the 2D image projection, and the z-position represents the depth that is used for the depth test which will be explained later.

In order to transform the vertices into the canonical volume, each vertex is transformed through the following coordinate systems:

1. **Model space** (Figure 2.3): The coordinate system of the original mesh is called model space.
2. **World space** (Figure 2.4): This coordinate system describes how all objects are related to each other within the scene.
3. **Camera space** (Figure 2.5): This coordinate system represents the scene from the camera's perspective. The system origin is the camera position and the z-axis is the view direction of the camera. The area between the red rays describes the camera's field of view (FOV).
4. **Canonical volume** (Figure 2.6): The blue area in the camera space is called the view frustum. This frustum is the part of the actual field of view that will be transformed into the canonical volume. This transformation is not linear and distorts the geometry (see Figure 2.7).

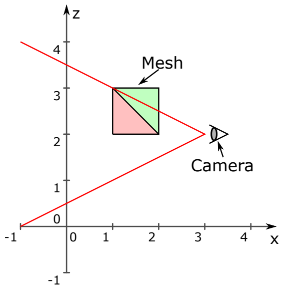


Figure 2.4: World space.

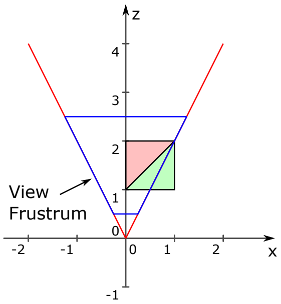


Figure 2.5: Camera space.

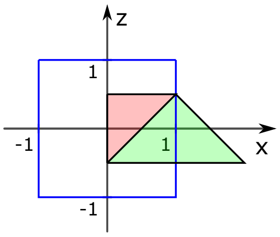


Figure 2.6: Canonical Volume.

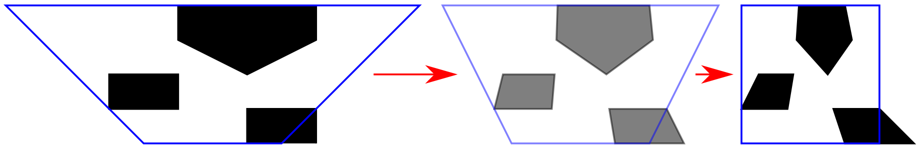


Figure 2.7: A perspective distortion occurs when the view frustum is transformed into the canoncial volume.

Additionally, the vertex shader can forward information to the fragment shader. Note that the vertex shader merely possesses information about the currently processed vertex and not about the entire triangle or mesh.

Clipping

In this stage, every triangle is clipped[2] into the canonical volume which can result in subdividing a triangle as shown in Figure 2.8. Clipping is performed to avoid computing pixel colors for pixels that are outside of the screen.

Rasterization

The rasterizer generates a fragment for each pixel of the color buffer that would be covered by the previously clipped triangle. The triangle covers a pixel if the pixel center is inside of the triangle as shown in Figure 2.9. A fragment holds information about a single point on a triangle and will be used to determine the pixel color. The white points in Figure 2.9 are the generated fragments with interpolated triangle information. For transparency rendering, multiple fragments are combined to get the final pixel color. The following sections describe what happens with each fragment.

[2] cutting out parts that are not in the volume.

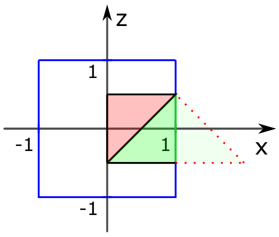


Figure 2.8: Clipped vertices.

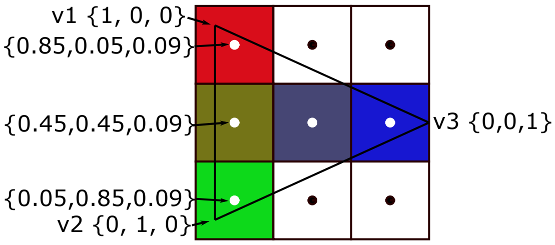


Figure 2.9: Rasterized triangle with interpolated colors in a 3x3 color buffer. The colored pixels are fragments.

Stencil Test

In addition to the color buffer which contains color information, there is a stencil buffer of the same dimension that contains a bitmask. The stencil test uses the stencil buffer to mask out specific areas of the screen. However, by default, the stencil test is disabled and can be disregarded for now. This test will be explained in a later chapter.

Depth Test

Similarly to the stencil test, the depth test uses a depth buffer to compare the depth of the current fragment with the corresponding depth in the depth buffer and it can further decide to discard this fragment. As for classical hidden surface removal, the depth test succeeds if and only if the current fragment depth is smaller than the value stored in the depth buffer. This is the case when the current fragment is closer to the camera than the previously stored fragment.

Fragment Shader

Each generated fragment that passes the depth and the stencil test will result in a new fragment shader invocation. Usually, this stage determines the color of the fragment. This shader receives the interpolated output from the vertex shaders as shown in Figure 2.9.

Furthermore, this shader is able to change the fragment depth or discard the fragment entirely. However, if the shader is programmed to do this, the stencil test and the depth test cannot be evaluated before execution of the fragment shader and have to be evaluated afterward.

The act of performing the depth and stencil test before the fragment shader is called **early depth test**. Using the early depth test is recommended because it can avoid invocations of the fragment shader.

Framebuffer

This stage decides how the color of the fragment, which was determined by the fragment shader, modifies the actual color buffer. If nothing is specified, the new fragment overwrites the fragment in the color buffer. Additionally, the depth of the fragment will be written into the depth buffer if **depth writing** is enabled. **Blending** allows combining the old and new fragment color in a predefined fashion instead of just choosing the new one. The following formula is used for blending:

$$C = C_{src} \cdot O_{src} + C_{dst} \cdot O_{dst}$$

C_{src} and C_{dst} are the new fragment color and the old fragment color respectively. O_{src} and O_{dst} can be set by the application. By default O_{src} is set to one and O_{dst} is set to zero:

$$C = C_{src} \cdot 1 + C_{dst} \cdot 0$$

This effectively overwrites the old fragment color with the new fragment color. However, for more advanced rendering techniques the alpha component of the color can be used as well. A fragment has four color channels. The red, green and blue channels describe the color. The fourth channel is called the alpha channel (opacity) and can be used for color manipulation with the blending equation. Traditional alpha blending (see Section 2.3) requires the following formula:

$$C = C_{src} \cdot \alpha_{src} + C_{dst} \cdot (1 - \alpha_{src})$$

The alpha component of the new fragment is used to combine both fragments. O_{src} is set to α_{src} and O_{dst} is set to $(1 - \alpha_{src})$.

2.2 GPU ARCHITECTURE

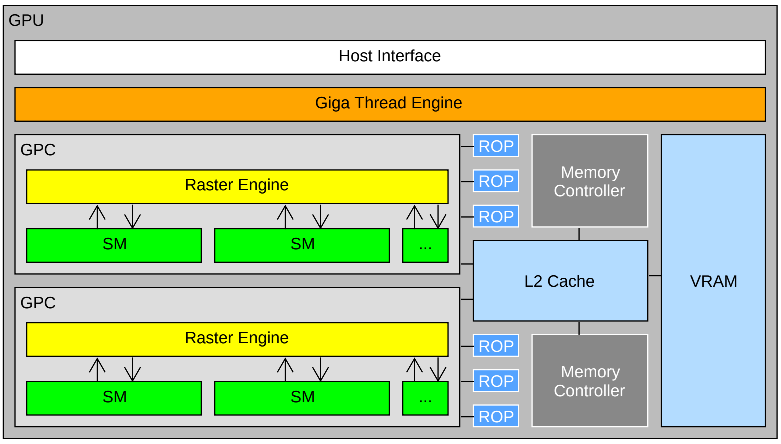


Figure 2.10: GPU architecture.

This section describes the journey of a triangle through the hardware architecture of a GPU, based on the NVIDIA GeForce GTX 980 graphics card description [NVI]. Additional information was taken from *Life of a triangle - NVIDIA's logical pipeline* [Kub15].

Figure 2.10 shows a simplified model of a GPU. The **Host Interface** receives the mesh draw call from the CPU. The **Giga Thread Engine** distributes the work to multiple Graphics Processing Clusters (GPCs). Each GPC is a self-contained GPU with a dedicated Raster Engine and several Streaming Multiprocessors (SMs).

After receiving the triangle data, the GPC passes the triangles to its SMs (see Figure 2.11). The **PolyMorph Engine** is responsible for executing the vertex shader for each triangle. First, the engine fetches the vertices from the Video Random Access Memory (VRAM). Afterwards, the engine schedules warps to execute the vertex shader for individual triangles.

A **warp** (see Figure 2.12) is a cluster of small processors which execute the shader code in parallel. However, the warp cores are not as powerful as CPU cores. Each core has a dedicated Arithmetic Logical Unit (ALU) and Floating Point Unit (FPU) but no Program Counter (PC) or stack. The **Warp Scheduler** manages the PC and dispatches commands to all cores. Individual cores can be masked[3] out to enable conditional code execution. Consequently, diverging cores increase the execution time as shown in Figure 2.13. Cores use the registers from the register file for intermediate storage. The register file features far more registers than typical CPU's. The GTX 980 has about 16.384 32-

[3] cores that are masked out ignore all incoming commands from the dispatch unit

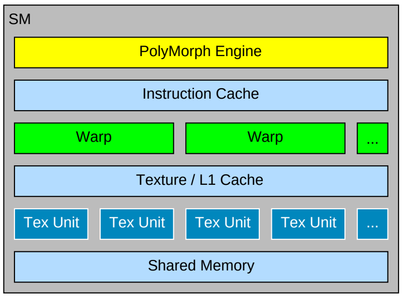


Figure 2.11: Streaming Multiprocessor.

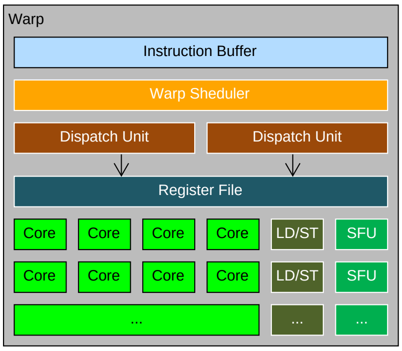


Figure 2.12: GPU warp.

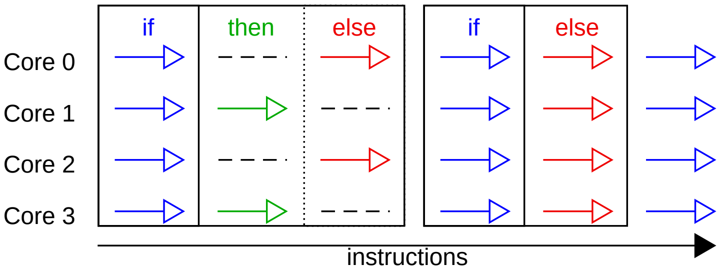


Figure 2.13: Due to the shared PC, execution of the first branch leaves half the cores inactive because they diverge (left). However, if-then-else causes no problem if each core executes the same path (right).

bit registers and 32 cores per warp. Therefore, the Warp Scheduler can execute multiple tasks at once. For instance, if the current task is waiting for a block of memory, the Warp Scheduler switches to another task in the meantime. Switching tasks is very fast since every task has its own set of registers. However, the more registers a task needs, the fewer tasks per warp are executed, and less work is done while waiting for instructions to complete. This phenomenon is called register pressure.

The Load/Store Unit (LD/ST) is used to communicate with global memory. The Special Function Unit (SFU) can compute a predefined complex function, such as sine, cosine or square root, within one clock cycle. The GTX 980 features eight LD/STs and eight SFUs.

Additionally, each warp can access the **Texture Units** of its SM. Texture Units are used to retrieve interpolated texture colors from existing textures. This process is hardware accelerated and is as expensive as retrieving non-interpolated colors.

Shared memory can only be accessed by warps within the SM. Shared memory is used as additional fast intermediate storage for more complex computations.

After the vertex shader computation finishes, the PolyMorph Engine clips the generated triangle and prepares it for rasterization. Now, the triangle leaves its current GPC, and the PolyMorph Engine notifies the Giga Thread Engine. Each GPC is responsible for rasterizing different rectangular areas of the framebuffer. The Giga Thread Engine splits the triangle accordingly and forwards it to the corresponding GPCs.

The Raster Engine within the GPC is used to generate the fragments. First, the stencil and depth-tests are executed. Afterwards, groups of 32 fragments are forwarded to one of the local SMs to execute the fragment shader. After execution of the fragment shader, the Raster Engine uses Render Output Units (ROPs) to write the fragments into the framebuffer. The ROP compresses fragments before sending them to the framebuffer to reduce bandwidth.

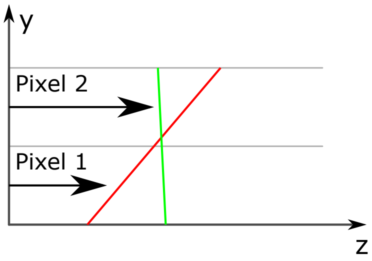


Figure 2.14: Two lines that cannot be sorted according to depth due to their orientation.

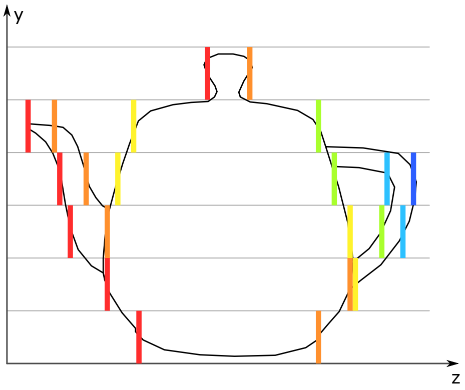


Figure 2.15: Fragment layers along the z-axis. Fragments with the same color form a fragment layer.

2.3 TRANSPARENCY

In a real-world scenario, several objects have transparent properties such as water and glass. Those objects typically have a material property called *alpha* $\alpha \in [0, 1]$ which represents the opacity where $\alpha = 1.0$ is fully opaque and $\alpha = 0.0$ is fully transparent. *Alpha blending* [PD84] was introduced to combine transparent objects. Equation 2.1 describes how to blend a list of fragments with α_i and c_i being the alpha value and color of each fragment.

$$\begin{aligned} C_0 &= \alpha_0 c_0 \\ C_{n-1} &= \alpha_{n-1} c_{n-1} + (1 - \alpha_{n-1}) \cdot C_{n-2} \end{aligned} \tag{2.1}$$

However, the fragment list must be sorted with descending fragment depth because the fragment combination of Equation 2.1 is not commutative (see Figure 1.2). Thus, this equation is (fragment) order-dependent.

One approach for rendering transparent objects requires the application to sort every triangle of all meshes before submitting them to the GPU. However, sorting triangles does not suffice to ensure correct fragment ordering. In the case of overlapping triangles fragments, the fragment order cannot be achieved by ordering triangles (see Figure 2.14). This problem can be fixed by subdividing conflicting triangle. Note that everything needs to be re-sorted if the camera view direction changes. Thus, this approach is not very efficient for rendering transparent objects.

Another approach resolves the ordering problem on the fragment level by temporarily storing every generated fragment into a per-pixel list (see Figure 2.15). Afterwards, the list is being sorted and

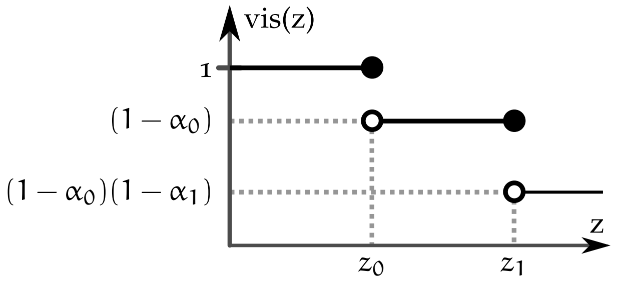


Figure 2.16: Example of the visibility function.

Equation 2.1 is used to blend the fragments. Since this approach does not require a sorted geometry, it is known as Order-Independent Transparency (OIT).

It is possible to derive a different approach for OIT from the alpha blending equation (Equation 2.1). First, the recursion is resolved:

$$C_{n-1} = \sum_{i=0}^{n-1} \left(\alpha_i c_i \prod_{j=i+1}^{n-1} (1-\alpha_j) \right) \tag{2.2}$$

Note that, depending on the fragment index, only alpha values of fragments with a smaller depth value are used in the product. After addition of the depth information z_i to each fragment, this product can be written as:

$$\text{vis}(z) := \prod_{(z_i,\alpha_i):z_i<z} (1-\alpha_i) \tag{2.3}$$

Which leads us to the following formula:

$$C_{n-1} = \sum_{i=0}^{n-1} \alpha_i c_i \text{vis}(z_i) \tag{2.4}$$

Equation 2.3 is also known as the visibility function since it indicates the occlusion of a fragment at any depth. Figure 2.16 illustrates an example of the visibility function. The geometry can be rendered in any order if the visibility function is known. In general, two render passes are required to implement this approach. The first render pass determines $\text{vis}(z)$ by storing the relevant alpha-depth pairs. The second render pass is used to blend all fragments. This approach uses less memory than the previous one since only depth and alpha must be stored and not the fragment color, but an additional render pass is required.

2.4 SCENES

This section gives a brief overview of the scenes that were used for this work. All scenes are rendered with a screen resolution of 800x800 pixels. The median after 2000 iterations was used to describe the rendering times.

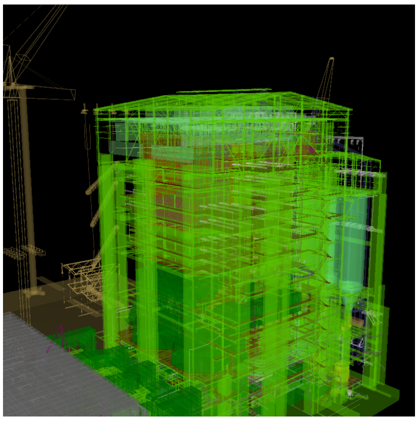
San Miguel

San Miguel scene.

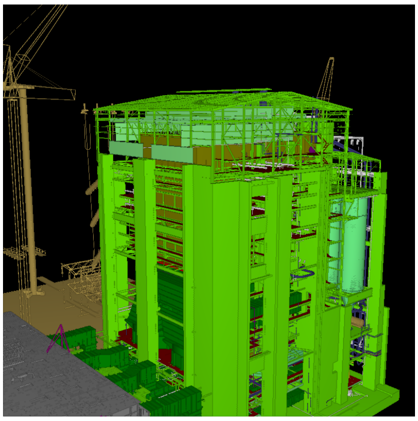
San Miguel (different perspective).

The San Miguel scene from Guillermo M. Leal Llaguno consists of about 10 million triangles and has 0.5 transparent fragments per pixel on average. This scene is used to represent a real-world scenario with few transparent objects.

Powerplant



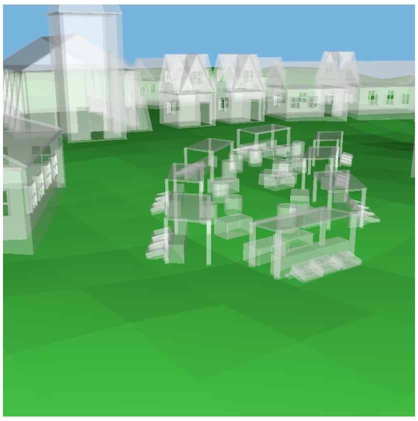
Powerplant scene.



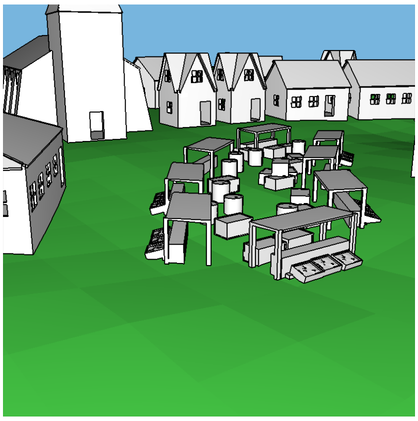
Powerplant without transparency.

The Powerplant scene published by the University of North Carolina consists of about 13 million triangles and has 11 transparent fragments per pixel on average. The original materials were modified into transparent materials to achieve the high number of transparent fragments per-pixel. Thus, this scene is used to model a scenario with an extraordinary amount of transparent fragments.

Village



Village scene.



Village without transparency.

The village scene was created by me and consists of only 17 thousand triangles and has 1.6 transparent fragments per pixel on average. This scene is used because it has slightly more transparent fragments than the average real-world scenario but not as many as the Powerplant scene.

3 RELATED WORK

This chapter gives a brief overview of the evolution of Order-Independent Transparency (OIT) algorithms. Later, Adaptive Transparency (AT) and Multi-Layer Alpha Blending (MLAB) will be explained in more detail because this work focuses on optimizing these algorithms.

3.1 PER-PIXEL LISTS

The A-Buffer algorithm [Car84] requires a per-pixel list to capture all fragments. After capturing all fragments, each list gets sorted according to fragment depth. Then, all fragments are blended to retrieve the final pixel color. This algorithm could not be implemented on graphics cards for a long time but was frequently used for offline rendering[4].

After the introduction of atomic operations on graphics cards[5], Yang et al. [YHGT10] realized the A-Buffer with a linked list implementation for modern GPUs. A memory pool with a fixed size is allocated to store the nodes for the list because dynamic memory allocation (for nodes) was not possible on the GPU[6] at the time. Thus, this implementation can only store a fixed number of fragments.

The dynamic fragment buffer [MCTB12] solves this problem by using two render passes. In the first pass, a buffer is used to count the number of transparent fragments per pixel (counting buffer in Figure 3.1). Afterwards, a prefix sum[7] is performed to generate the offsets for the per pixel arrays (base buffer). Then, the fragment buffer gets resized if required. In the second render pass, the fragments are transferred into the fragment buffer according to the base buffer offsets. After a fragment is inserted into the buffer, the corresponding value in the base buffer gets incremented in order to store the next fragment appropriately. Finally, each list gets sorted and blended.

Fast sorting for exact OIT of complex scenes [KLZ14] proposes optimizations for per-pixel lists. Some of those optimizations are applicable for this work as well and will be explained in the next chapter.

[4] refers to rendering on the CPU
[5] 2009 for DirectX, 2011 for OpenGL
[6] dynamic memory allocation is possible in newer versions of CUDA
[7] each value in the array becomes the sum of all previous values

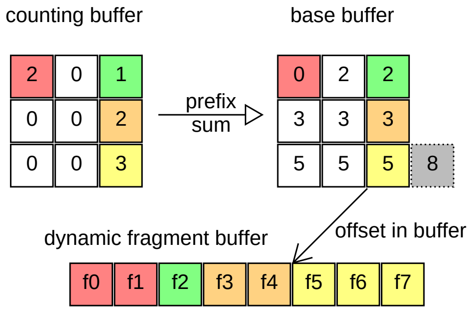


Figure 3.1: Dynamic fragment buffer algorithm.

3.2 DEPTH PEELING

Depth Peeling [Eve01] solves OIT by capturing only one fragment layer per render pass (fragment layers are shown in Figure 2.15). However, it needs k render passes, with k being the maximum number of fragments per pixel. This approach does not require a variable amount of memory since each layer gets immediately blended with the framebuffer after it is captured. More recent Depth Peeling techniques can capture up to 32 layers at once [BM08; LWXW09; LHLW09].

3.3 K-BUFFER

The k-Buffer [BCL+07] captures the k foremost fragments for each pixel in a single render pass and discards the remaining ones. This implementation suffers from flickering artifacts caused by read-modify-write hazards during fragment insertion. Those hazards could not be avoided at that time since mutual exclusion for fragment insertion was not possible due to the lack of atomic operations.

The k^+-Buffer [VF14] gets rid of the flickering artifacts by using spin-locks powered by atomic operations. Additionally, a dynamic memory allocation strategy estimates the optimal k value for the current frame and resizes the k-Buffer accordingly.

The variable k-Buffer [VVPM17] allocates a fixed amount of memory such that each pixel can store up to k fragments on average. However, the per-pixel fragment lists are smaller or bigger than k depending on the depth complexity[8] of the current pixel and other factors. This approach uses its storage more efficiently than the previous k-Buffer techniques.

[8] indicates the amount of transparent fragments for one pixel

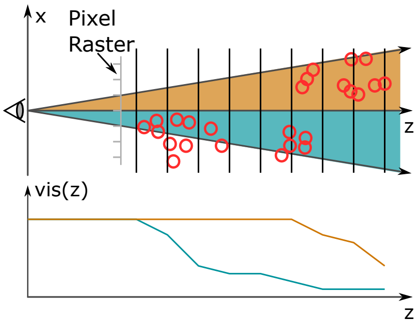


Figure 3.2: Opacity Shadow Maps [KN01].

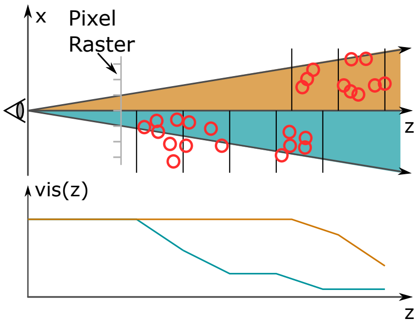


Figure 3.3: Deep Opacity Maps [YK08].

A scenario with unevenly distributed transparent geometry. Opacity Shadow Maps require a higher grid resolution than Deep Opacity Maps to approximate the visibility function.

3.4 PER-PIXEL PACKED ARRAYS

Most of the following algorithms store the visibility function in a compressed form that requires a fixed amount of memory per-pixel. Note that those algorithms require two render passes as described in Section 2.3.

Opacity Shadow Maps [KN01] capture fragment occlusion (inverse visibility) in a uniformly discretized grid as shown in Figure 3.2.

Deep Opacity Maps [YK08] work similar to Opacity Shadow Maps. In this technique, each pixel grid starts at the first occurring fragment (see Figure 3.3).

Occupancy Maps [SA09] are using a bitfield to indicate the occlusion. This works well for occluders with the same transparency, for example hair. The bitfield basically is a higher resolution Opacity Shadow Map that only indicates if a grid cell is occluded. Thus, the cell has no information about the amount of occlusion.

Fourier Opacity Maps [JB10] are storing the visibility function in the Fourier basis with a fixed number of Fourier terms. This results in soft transition within the visibility function which is good for soft occluders like hair and smoke but not good for sharp occluders like glass.

Adaptive Transparency (AT) [SML11] stores a fixed number of depth-, transmittance pairs per-pixel to represent the visibility as a piecewise constant function.

Multi-Layer Alpha Blending (MLAB) [SV14] stores a fixed number of depth-, color-, alpha tuples per-pixel. If the tuple limit per-pixel is reached, two tuples will be blended to allow insertion of a new tuple. Note that this approach does not require an additional geometry pass since all color information is stored as well. Only a single fullscreen

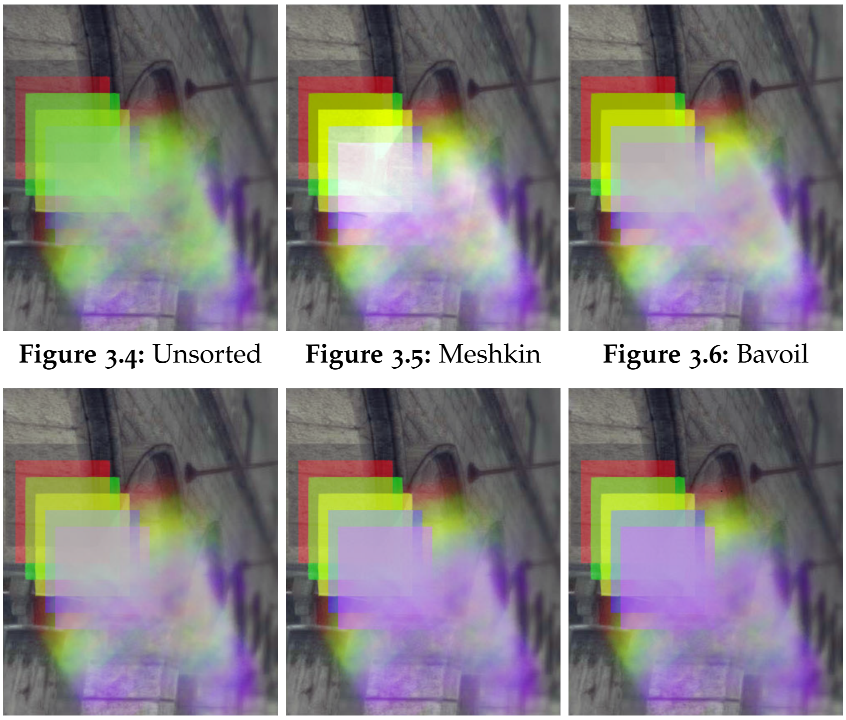


Figure 3.4: Unsorted

Figure 3.5: Meshkin

Figure 3.6: Bavoil

Figure 3.7: McGuire

Figure 3.8: McGuire Depth

Figure 3.9: Sorted

Six colored squares and particle billboards with $\alpha = 0.35$ at different depths. From left to right: The unsorted OVER worst case, blended order-independent transparency approximations of increasing quality, and the common sorted OVER compositing [MB13].

render pass[9] is required to blend all tuples to obtain the final fragment color.

Moment-Based Transparency [MKKP18] stores a fixed number of moments (similar to Fourier terms) to represent the visibility function.

3.5 OTHER OIT TECHNIQUES

Sort-Independent Alpha Blending [Mes07] modified the non-commutative blending operator to a similar commutative blending operator (Figure 3.5). Unfortunately, this approach only works well if the α values are small and the fragment colors are similar to each other.

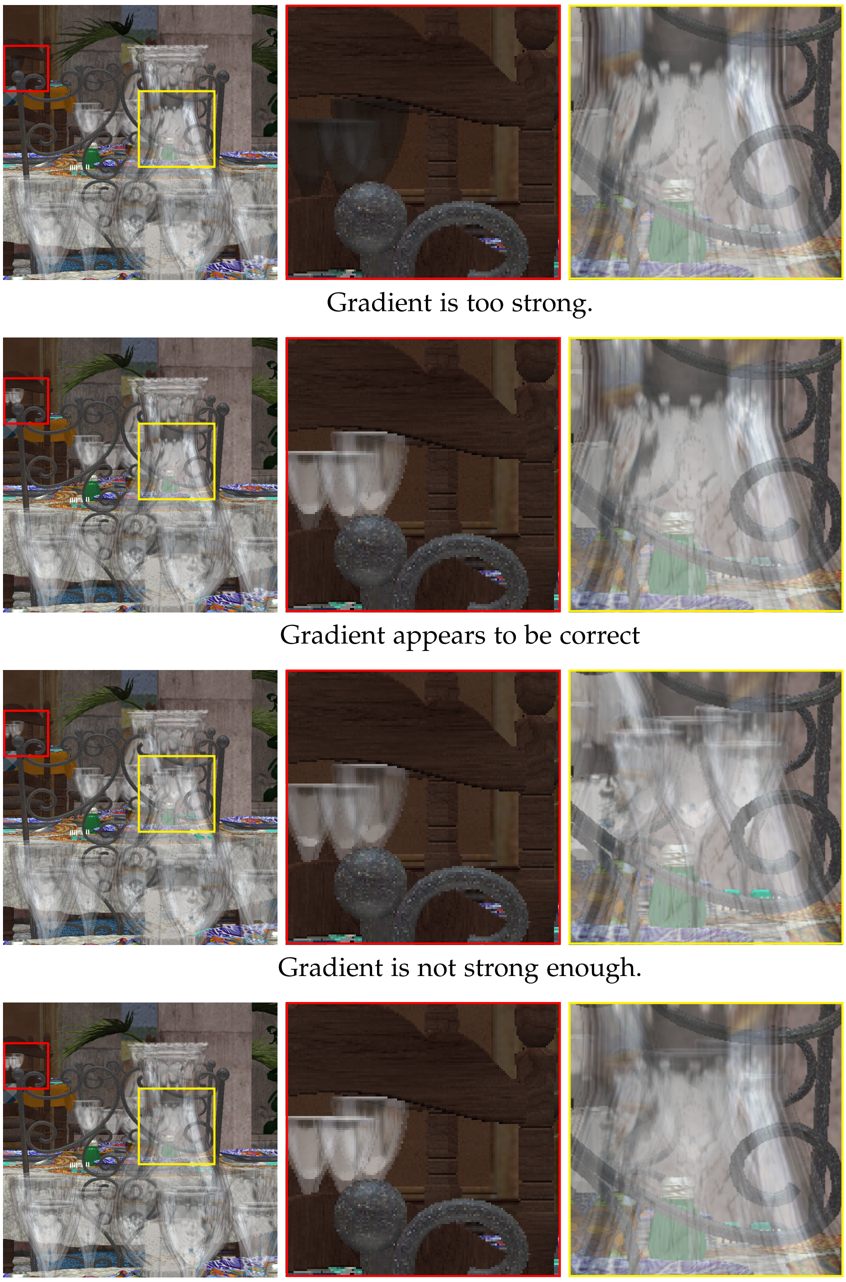
Bavoil & Meyers *Weighted Average* [BM08] refined Meshkin's blending operator with a better approximation for fragment color and coverage (see Figure 3.6).

[9] This can be done by drawing a quad that fills the entire screen. Thus, the fragment shader will be executed for each pixel on the screen.

Weighted OIT [MB13] refined the commutative blending operator again and added additional depth weights to each fragment which is similar to the occlusion given by the visibility function. See Figure 3.7 for the result of the new blending operator without depth weights. Figure 3.8 shows the result with the addition of depth weights. Unfortunately, the depth weight function needs to be reconfigured depending on the scene depth complexity which can be quite challenging (see Figure 3.10).

Stochastic Transparency [ESSL10] uses a stochastic approach to blend fragments but requires a large number of samples per pixel to avoid noise artifacts.

Hybrid Transparency [MCTB13] proposes a transparency algorithm that uses a slow but precise algorithm to combine important fragments and a fast approximate algorithm to combine unimportant fragments. The important fragments are usually the fragments which are closest to the camera because they are less occluded. Therefore, a k-Buffer is used for the precise algorithm. Weighted Average [BM08] is used for the fast approximate algorithm.



Gradient is too strong.

Gradient appears to be correct

Gradient is not strong enough.

Sorted reference.

Figure 3.10: Picking the correct depth weights for Weighted OIT can be quite challenging.

3.6 ADAPTIVE TRANSPARENCY

Adaptive Transparency (AT) requires two render passes. In the first render pass, a compressed version of the visibility function is computed. In the second render pass, the transparent objects are

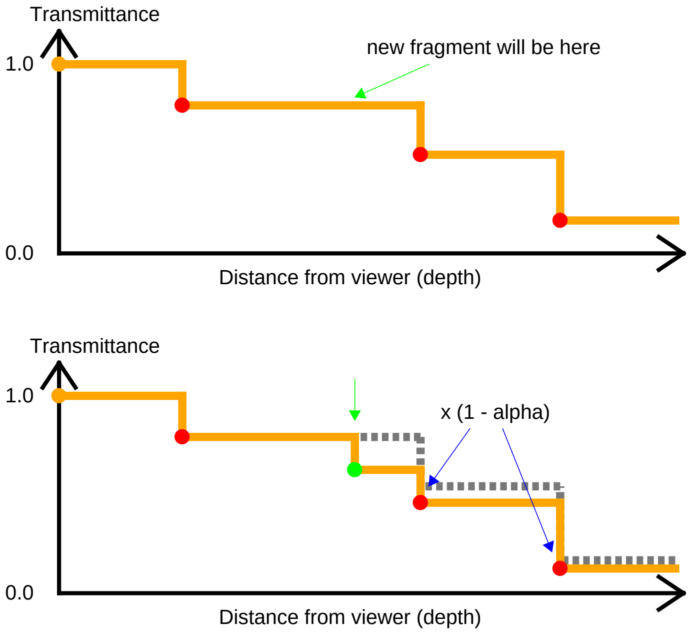

Transmittance
1.0
new fragment will be here
0.0
Distance from viewer (depth)
Transmittance
1.0
x (1 - alpha)
0.0
Distance from viewer (depth)

Figure 3.11: Fragment insertion for the AT visibility function. The red nodes contain the fragment depth z and transmittance $\text{vis}(z)$.

blended by using the computed visibility function (Equation 2.4). Since the visibility function is a stepwise constant function by definition, AT only stores the nodes where the function changes (see red nodes in Figure 3.11). A fixed number of $(\text{depth}, \text{transmittance})$ nodes is stored in a per-pixel array with ascending depth. Each node requires 8 bytes of storage: 4 bytes for transmittance and 4 bytes for depth.

If a new fragment with opacity α should be inserted, the appropriate insertion position is determined first. Afterwards, the transmittance of each node that is occluded by the inserted fragment gets multiplied with $(1 - \alpha)$ (see Figure 3.11).

If the newly inserted fragment exceeds the fixed number of nodes, additional compression is performed. To minimize the error caused by the compression, the node with the least contribution to the visibility function gets removed. Since the visibility function is a stepwise constant function, the introduced error by removing a node is proportional to the rectangular area between two nodes (see Figure 3.12). Generally, three node removal strategies are possible:

1. Overestimation: The node gets removed from the list which raises the visibility functions value until the next node.

2. Underestimation: The node gets removed from the list, and the transmittance of the previous node is set to the transmittance of the removed node.

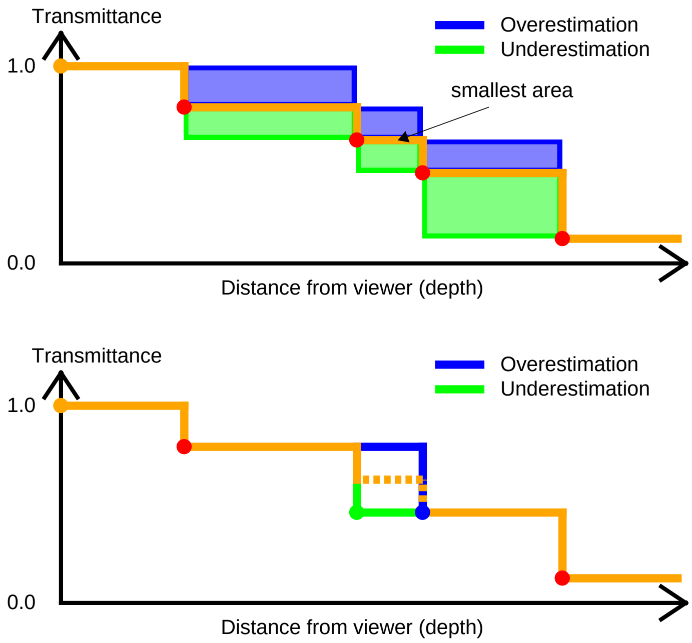


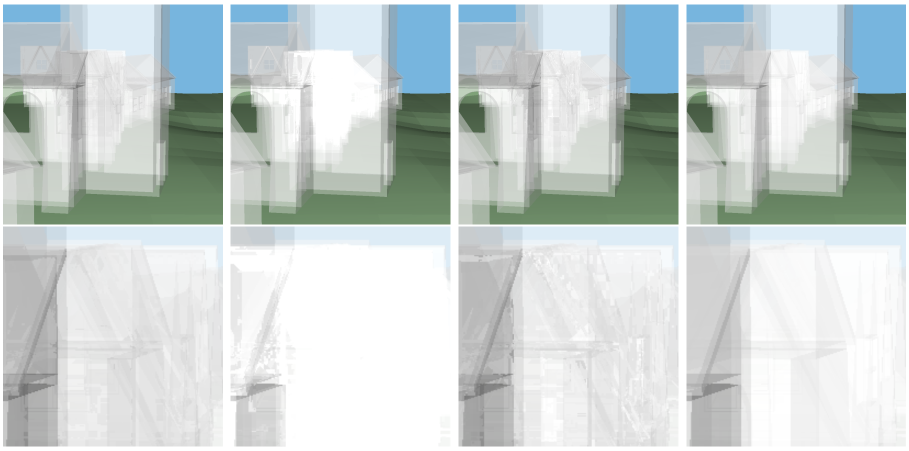
Figure 3.12: Areas used for the AT compression. After the smallest area was determined, an overestimation (blue) or an underestimation (green) of the visibility function can be used to remove one node.

3. Both: Using either overestimation or underestimation depending on the smallest error.

Generally, the underestimation has the least visible artifacts (see Figure 3.13). Overestimation causes more artifacts because a lot of fragments appear brighter than they should be. Darker fragments that are produced by the underestimation do not stand out as much as brighter fragments. Thus, AT uses only the underestimation.

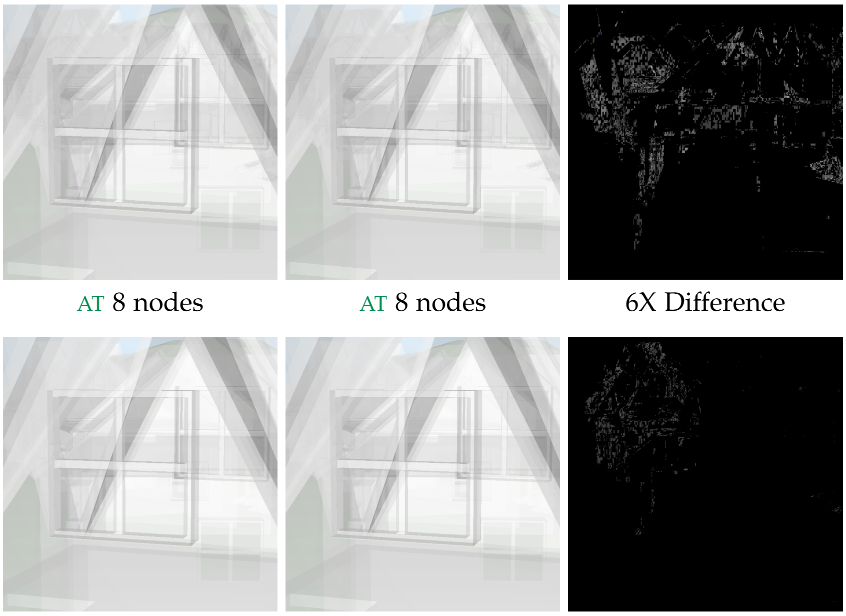
The insertion of a new node takes place in the fragment shader of the first geometry pass. The insertion must be protected by a spinlock because concurrently inserting new fragments and compressing the function does not work with the presented insertion algorithm. In general, all nodes are initialized with a depth value of infinity and a transmittance value of one to avoid additional insertion code for the first n fragments.

Note that the compression of the visibility function depends on the order of the inserted fragments. It is therefore recommended to submit the geometry in the same order to avoid artifacts. However, the fragment insertion order slightly changes between frames due to the concurrent execution on the GPU (see Figure 3.14). Unfortunately, the GPU only assures ordering for the fragments when they are written to the framebuffer. The fragment shaders can be executed concurrently



Underestimation Overestimation Both Reference

Figure 3.13: The village scene rendered by AT with 8 nodes per-pixel.



AT 8 nodes AT 8 nodes 6X Difference

AT 12 nodes AT 12 nodes 6X Difference

Figure 3.14: AT flickering artifacts caused by the varying order of fragment insertions between frames.

and out of order. Increasing the number of nodes reduces those artifacts.

3.7 MULTI-LAYER ALPHA BLENDING

The concept of Multi-Layer Alpha Blending (MLAB) is based on the recursive Alpha Blending Equation (Equation 2.1).

Two fragments $f_a := (c_a, \alpha_a, z_a)$ and $f_b := (c_b, \alpha_b, z_b)$ with $z_a < z_b$ can be blended like:

$$C_c = c_a \alpha_a + (1 - \alpha_a) c_b \alpha_b \tag{3.1}$$

We define for each fragment f:

1. The pre-multiplied color $C_f := c_f \alpha_f$
2. The transmittance $T_f := (1 - \alpha_f)$
3. The depth $Z_f := z_f$

Thus, Equation 3.1 can be rewritten in the following manner:

$$C_c = C_a + T_a C_b$$

Another fragment $f_d = (C_d, T_d, Z_d)$ can be blended with C_c if $Z_d < Z_a$ or $Z_d > Z_b$:

$$\begin{aligned} T_c &:= T_a T_b \\ C_{\text{final}} &= \begin{cases} C_d + T_d C_c & Z_d < Z_a \\ C_c + T_c C_d & Z_d > Z_b \end{cases} \end{aligned}$$

C_d can not be blended with C_c if $Z_d > Z_a$ and $Z_d < Z_b$ (Fragment f_d is positioned between f_a and f_b). Therefore, if more fragments are blended this way, the range in which correct blending is not possible grows.

MLAB tries to solve this problem by storing n (C, T, Z) nodes per-pixel. The nodes are sorted with ascending depth. Each node requires 8 byte of storage: 3 bytes for color, 1 byte for transmittance and 4 bytes for depth. The final color, with respect to the opaque background color C_{opaque}, is computed as follows:

$$C_{\text{final}} = C_{\text{opaque}} \prod_{i=0}^{n-1} T_i + \sum_{i=0}^{n-1} C_i \prod_{j=0}^{i-1} T_i$$

The first n fragments are inserted without merging nodes. Additional fragments are inserted into the correct position within the sorted per-pixel array first and then two adjacent nodes are merged since only n nodes can be stored. Two elements of the array are merged as follows:

$$\begin{aligned} C_i &= C_i + T_i C_{i+1} \\ T_i &= T_i T_{i+1} \\ Z_i &= Z_i \end{aligned}$$

The depth value of the first node (which is the smaller depth value) is taken to guarantee correct blending of opaque surfaces which have a transmittance of zero. Unfortunately, if another fragment is inserted that is located between two merged nodes, proper blending is still not possible. However, this scenario becomes less likely with more nodes.

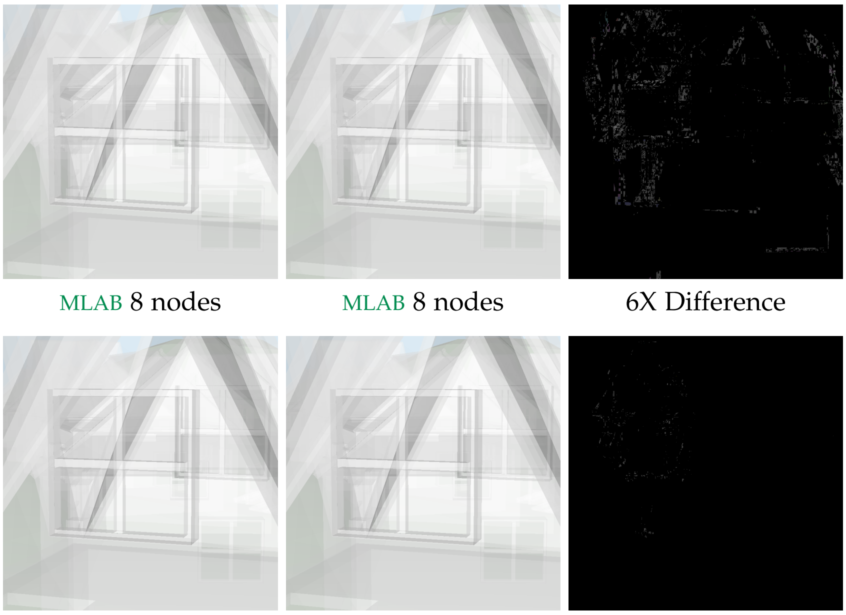
The authors of MLAB tried to determine the most appropriate merge position based on smallest color deltas, transmittance, and distance from the viewer. According to the authors, merging nodes with the smallest transmittance produces the most visually pleasing images. Since the visibility function is a monotonically decreasing function, the nodes with the smallest transmittance are the last two nodes.

Therefore, the algorithm to insert a fragment is relatively short:

Listing 3.1: MLAB fragment insertion

```
void insert(Fragment f){
   Fragment frags[n + 1];
   frags[0] = f;

   // load per-pixel array
   for(int i = 0; i < n; ++i)
      frags[i+1] = LOAD(i);

   // one-pass bubble sort to insert fragment
   for(int i = 0; i < n; ++i){
      if(frags[i].Z > frags[i + 1].Z){
         Fragment temp = frags[i];
         frags[i] = frags[i + 1];
         frags[i + 1] = temp;
      }
   }

   // merge fragments accordingly
   Fragment merge;
   merge.C = frags[n-1].C + frags[n-1].T * frags[n].C;
   merge.T = frags[n-1].T * frags[n].T;
   merge.Z = frags[n-1].Z;
   frags[n-1] = merge;

   // store per-pixel array
   for(int i = 0; i < n; ++i)
      STORE(i, frags[i]);
}
```

Usually, all nodes are initialized with $(C = 0, T = 0, Z = \infty)$ to avoid additional insertion code for the first n fragments. Furthermore, the insertion of a new fragment must be protected by a per-pixel spinlock. Note that the fragment compression of MLAB depends on the order of the inserted fragments as well. Therefore, the compression changes slightly between frames (see Figure 3.15).



MLAB 8 nodes

MLAB 8 nodes

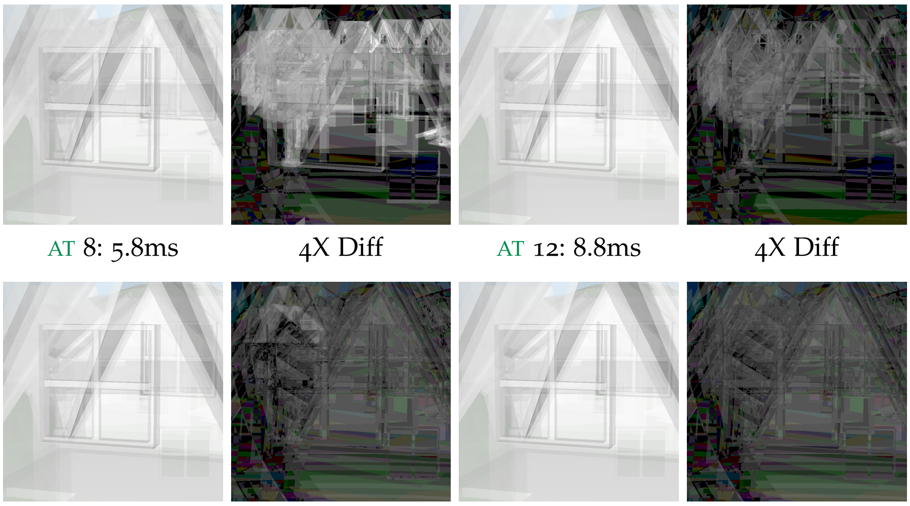
6X Difference

MLAB 12 nodes

MLAB 12 nodes

6X Difference

Figure 3.15: MLAB flickering artifacts caused by the varying order of fragment insertions between frames.



AT 8: 5.8ms

4X Diff

AT 12: 8.8ms

4X Diff

MLAB 8: 4ms

4X Diff

MLAB 12: 5.8ms

4X Diff

Figure 3.16: Comparison between AT and MLAB. The diff-pictures describe the difference to the sorted alpha blending solution.

MLAB produces faster and better images than AT (Figure 3.16). The main overhead of AT are the two geometry passes.

4 GENERAL OPTIMIZATIONS

This chapter discusses how to implement per-pixel packed array OIT techniques in OpenGL. The example shader code is written in GLSL (OpenGL Shading Language).

4.1 STORAGE

In order to access our per-pixel array, some kind of read/write storage is required. OpenGL provides two different types for read/write storage that can be accessed from shaders. The Shader Storage Buffer Object (SSBO) is similar to a CPU buffer that was allocated from the heap. A texture mainly differs from an SSBO when accessing individual elements.

Shader Storage Buffer

SSBO storage is a contiguous block of memory. Data within a SSBO can be accessed by using a one dimensional index. The index for a per-pixel array located at $(x, y) \in \mathbb{N}^2$ can be calculated as follows:

$$\text{index}(x, y, i) = (y \cdot w + x) \cdot n + i \tag{4.1}$$

$w \in \mathbb{N}$ is the width of the screen in pixels. $n \in \mathbb{N}$ is the number of array nodes per pixel. $i \in \{0, 1, ..., n-1\}$ is the node index within a per-pixel list .

Textures

Textures map 1D storage to 2D or 3D storage while preserving locality of the data. This is achieved by using a space filling curve. Possible space filling curves are the Z-order curve [Mor66] (shown in Figure 4.2) or the Hilbert curve [Hub91]. Data locality is important because the threads within a warp have very similar pixel coordinates. A 3D texture can be used for Per-Pixel arrays. 3D textures preserve locality in all three directions. However, our use-case requires a high locality in z-direction (array nodes) and a lower locality for the other axes (pixel coordinates).

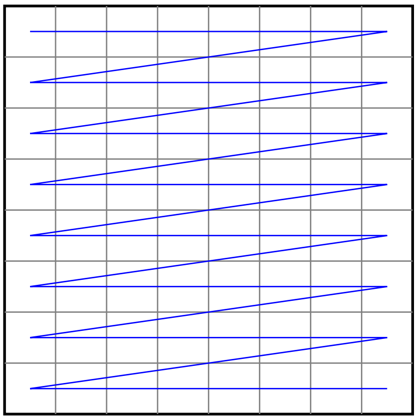


Figure 4.1: SSBO (Equation 4.1).

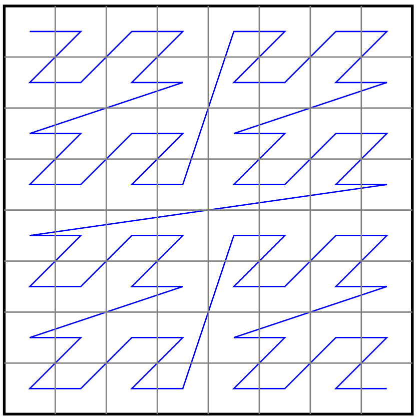


Figure 4.2: 2D Z-order curve.

Different 8x8 memory layouts (Number of nodes $n = 1$). The blue curve describes the order of bytes in memory.

Interleaved Buffer

The interleaved SSBO (inspired by [Kno15]) provides a higher locality for array nodes. Nodes with similar x-, y-coordinates are packed into a group (see Figure 4.4). $g_x \in \mathbb{N}$ and $g_y \in \mathbb{N}$ describes the size of the group in x and y direction.

$$\begin{aligned} \text{gID} &= \frac{y}{g_y} \cdot \frac{w}{g_x} + \frac{x}{g_x} \\ \text{gOffset} &= \text{gID} \cdot n \cdot g_x \cdot g_y \\ \text{localID} &= (y \mod g_y) \cdot g_x + (x \mod g_x) \\ \text{stride} &= g_x \cdot g_y \\ \text{index}(x, y, i) &= \text{gOffset} + \text{localID} + \text{stride} \cdot i \end{aligned} \tag{4.2}$$

Table 4.1 shows the performance of the discussed storage options. The interleaved SSBO is up to 8% faster than the texture. However, this seems to strongly depend on the scene.

Unfortunately, the interleaved SSBO can cause problems if single nodes are not aligned to 32 byte. The graphics card can only read/write global memory in 32 byte chunks [Har13] which can cause lost updates if the nodes are not aligned accordingly. Fortunately, this does not seem to happen in the default MLAB implementation. However, the Unsorted Multi-Layer Alpha Blending (UMLAB) which will be introduced in Chapter 5 does not work with the presented interleaved SSBO. The lost updates increase the flickering artifacts between two frames as shown in Figure 4.5. This probably happens because UMLAB stores the nodes in a random pattern after the algorithm is finished (scattered writes).

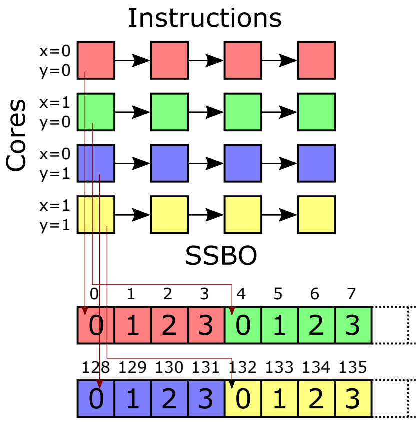


Figure 4.3: Simple SSBO layout (Equation 4.1).

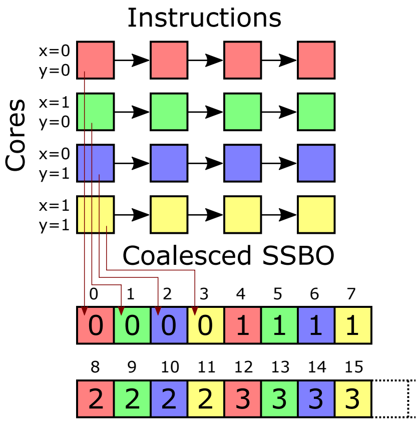
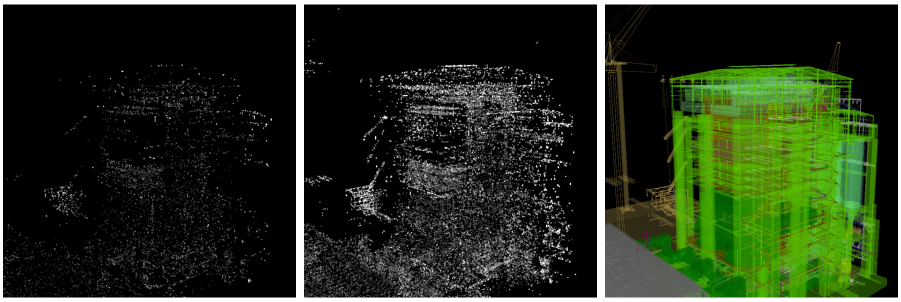


Figure 4.4: Interleaved storage Equation 4.2.

A 2x2 warp accessing nodes concurrently ($n = 4, w = 32$). The interleaved storage has better memory locality ($g_x = 2, g_y = 2$).



Aligned noise Unaligned noise Scene

Figure 4.5: Flickering artifacts in UMLAB (16 nodes) with interleaved SSBO storage aligned to 32 bytes and without alignment. The noise was blurred with a radius of two pixels to emphasize the difference.

Table 4.1: Texture (Tex), SSBO and interleaved SSBO (ISSBO) rendering times in milliseconds (NVIDIA GTX 1080). The ISSBO has the group size of 8 ($g_x = 2, g_y = 4$) because it performed best. MLAB with 4 and 8 nodes per-pixel was used for rendering. The time for rendering San Miguel times exclude the opaque rendering time.

SCENE(NODES)	TEX	SSBO	ISSBO
Powerplant(4)	15.54	16.17	**15.22**
Powerplant(8)	17.93	18.28	**16.94**
Village(4)	**1.29**	1.60	1.31
Village(8)	**1.74**	2.32	**1.74**
San Miguel(4)	**2.19**	2.24	2.21
San Miguel(8)	**2.30**	2.44	2.42

4.2 SPINLOCK

Spinlocks provide mutual exclusion for fragment insertion. To maximize the fragment throughput, one lock per-pixel is required. A 2D integer texture can be used as the lock data. A pixel at (x, y) is locked if the texture at (x, y) is equal to 1. The Pixel is unlocked if the texture at (x, y) is equal to 0. The texture data is accessed via atomic operations to avoid race conditions. The atomic compare-and-swap operation can be used to acquire the lock. An atomic compare-and-swap operation compares a reference value with the atomic value and modifies the atomic value if and only if they match. In our case, the atomic compare-and-swap tests if the atomic has a value of zero. If the test succeeds, the atomic value is set to one and the lock is acquired. After the fragment insertion, the atomic value will be set to zero by an atomic exchange operation.

GLSL provides the imageAtomicCompSwap(texture, position, compare, data) function for the atomic compare-and-swap. Note, that this functions returns the value of the atomic before the atomic operation took place.

A naive implementation of a spinlock is shown in Listing 4.1. The memory barrier in line 3 is required to make sure that the data written inside the *insert()* function becomes visible to other warps.

Listing 4.1: Naive spinlock

```
1  while(imageAtomicCompSwap(tex↩
       , xy, 0, 1) != 0);
2  insert(...);
3  memoryBarrier();
4  imageAtomicExchange(tex, xy, ↩
       0);
```

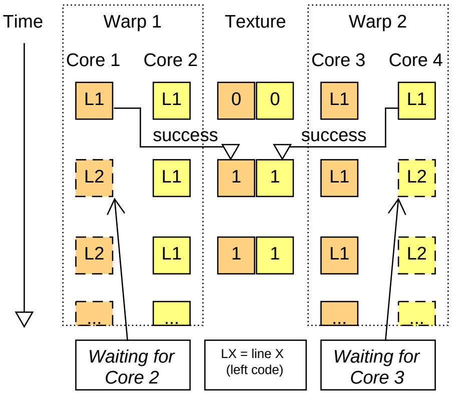


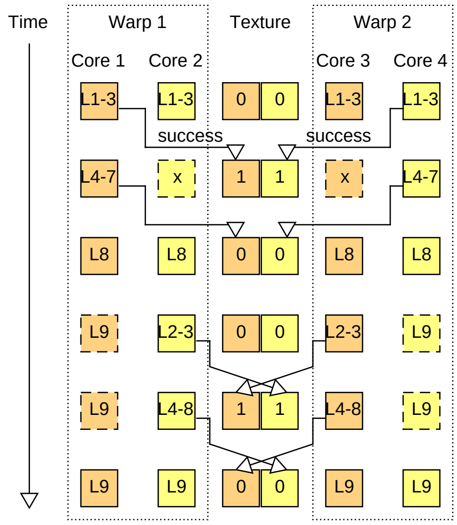
Figure 4.6: Deadlock.

Figure 4.6 describes how a deadlock occurs if two warps want to lock the same two pixels of the screen. First, both warps request the lock for the same two pixels. In this example, cores with the same color are trying to lock the same pixel. However, only one core from the first warp and one core from the second warp acquire the lock. This causes a deadlock on the GPU because the individual cores can only advance as a warp [Kub15]. This means, that one warp can only advance if all cores acquired the lock in line 1.

The critical section needs to be encapsulated in the loop to avoid a deadlock (see Listing 4.2).

Listing 4.2: Improved spinlock

```
1  bool keepWaiting = true;
2  while(keepWaiting) {
3     if(imageAtomicCompSwap(tex↩
        , xy, 0, 1) == 0) {
4        insert(...);
5        memoryBarrier();
6        imageAtomicExchange(tex↩
           , xy, 0);
7        keepWaiting = false;
8     }
9  }
```



In this case, the deadlock does not occur because the *if* statement within the loop assures that cores release the lock after one iteration (line 6).

Unfortunately, Listing 4.2 can cause some graphic cards drivers to crash. This happens, because fragment shaders are always executed in multiples of 2x2 pixel quads [Kub15]. These pixel quads are needed to evaluate derivates for specific texture lookups (for example linear interpolated texture lookups). Threads within that group, which do not belong to the triangle are called helper invocations. According to the specification: "Atomic operations to image, buffer, or atomic counter variables performed by helper invocations have no effect on the underlying image or buffer memory. The values returned by such atomic operations are undefined." Thus, line 3 in Listing 4.2 returns an undefined result for helper invocations which can cause those invocations to repeat the loop until the driver eventually crashes[10]. Fortunately, the addition of *if(!gl_HelperInvocation)* before the loop can prevent the helper invocations from being stuck as shown in Listing 4.3.

[10] the graphics card driver resets itself when a program does not terminate after several seconds

Listing 4.3: Final spinlock

```
if(!gl_HelperInvocation) {
   bool keepWaiting = true;
   while(keepWaiting) {
      if(imageAtomicCompSwap(tex, xy, 0, 1) == 0) {
         insert(...);
         memoryBarrier();
         imageAtomicExchange(tex, xy, 0);
         keepWaiting = false;
      }
   }
}
```

4.3 REGISTER

Working in global memory for the fragment insertion is too expensive if each per-pixel node must be accessed more than once. It is therefore recommended to create a temporary copy of the per-pixel array within the fragment shader. Arrays are placed in registers or local memory (L1 cache or shared memory) depending on the usage. Registers are faster than local memory but they cannot be dynamically indexed [KLZ14].

Thus, if the array indices are known at compile time, the array is placed in registers. The following sections describe how to properly work in registers.

Loop Unrolling

Loop unrolling eliminates loops with n iterations by writing the loop body n times (see Listing 4.4).

Listing 4.4: Unrollable Loop

```
// before unrolling:
int array[4];
for(int i = 0; i < 4; ++i) {
   array[i] = i;
}
// after unrolling:
int array[4];
array[0] = 0;
array[1] = 1;
array[2] = 2;
array[3] = 3;
```

Fortunately, the array indices in the example are known at compile time due to the loop unrolling and the array can be placed in registers.

Sometimes more complex loop conditions can prevent the compiler from unrolling loops. Adding an additional condition that indicates the maximum loop iteration count can re-enable loop unrolling (see Listing 4.5). Note, that the maximal loop iteration count must be known at compile time to enable loop unrolling.

Listing 4.5: Complex loop condition

```
int array[4];
...
// not unrollable:
for(int i = 0; array[i] > 0; ++i) {
   array[i] -= 1;
}
// unrollable:
for(int i = 0; array[i] > 0 && i < 4; ++i) {
   array[i] -= 1;
}
// after unrolling
if(array[0] > 0){
   array[0] -= 1;
   if(array[1] > 0){
      array[1] -= 1;
      if(array[2] > 0){
         array[2] -= 1;
         if(array[3] > 0){
            array[3] -= 1;
         }
      }
   }
}
```

Note, that the compiler can decide to refuse loop unrolling if the loop iteration count is too high (sometimes even 12 iterations can be too many). Unfortunately, GLSL does not provide an option to force unrolling. However, the graphic card manufacturers often provides a preprocessor directive for this (#pragma optionNV (unroll all) for NVIVIA graphic cards).

Assigning with dynamic index

Some algorithms require an assignment with a dynamically calculated index. However, doing the dynamic assignment in Listing 4.6 will prevent the compiler from placing the array in registers.

Listing 4.6: Dynamic assignment preventing register usage

```
int array[4];
// i is not known at compile time
int i = dynamicIndex();
array[i] = 2;
```

Assigning a value in an unrollable loop (see Listing 4.7) allows the array to be placed in registers. I also tested the assignment with a switch construct and with a if-else-tree construct but they were a little bit slower than the loop construct due to the higher warp divergence (Table A.1).

Listing 4.7: Dynamic assignment allowing register usage

```
int array[4];
// i is not known at compile time
int i = dynamicIndex();
for(int j = 0; j < 4; ++j){
   if(i == j){
      array[j] = 2;
   }
}
// unrolled:
if(i == 0) array[0] = 2;
if(i == 1) array[1] = 2;
if(i == 2) array[2] = 2;
if(i == 3) array[3] = 2;
```

4.4 INSERTION INTO SORTED LIST

Inserting one element into a sorted array is pretty straightforward. In this example, the sorted array starts at index one and the new element is inserted at index zero. Then, the new element is put into the correct position by comparing it with the next element in the list and swapping them if they were in the wrong order. However, there is still a choice between bubbling up with and without early out:

Listing 4.8: Insertion with early out

```
for(int i = 0; i < SIZE - 1; ++i){
   if(array[i] <= array[i+1]) break;
   // swap
   int tmp = array[i];
   array[i] = array[i + 1];
   array[i + 1] = tmp;
}
```

Listing 4.9: Insertion without early out

```
for(int i = 0; i < SIZE - 1; ++i){
   if(array[i] > array[i+1]) {
      // swap
      int tmp = array[i];
      array[i] = array[i + 1];
      array[i + 1] = tmp;
   }
}
```

The version without early out is faster for small arrays (size < 16) if a lot of elements are inserted (see Powerplant times in Table A.2). The early out is only effective if all cores within a warp are able to take it. If that is not the case, the early out version is an additional overhead. However, the insertion with early out is faster if less elements are inserted because the early out is likely to be taken in all warp cores (see Village times in Table A.2). Overall, the difference in time for one insertion is so small, such that it does not matter which version is used.

Additionally, a version that used two moves instead of a swap was tested as well (Listing A.1). However, the swap always performs better which leads to the conclusion that the GPU probably has a register swap instruction or is capable of doing register renaming[11].

4.5 SORTING

Some algorithms require an entire array to be sorted. The following algorithms can be used to sort an array of fixed size in registers.

BUBBLE SORT

Bubble sort compares adjacent pairs and swaps them if they are in the wrong order. This gets repeated until the array is sorted. The standard

[11] register renaming eliminates false data dependencies which results in higher instruction level parallelism

bubble sort implementation (Listing 4.10) is compatible with registers. The early out condition (line 11) exits the loop if the array is sorted prior to the last iteration. The version without the early out (line 11) performs worse (see Table A.3).

Listing 4.10: Bubble Sort

```
 1  for(int n = size; n > 1; --n) {
 2     bool sorted = true;
 3     for(int i = 0; i < n - 1; ++i) {
 4        if(array[i] > array[i + 1]) {
 5           int tmp = array[i];
 6           array[i] = array[i + 1];
 7           array[i + 1] = tmp;
 8           sorted = false;
 9        }
10     }
11     if(sorted) break;
12  }
```

INSERTION SORT

Insertion sort is similar to sorting cards by hand. Each iteration, a new element is inserted into an already sorted sequence until all elements are sorted. The standard insertion sort implementation (Listing 4.11) is not compatible with registers due to the dynamic array access in line 9.

Listing 4.11: Standard Insertion Sort

```
 1  for(int i = 1; i < size; ++i) {
 2     // i elements are sorted
 3     int key = array[i];
 4     int j = i - 1;
 5     while(j >= 0 && array[j] > key) {
 6        array[j+1] = array[j];
 7        --j;
 8     }
 9     array[j+1] = key;
10 }
```

The dynamic access can be replaced with the code described in Listing 4.7 (assigning within a loop) but this would result in two consecutive loops (this will be referred to as insertion v1). Fortunately, both loops can be combined into a single loop by simply swapping the key value into its destination:

Listing 4.12: Register Insertion Sort (Insertion v2)

```
for(int i = 1; i < size; ++i) {
   // i elements are sorted
   for(int j = i; j > 0 && array[j - 1] > array[j]; --j) {
      int tmp = array[j];
      array[j] = array[j - 1];
      array[j - 1] = tmp;
   }
}
```

Figure 4.7 shows the speedup of using registers instead of local memory.

SHELL SORT

Shell sort is a variation of insertion sort. First, the list is roughly pre-sorted by swapping elements from the left side of the array with elements from the right side of the array. After the list was roughly sorted, a normal insertion sort is performed. Since it swaps elements that are far apart at the beginning, the final insertion sort should perform better. Unfortunately, the standard shell sort implementation (Listing 4.13) is not compatible with registers for the same reasons insertion sort was not compatible.

Listing 4.13: Standard Shell Sort

```
// global
const int gaps[] = {20, 9, 4, 1};
const int startGap = SIZE > 20 ? 0 :
               (SIZE > 9 ? 1 :
               (SIZE > 4 ? 2 : 3));
const int maxGaps = gaps.length();
// main
for(int gapIdx = startGap; gapIdx < maxGaps; ++gapIdx) {
   const int gap = gaps[gapIdx];
   for(int i = gap; i < size; ++i) {
      int key = array[i];
      int j = i;
      while(j >= gap && array[j - gap] > key){
         j -= gap;
         array[j] = array[j - gap];
      }
      array[j] = key;
   }
}
```

Using the tricks from insertion sort for shell sort, the following register compatible algorithm arises:

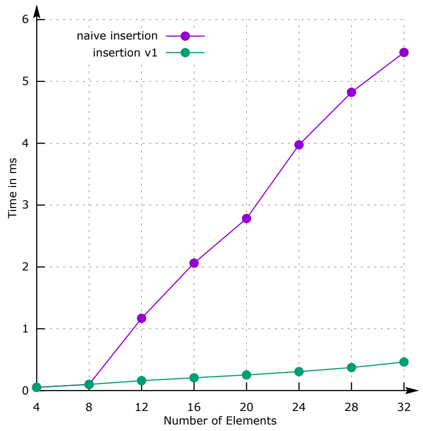


Figure 4.7

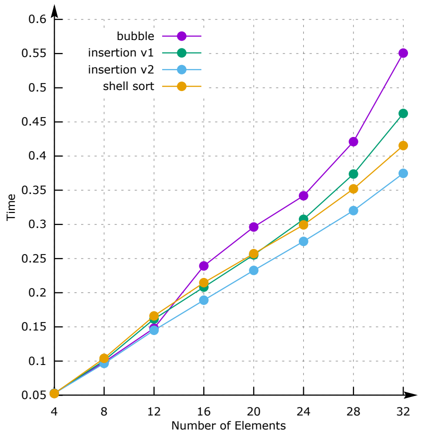


Figure 4.8

Sorting algorithms tested on the GTX 980 TI. Naive insertion refers to the insertion sort from Listing 4.11 which does not work in registers. Insertion v1 is the naive insertion sort that was modified to work in registers. Insertion v2 was introduced in Listing 4.12.

Listing 4.14: Register Shell Sort

```
// main
for(int gapIdx = startGap; gapIdx < maxGaps; ++gapIdx) {
   const int gap = gaps[gapIdx];
   for(int i = gap; i < size; ++i) {
      for(int j = i; j >= gap && array[j - gap] > array[j]; j -= gap)
      {
         int tmp = array[j];
         array[j] = array[j - gap];
         array[j - gap] = tmp;
      }
   }
}
```

Results

The algorithms were tested for arrays between 4 and 32 elements (see Figure 4.8). The register based insertion sort (Listing 4.12) outperforms all other algorithms.

4.6 STENCIL BUFFER

The stencil test can mask out certain areas of the screen which effectively prevents fragment shader invocations. The stencil test is performed per fragment.

MLAB uses a fullscreen pass to blend all per-pixel lists to retrieve the final fragment color. However, the per-pixel lists only need to be blended if any transparent fragments were inserted in the first place. The stencil test can be used to create a mask that prevents unnecessary fragment shader invocations for this scenario.

glStencilFunc(comparison, ref, mask) controls the condition for the stencil test. The stencil test looks like this:

$$\text{test} = (\mathit{ref} \,\&\, \mathit{mask})\ \mathit{comparison}\ (\mathit{stencil} \,\&\, \mathit{mask})$$

stencil is the current value in the stencil buffer, *mask* is a user defined bitmask, *ref* is the reference value for the stencil test and & is a bitwise AND.

Some possible values for *comparison* are:

1. EQUAL: Test passes if $(\mathit{ref} \,\&\, \mathit{mask}) = (\mathit{stencil} \,\&\, \mathit{mask})$
2. ALWAYS: Always passes.
3. GREATER: Test passes if $(\mathit{ref} \,\&\, \mathit{mask}) > (\mathit{stencil} \,\&\, \mathit{mask})$

glStencilOp(sFail, zFail, zPass) describes how the stencil buffer is modified for certain events.

sFail specifies the action to take if the stencil test fails.

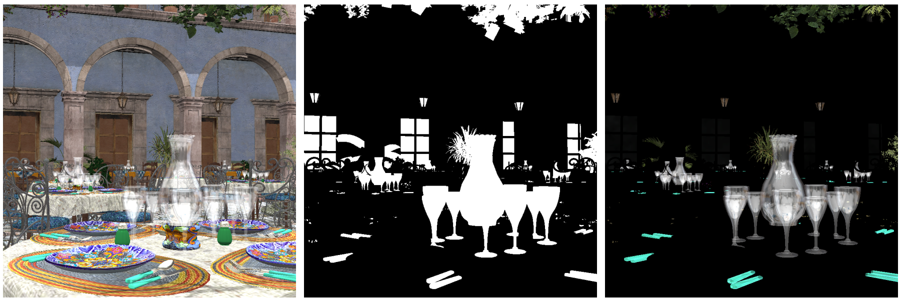
zFail specifies the action to take if the depth test fails.

zPass specifies the action to take if the stencil test and the depth test succeeded.

Some possible actions are:

1. KEEP: The stencil buffer value remains unchanged.
2. REPLACE: The stencil buffer value is replaced by *ref* which was specified by *glStencilFunc(...)*.
3. INCR: The stencil buffer value is incremented by one.

Listing 4.15 shows how to make use of the stencil test for MLAB. First, the stencil buffer is cleared and the opaque geometry is drawn. Thereafter, the stencil test (line 6-8) is configured to write a value of one into the stencil buffer for each transparent fragment (see Figure 4.9). Thus, the stencil test at the end will only succeed for pixels where transparent fragments were drawn (line 11-13). Line 13 draws a quad that covers the entire screen in order to spawn exactly one fragment shader invocation for each pixel that was not masked out by the stencil test.



Scene Stencil Buffer Transparent objects

Figure 4.9: The stencil buffer is used to create a mask from transparent object.

Listing 4.15: Stencil Test in MLAB

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
drawOpaqueGeometry();

// set areas with transparent geometry to 1
glEnable(GL_STENCIL_TEST);
glStencilFunc(GL_ALWAYS, 1, 0xFF);
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
drawTransparentGeometry();

// only invoke shader if a 1 is in the stencil buffer
glStencilFunc(GL_EQUAL, 1, 0xFF);
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
drawFullscreenQuad();
glDisable(GL_STENCIL_TEST);
```

Unfortunately, the stencil test is not very effective for MLAB. The fullscreen pass at the end gets faster but the overhead introduced by the additional stencil test nullifies the gained performance (Table A.4). However, algorithms with more expensive calculations in the fullscreen pass, like UMLAB, should make use of the stencil test to increase performance (Table A.5). The stencil test accelerates UMLAB for up to 5%.

5 UNSORTED MULTI-LAYER ALPHA BLENDING

This chapter presents Unsorted Multi-Layer Alpha Blending (UMLAB), an improved version of MLAB that requires less memory bandwidth making it faster.

5.1 IDEA

The standard MLAB algorithm (Listing 3.1) requires a lot of memory bandwith because the entire per-pixel list is rewritten after the insertion of a single fragment. However, rewriting the entire per-pixel list is not necessary because only the newly inserted fragment and the merged fragment need to be stored.

5.2 ALGORITHM

UMLAB does no longer store the per-pixel list in a sorted order to avoid re-writing the entire per-pixel list after a fragment insertion. This changes the algorithm for the fragment insertion (see Listing 5.1). First, the two fragments with the highest depth values are determined (line 10-33). Then, both fragments are merged similar to the standard MLAB. Finally, only the inserted fragment and the merged fragment are written into the per-pixel list (line 39-50). The inserted value is not stored if it was involved in the merge.

Listing 5.1: UMLAB insertion

```
1   void insert(Fragment f){
2      Fragment frags[n + 1];
3      frags[n] = f;
4
5      // load per-pixel array
6      for(int i = 0; i < n; ++i)
7         frags[i+1] = LOAD(i);
8
9      // fragment with highest depth
10     Fragment maxFrag;
11     maxFrag.Z = -1.0;
12     int maxFragIdx = 0;
13
14     // fragment with second highest depth
15     Fragment smaxFrag;
16     smaxFrag.Z = -1.0;
17    int smaxFragIdx = 0;
18
19    // find maximum
20    for(int i = 0; i <= n; ++i) {
21       if(frags[i].Z > maxFrag.Z) {
22          maxFrag = frags[i];
23          maxFragIdx = i;
24       }
25    }
26
27    // find other maximum
28    for(int i = 0; i <= size; ++i) {
29       if(frags[i].Z > smaxFrag.Z && i != maxFragIdx){
30          smaxFrag = frags[i];
31          smaxFragIdx = i;
32       }
33    }
34
35    // merge the two highest fragments
36    vec2 merged = merge(smaxFrag, maxFrag);
37
38    // store changed values
39    if(maxFragIdx == n) {
40       // the inserted value was merged
41       // just overwrite the second highest value
42       maxFragIdx = smaxFragIdx;
43       // prevent the inserted value from being stored
44       // (because it was merged)
45       smaxFragIdx = n;
46    }
47
48    STORE(maxFragIdx, merged);
49    if(smaxFragIdx != n)
50       STORE(smaxFragIdx, f);
51 }
```

Since the per-pixel is no longer sorted, it needs to be sorted before blending all transparent fragments for the final color in the fullscreen pass. This can be done with the register based insertion sort from Chapter 4.

Additionally, the insertion of the first n fragments can be accelerated since at least one initialization node is in the per-pixel list and compression is not required. Listing 5.2 can be put after line 33 of the UMLAB code to accelerate the insertion of the first n fragments.

Table 5.1: MLAB and UMLAB rendering times in milliseconds (NVIDIA GTX 1080). The interleaved SSBO was used for the Powerplant scene. A 3D Texture was used for the other two scenes, because it was faster than the interleaved SSBO. San Miguel times exclude the opaque rendering times.

SCENE(NODES):	MLAB	UMLAB
Powerplant(4)	15.22	**14.75**
Powerplant(8)	16.94	**15.40**
Powerplant(12)	19.06	**16.27**
Village(4)	1.29	**1.14**
Village(8)	1.74	**1.34**
Village(12)	1.97	**1.55**
San Miguel(4)	2.19	**2.15**
San Miguel(8)	2.30	**2.27**
San Miguel(12)	2.54	**2.36**

Listing 5.2: UMLAB early insertion

```
// is a default value in this list?
if(maxFragIdx.Z == FLOAT_MAX) {
   // only insert the new fragment (no need to merge)
   STORE(maxFragIdx, f);
   return;
}
```

5.3 RESULTS

UMLAB generally performs better than MLAB (see Table 5.1). Depending on the scene, UMLAB is up to 30% faster (Village(8)).

6 ADAPTIVE TRANSPARENCY OPTIMIZATIONS

Similar to MLAB, the standard AT algorithm requires a lot of memory bandwidth because the entire per-pixel list is rewritten after the insertion of a single fragment. The following algorithms try to reduce the bandwidth by storing the nodes in a different way than AT.

First of all, the nodes are no longer stored in a sorted order to avoid re-writing the entire per-pixel list after fragment insertion. Unfortunately, that alone is not enough because AT modifies all nodes with a bigger depth than the inserted fragment (see Figure 3.11). This happens because each node stores the absolute height of the visibility function. However, instead of storing the absolute height, storing the relative height to the previous node is enough.

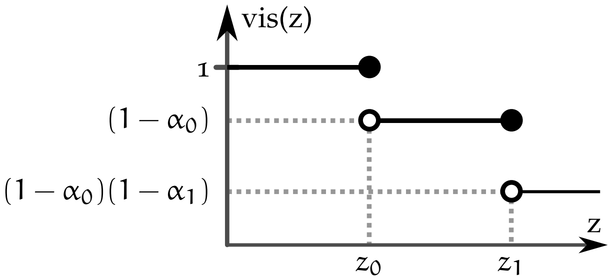


Figure 6.1: Example of the visibilty function.

Example: Take Figure 6.1. AT would store two nodes: $(z_0, (1-\alpha_0))$ and $(z_1, (1-\alpha_0)(1-\alpha_1))$. However, it would suffice to store: $(z_0, (1-\alpha_0))$ and $(z_1, (1-\alpha_1))$. The original visibility function height of a node can be reconstructed by multiplying the relative heights of all previous nodes. If the nodes are stored this way, nodes with a bigger depth than the inserted fragment do not need to be modified anymore.

Unfortunately, due to the node compression strategy of AT, the order of nodes must be reconstructed within the shader to determine the areas between nodes for the underestimation.

6.1 UNSORTED ADAPTIVE TRANSPARENCY

Unsorted Adaptive Transparency (UAT) stores the previously described visibility nodes in an unsorted array. In the shader, the local node array gets sorted according to the node depth. Afterward, the fragment compression is performed similar to AT. Finally, up to two nodes are written back into the global per-pixel list. Those nodes are the inserted

Table 6.1: AT and UAT rendering times (ms) in the Powerplant and the Village scene (NVIDIA GTX 1080). The interleaved SSBO was used to store the nodes.

NODES	AT(PPLANT)	UAT(PPLANT)	AT(VILL)	UAT(VILL)
8	**20.07**	20.13	**1.82**	2.13
12	**22.80**	23.91	**2.15**	2.62
16	**25.59**	30.54	**2.58**	3.25

node and the node, that was compressed by the underestimation. The inserted node replaces the node that was removed due to the compression.

Unfortunately, the overhead from sorting the array before each insertion dominates this algorithm (see Table 6.1). Thus, the default AT was either equal to or faster than UAT in all tested scenes.

6.2 LINKED ARRAY ADAPTIVE TRANSPARENCY

Linked Array Adaptive Transparency (LAAT) stores for each node an additional link to the next node to avoid using a sorting algorithm with quadratic runtime (see Figure 6.2). The fragment insertion works similar to the UAT fragment insertion. However, instead of sorting the nodes, the ordering is reconstructed by loading the nodes according to the linked list. After the fragment compression, up to three nodes are written back into the global per-pixel list. Since the link to the compressed node can change as well, LAAT stores one node more than UAT.

Unfortunately, LAAT performs even worse than UAT (see Table 6.2). The main drawback of LAAT is the reconstruction of the node ordering from the linked list. Reading from random locations in global memory seems to be a bad idea.

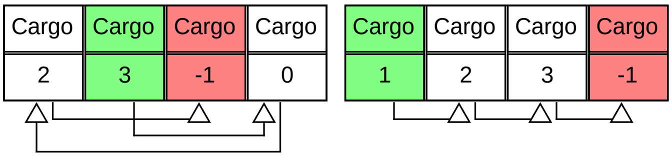


Figure 6.2: Nodes for LAAT. In addition to the nodes cargo (depth and transmittance) the index of the next node is included as well. An index of minus one indicates the end of the per-pixel list.

Table 6.2: AT and LAAT rendering times (ms) in the Powerplant and Village scene (NVIDIA GTX 1080). The interleaved SSBO was used for AT. The texture was used for LAAT.

NODES	AT(PPLANT)	LAAT(PPLANT)	AT(VILL)	LAAT(VILL)
8	**20.07**	24.57	**1.82**	2.49
12	**22.80**	30.21	**2.15**	3.30
16	**25.59**	38.81	**2.58**	4.43

6.3 HEIGHT ADAPTIVE TRANSPARENCY

For the next approach, a different node compression metric was tested. Previously, the area between two nodes determined the node for compression.

This compression metric minimizes the integration error by removing the node with the least contribution to the visibility function. The integration error can be describes as:

$$E_{\text{integration}} = \int_0^\infty (\text{vis}_t(z) - \text{vis}_c(z))\, dz$$

With $\text{vis}_t(z)$ being the true visibility function and $\text{vis}_c(z)$ being the compressed function. However, the integration error is not the same as the visible error because fragments are blended according to the following formula:

$$C_{\text{visible}} = \sum_{i=0}^{n-1} \alpha_i c_i \text{vis}(z_i)$$

Therefore, if we assume that all fragment colors are similar, the visible error is proportional to:

$$E_{\text{visible}} = \sum_{i=1}^{n-1} \alpha_i \left(\text{vis}_t(z_i) - \text{vis}_c(z_i)\right)$$

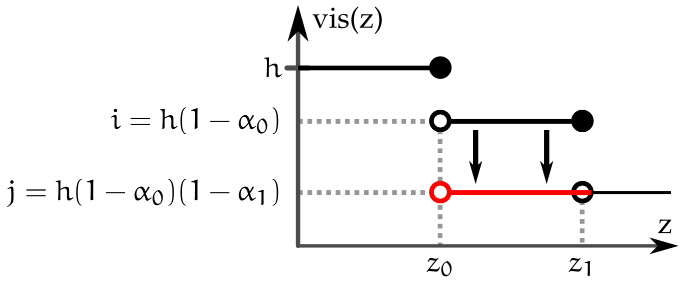


Figure 6.3: Underestimation (red) of the visibility function.

The uncompressed visibility function can be described as:

$$\text{vis}_t(z) := \prod_{\forall(z_i,\alpha_i):z_i<z} (1-\alpha_i)$$

Thus, each node in the visibility function is the product of the transmittance of fragments with a smaller depth value. Using this information, the visible error caused by an underestimation can be calculated as follows (see Figure 6.3):

$$\begin{aligned}
\text{vis}_t(z_1) &= h(1-\alpha_0) = i \\
\text{vis}_c(z_1) &= h(1-\alpha_0)(1-\alpha_1) = j \\
\alpha_1 &= 1 - \frac{h(1-\alpha_0)(1-\alpha_1)}{h(1-\alpha_0)} = 1 - \frac{j}{i} \\
E_{\text{visible}} &= \alpha_1(\text{vis}_t(z_1) - \text{vis}_c(z_1)) \\
&= \left(1 - \frac{j}{i}\right)(i-j) = \frac{(i-j)^2}{i} \qquad (6.1)
\end{aligned}$$

h denotes the visibility function at z_0 ($h = \text{vis}(z_0)$). In this case, the fragment affected by the underestimation was (c_1, α_1, z_1). Therefore, a compression at the node which causes the smallest visible error E_{visible} should produce better results. However, the visible error for a node is only correctly computed, if that node was not previously compressed. Unfortunately, the error computed with Equation 6.1 is smaller than the actual error, if the node (in this case (z_0, α_0)) was previously compressed (proof: Section A.1). Therefore, it is likely that the visible error is not kept minimal after multiple compressions.

The new node compression metric will be referred to as *height metric* because $i - j$ describes the height difference between two nodes. The original node compression metric will be referred to as *rect metric* since it uses the rectangular area between two nodes. In the following, the *rect metric* and the *height metric* are compared.

Figure 6.4 shows a comparison of the two compression metrics. The rect metric produces a smaller integration error but the height metric is almost always better concerning the visible error.

The new metric has an interesting characteristic that can be seen in the graph: The first node is compressed by the rect metric because the area between the first and the second node is relatively small. However, the height metric decides to preserve the first node because the difference in height to the second node is relatively high. The height metric completely ignores the depth distance between two nodes.

Ignoring the depth has an advantage and a disadvantage. On one hand, very close nodes from fragments with a high opacity are likely to remain uncompressed (see Figure 6.5). This is generally a good trait, because those fragments usually have a high contribution to the final pixel color. On the other hand, underestimating large depth ranges causes all future fragments that are within that range to be biased as well (see Figure 6.5 bottom graph). This is bad if the fragments within that range have a high contribution to the visibility function.

Figure 6.6 shows a comparison of the two metrics in the powerplant and the village scene with 8 nodes. HAT performs better in both scenes. Especially the roofs of the village scene are less biased in HAT. This probably happens due to the high density of fragments in the roof because AT is likely to merge important fragments if they are close together.

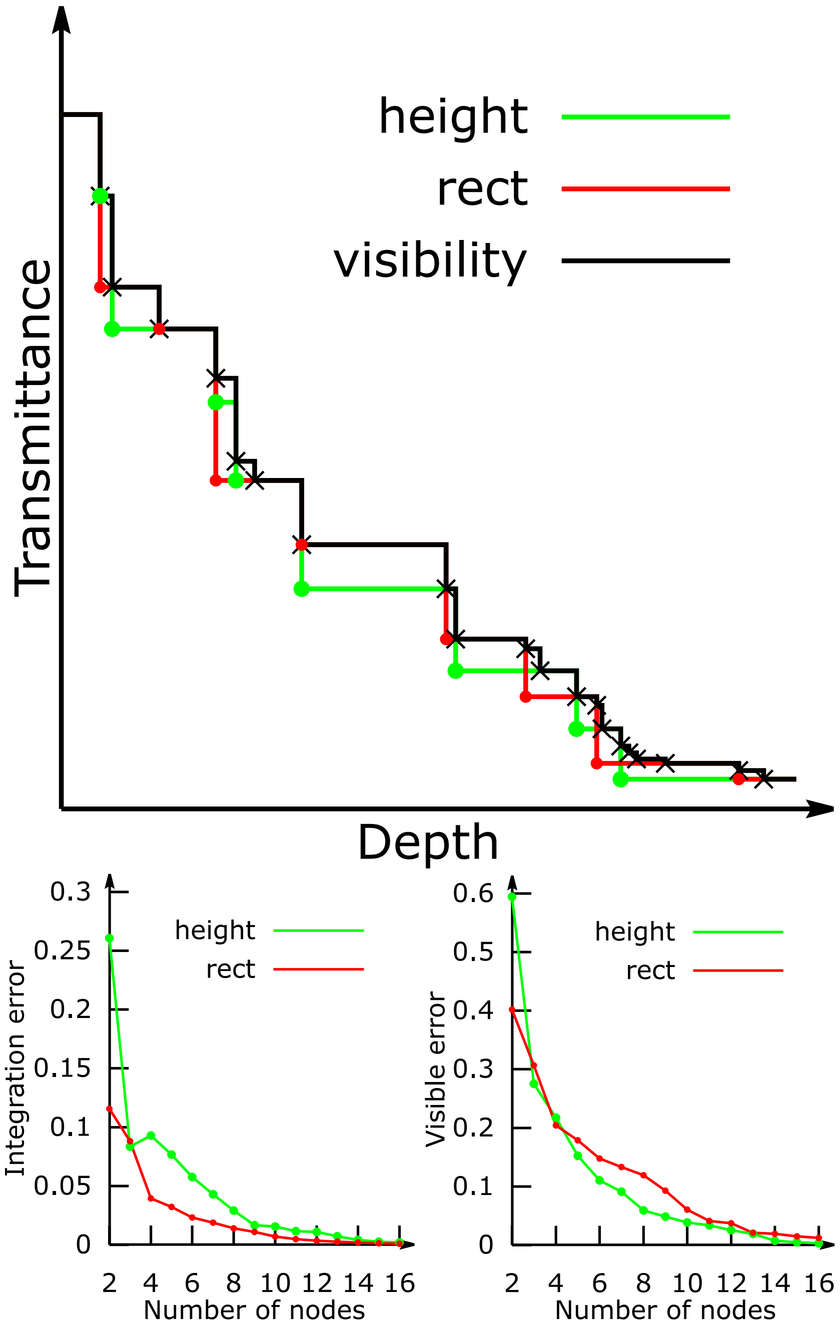


Figure 6.4: Comparison of the default compression metric (rect) and the new compression metric (height) in a scene with 20 transparent fragments with $\alpha \in [0.05, 0.25]$.

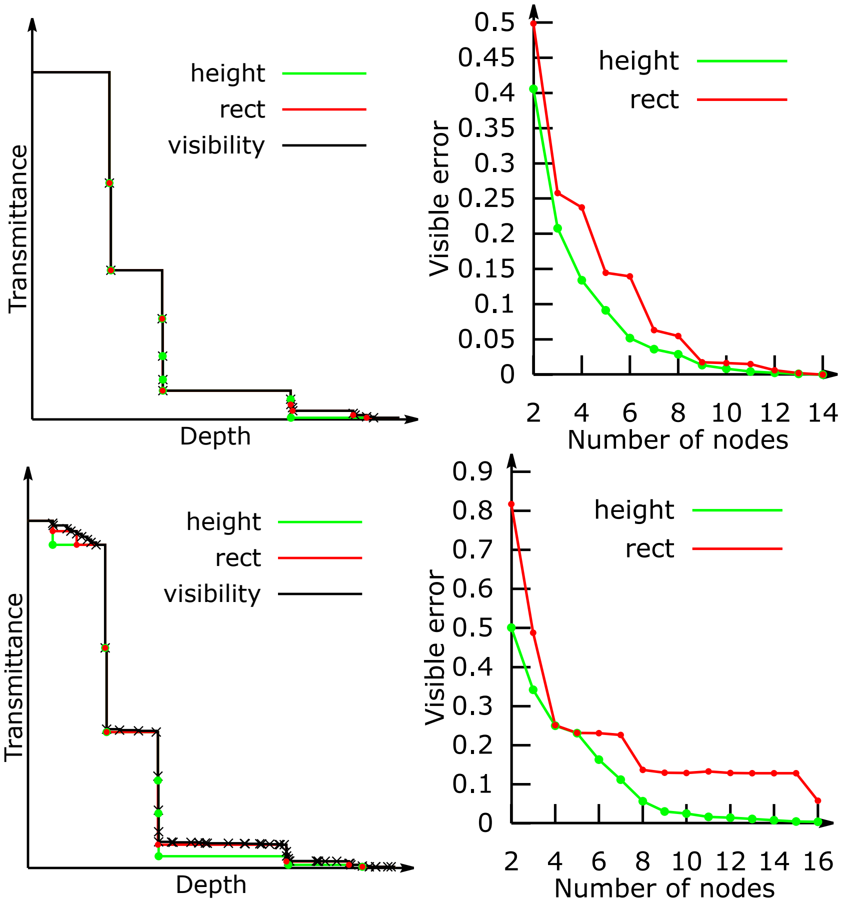
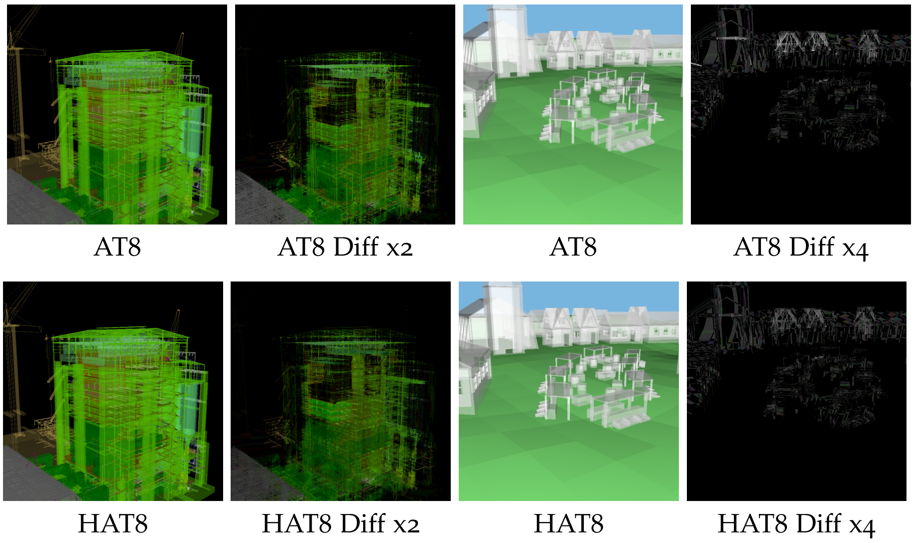


Figure 6.5: Rect and height compression in a scene with tightly packed fragments. The second scene contains 40 transparent fragments with very low opacity ($\alpha \in [0.005, 0.01]$).



AT8 AT8 Diff x2 AT8 AT8 Diff x4

HAT8 HAT8 Diff x2 HAT8 HAT8 Diff x4

Figure 6.6: AT and HAT comparison with 8 nodes.

6.4 UNSORTED HEIGHT ADAPTIVE TRANSPARENCY

Unsorted Height Adaptive Transparency (UHAT) stores the AT nodes in an unsorted array and compresses the nodes according to the height metric. First of all, the height metric needs to be reformulated to be compatible with the unsorted node storage. Note, that UHAT stores the relative height of the visibility function in its nodes as described in the beginning of Chapter 6. The visible error can be rewritten as follows:

$$\begin{aligned} E_{\text{visible}} &= \alpha_1(\text{vis}_t(z_1) - \text{vis}_c(z_1)) \\ &= \alpha_1(h(1-\alpha_0) - h(1-\alpha_0)(1-\alpha_1)) \\ &= h(1-\alpha_0)\alpha_1^2 \qquad (6.2) \\ &= \text{vis}_t(z_1)\alpha_1^2 \qquad (6.3) \end{aligned}$$

Unfortunately, Equation 6.2 requires the knowledge of all previous nodes ($h(1-\alpha_0)$) to determine the visible error. However, if we can approximate the current visibility function $\text{vis}_t(z)$, Equation 6.3 can be used to calculate the error for a single node without the knowledge of any other node.

An approximate visibility function can be described by an exponential function (see Figure 6.7). If all objects have the same opacity and are uniformly distributed throughout the scene, an exponential function is able to describe the visibility function without error. Two points are needed to determine the exponential function equation $f(x) = ae^{bx}$. The first point is $f(z_{\text{first}}) = 1$ with z_{first} being the depth of the first node. The second point is $f(z_{\text{last}}) = (\prod_{i=0}^{n-1}(1-\alpha_i))/(1-\alpha_{\text{last}})$ with $(z_{\text{last}}, \alpha_{\text{last}})$ being the last node. Thus, the exponential function described by the two points is:

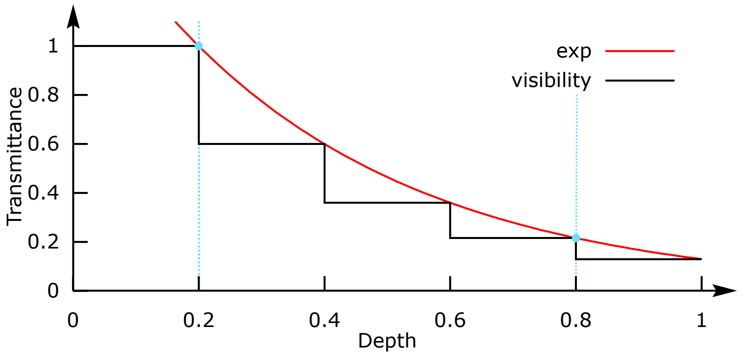


Figure 6.7: Visibility function approximated by an exponential function.

$$f(x) = \exp\left(\frac{\log \frac{\prod_{i=0}^{n-1}(1-\alpha_i)}{(1-\alpha_{\text{last}})}}{z_{\text{last}} - z_{\text{first}}}(x - z_{\text{first}})\right)$$

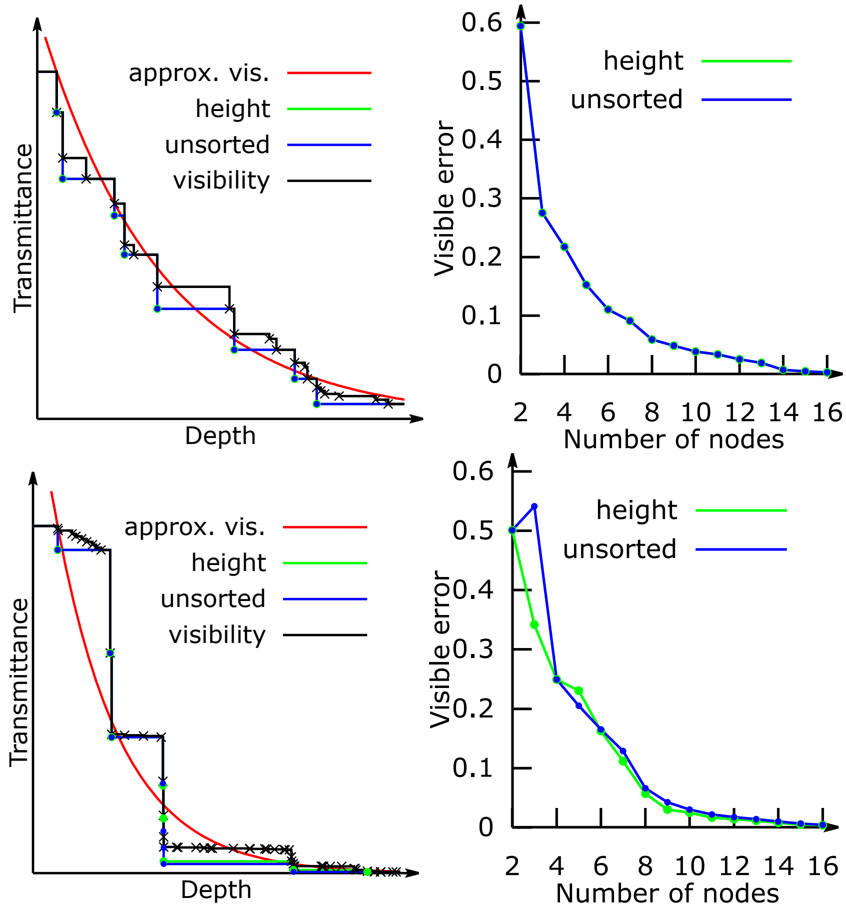


Figure 6.8: HAT (green) and UHAT (blue) in a scene with evenly distributed fragments (top) and unevenly distributed fragments (bottom). Approx. vis. is the visibility function approximation based on the UHAT nodes.

The new insertion algorithm is described in Listing 6.1. Line 9-29 determine the relevant parameters that are required to determine the exponential function. Line 39-42 offers a fast insertion, if no compression is required because one of the initialization values is inside the per-pixel list. The exponential function (line 1-5) is initialized in line 46-48. Thereafter, the node that produces the least visible error according to Equation 6.3 is determined (line 52-62). Afterwards, the predecessor of that node is determined (line 65-75). Finally, the visibility function gets underestimated (line 78) and the changed nodes are written back into the per-pixel list (line 81-88).

Listing 6.1: UHAT insertion

```
1  float g_visExponent = 0.0f;
2  float g_visOffset = 0.0f;
3  float vis(float x) {
4     return exp(g_visExponent * (x + g_visOffset));
```

```
}

void insert(Fragment f){
   // determine visibility function
   float maxDepth = depth;
   float minDepth = depth;
   float productAlpha = f.T;
   float lastAlpha = f.T;
   int maxIndex = n;
   int minIndex = n;

   Fragment fragments[n+1];
   fragments[n] = f;

   // load all fragments
   for(int i = 0; i < n; ++i){
      fragments[i] = LOAD(i);
      productAlpha *= fragments[i].T;

      // determine last node
      if(fragments[i].Z > maxDepth) {
         maxDepth = fragments[i].Z;
         lastAlpha = fragments[i].T;
         maxIndex = i;
      }

      // determine first node
      if(DEPTH(fragments[i]) < minDepth) {
         minDepth = fragments[i].Z;
         minIndex = i;
      }
   }

   // replace initialization values first
   if(maxDepth == FLOAT_MAX){
      STORE(maxIndex, f);
      return;
   }

   // store visibility function
   // avoid dividing by zero
   const float e = 0.00000001;
   g_visExponent = log(max(productAlpha / max(lastAlpha, e), e)) / max(↩
       maxDepth - minDepth, e);
   g_visOffset = -minDepth;

   // find the node with the smallest height difference.
   // this node will be removed
   float minHeight = FLOAT_MAX;
   int removeIndex = 0;
   Fragment removeFragment = fragments[0];
   for(int i = 0; i <= n; ++i){
      float height = vis(fragments[i].Z) * (1.0f -  fragments[i].T) * ↩
           (1.0f -  fragments[i].T);
      if(height < minHeight && i != minIndex){
         minHeight = height;
         removeIndex = i;
         removeFragment = fragments[i];
      }
   }

   // find the predecessor
   int compressIndex = -1;
   Fragment compressFragment;
   compressFragment.Z = -1.0;
   for(int i = 0; i <= n; ++i){
      if(i != removeIndex &&
         compressFragment.Z >= fragments[i].Z &&
         fragments[i].Z <= removeFragment.Z) {
         compressIndex = i;
         compressFragment = fragments[i];
      }
   }

   // adjust transmittance with underestimation
   compressFragment.T *= removeFragment.T;

   // store changed nodes
   if(compressIndex == n){
      STORE(removeIndex, compressFragment);
   }
   else {
      STORE(compressIndex, compressFragment);
      if(removeIndex != n)
         STORE(removeIndex, f);
   }
}
```

Figure 6.8 shows the results of UHAT next to the results of HAT. If the visibility function is similar to an exponential function (Figure 6.8 top) both algorithms produce the same results. Otherwise, UHAT still manages to produce good results which are similar to HAT (Figure 6.8 bottom).

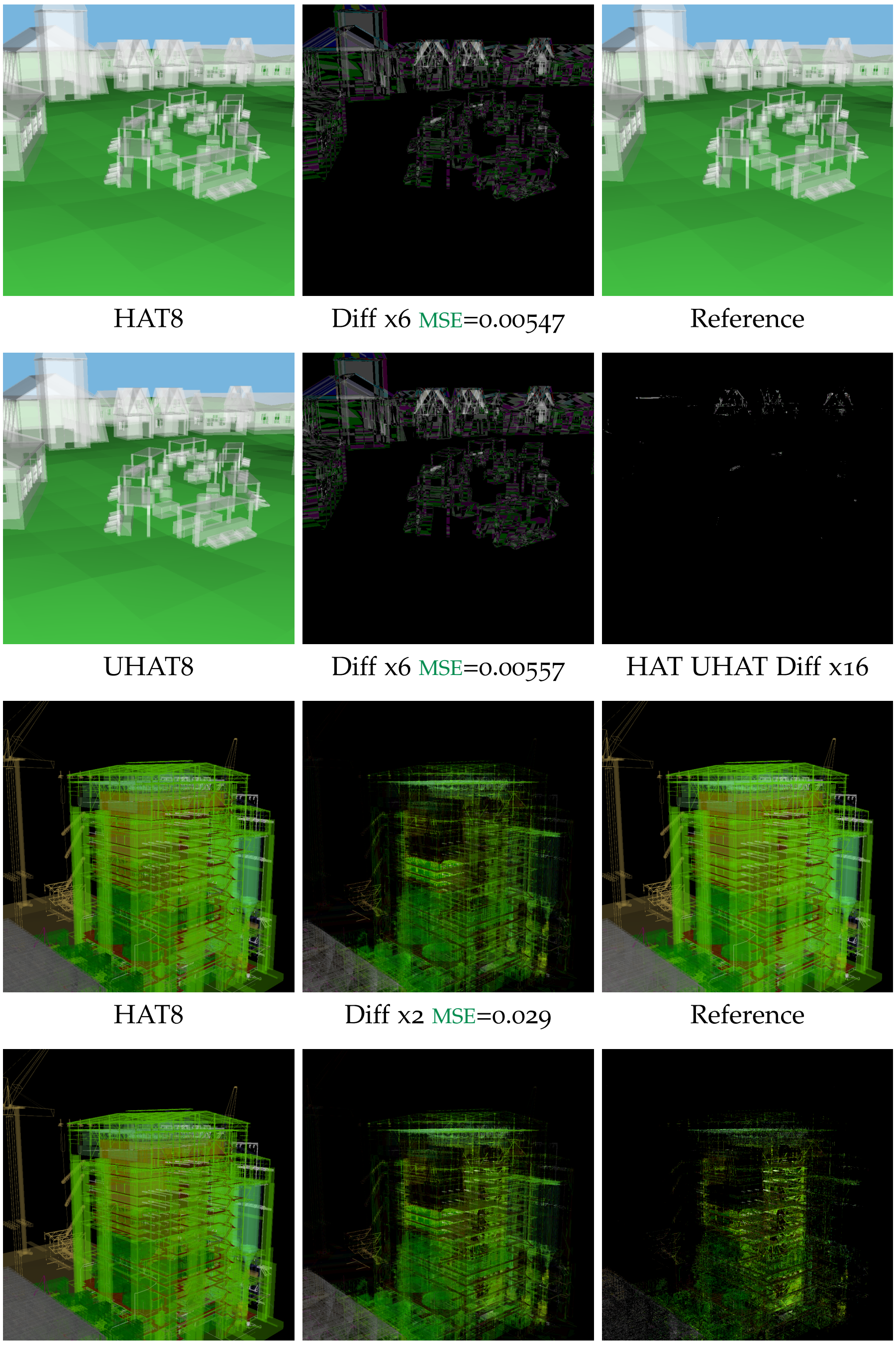
Figure 6.9 shows the results of both renderers in the village and powerplant scene. Overall, the approximation of the visibility function in UHAT produces images that are very similar to images produced by HAT for scenes with a medium amount of transparent objects (Village and San Miguel). Scenes with a high amount of transparent objects (e.g. Powerplant) are more difficult for UHAT and the difference to HAT becomes visible.

UHAT is faster than AT/HAT due to the reduced memory bandwidth usage (see Table 6.3). UHAT is up to 24% faster than HAT with 16 nodes. Unfortunately, UHAT with 8 nodes is only about 3% faster. Therefore, HAT is the better choice if only a small number of nodes ($\leqslant 8$) is

Table 6.3: AT and UHAT rendering times (ms) in the Powerplant and Village scene (NVIDIA GTX 1080). The interleaved SSBO was used for the Powerplant and the texture was used for the Village scene.

NODES	HAT(PPLANT)	UHAT(PPLANT)	HAT(VILL)	UHAT(VILL)
8	20.07	**19.38**	1.77	**1.72**
12	22.80	**20.76**	2.22	**1.93**
16	25.59	**22.20**	2.66	**2.14**

required for the scene because UHAT would produce a higher visual error with very low performance gain. However, if a higher resolution of the visibility function is required, UHAT provides a similar result as HAT in less time.



HAT8

Diff x6 MSE=0.00547

Reference

UHAT8

Diff x6 MSE=0.00557

HAT UHAT Diff x16

HAT8

Diff x2 MSE=0.029

Reference

UHAT8

Diff x2 MSE=0.035

HAT UHAT Diff x4

Figure 6.9: HAT and UHAT comparison with 8 nodes.

7 CONCLUSION AND FUTURE WORK

In this thesis several methods to accelerate OIT algorithms were investigated.

The interleaved SSBO and the 3D texture are fast data structures that provide good data locality for concurrent computations. However, it is still unclear under which circumstances the interleaved SSBO performs better than the 3D texture. Furthermore, the data locality of the interleaved SSBO can probably be improved by using a space filling curve for node groups.

Using registers instead of shared memory for OIT algorithms significantly improves the rendering time. Techniques like loop unrolling are especially useful to ensure register usage. The register based insertion sort provides a very fast method to sort small lists with a fixed size.

Using the stencil buffer to mask out transparent areas is useful for OIT algorithms with expensive calculations in the fullscreen pass. UMLAB and UHAT are using the stencil buffer because both algorithms perform a sort in the fullscreen pass.

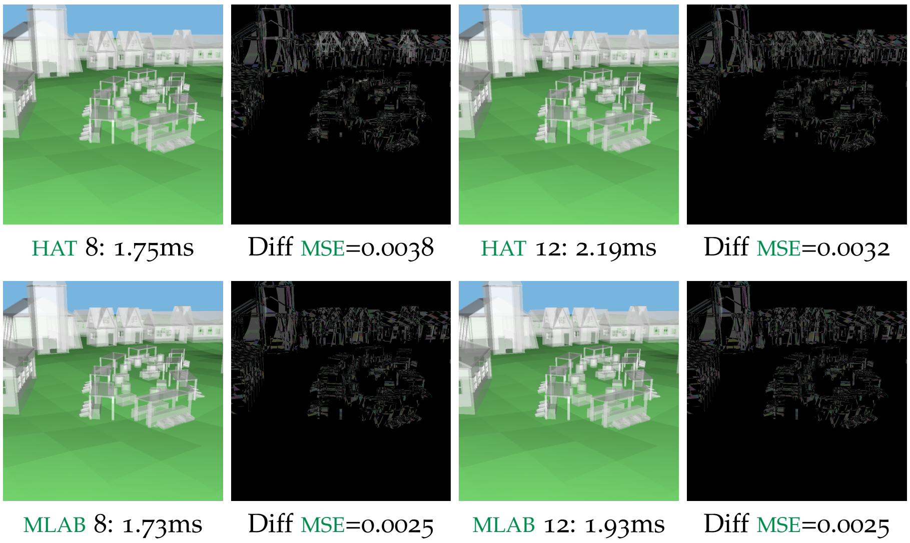
UMLAB introduced a variation of MLAB that minimizes memory bandwidth usage and therefore performs better than the original MLAB algorithm.

In Chapter 6 I tried to accelerate an OIT algorithm that requires its nodes to be sorted in order to work properly. Unfortunately, the unsorted storage (UAT) and the linked array (LAAT) only introduced more overhead to the existing algorithm.

HAT redefined the compression metric of AT to minimize the visible error. In general, the error introduced by HAT is either similar or lower than the error from AT and the performance is the same. Unfortunately, I have not tested those techniques in a scene with fog or hair. Those scenarios are interesting because ignoring the depth for the node compression might be a bad idea for a scene with uniformly distributed fog/hair.

UHAT is a faster alternative to HAT for constructing a higher resolution visibility function (more than eight nodes). However, HAT is still the better alternative for lower resolution visibility functions and it produces transparency images with more accuracy in complex scenes.

MLAB and UMLAB are better techniques for OIT than AT and HAT. They produce images with higher quality (Figure 7.1) in less time and they need less nodes (between 4 and 8) for good results.



HAT 8: 1.75ms

Diff MSE=0.0038

HAT 12: 2.19ms

Diff MSE=0.0032

MLAB 8: 1.73ms

Diff MSE=0.0025

MLAB 12: 1.93ms

Diff MSE=0.0025

Figure 7.1: Comparison between HAT and MLAB. The diff-pictures describe the difference to the sorted alpha blending solution multiplied by 8.

Note that everything was tested on the NVIDIA GTX 1080 and the results may differ between different GPUs. However, most optimizations should perform better on all GPUs. Registers are always faster than local or global memory and higher data locality will always maximize cache performance.

BIBLIOGRAPHY

[BCL+07] L. Bavoil, S. P. Callahan, A. Lefohn, J. a.L. D. Comba, and C. T. Silva. “Multi-fragment Effects on the GPU Using the K-buffer.” In: *Proceedings of the Symposium on Interactive 3D Graphics and Games*. I3D ’07. New York, NY, USA: ACM, 2007, pp. 97–104. ISBN: 978-1-59593-628-8. URL: http://doi.acm.org/10.1145/1230100.1230117.

[BM08] L. Bavoil and K. Myers. *Order independent transparency with dual depth peeling*. Tech. rep. 2008.

[Car84] L. Carpenter. “The A-buffer, an Antialiased Hidden Surface Method.” In: *Proceedings of the Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’84. ACM, 1984, pp. 103–108. ISBN: 0-89791-138-5. URL: http://doi.acm.org/10.1145/800031.808585.

[ESSL10] E. Enderton, E. Sintorn, P. Shirley, and D. Luebke. “Stochastic Transparency.” In: *Proceedings of the Symposium on Interactive 3D Graphics and Games*. I3D ’10. Washington, D.C.: ACM, 2010, pp. 157–164. ISBN: 978-1-60558-939-8. URL: http://doi.acm.org/10.1145/1730804.1730830.

[Eve01] C. Everitt. *Interactive Order-Independent Transparency*. 2001.

[Har13] M. Harris. *How to Access Global Memory Efficiently in CUDA C/C++ Kernels*. https://devblogs.nvidia.com/how-access-global-memory-efficiently-cuda-c-kernels/. Accessed 2-July-2018. Jan. 2013.

[Hub91] D. Hubert. “Ueber die stetige Abbildung einer Linie auf ein Flächenstück.” German. In: *Mathematische Annalen* 38 (1891), pp. 459–460.

[JB10] J. Jansen and L. Bavoil. “Fourier Opacity Mapping.” In: *Proceedings of the Symposium on Interactive 3D Graphics and Games*. I3D ’10. Washington, D.C.: ACM, 2010, pp. 165–172. ISBN: 978-1-60558-939-8. URL: http://doi.acm.org/10.1145/1730804.1730831.

[KN01] T.-Y. Kim and U. Neumann. “Opacity Shadow Maps.” In: *Proceedings of the Eurographics Workshop on Rendering Techniques*. Springer-Verlag, 2001, pp. 177–182. ISBN: 3-211-83709-4. URL: http://dl.acm.org/citation.cfm?id=647653.732282.

[Kno15] P. Knowles. "Real-Time deep image rendering and order independent transparency." Doctor of Philosophy (PhD). RMIT University, 2015.

[KLZ14] P. Knowles, G. Leach, and F. Zambetta. "Fast Sorting for Exact OIT of Complex Scenes." In: *Vis. Comput.* 30.6-8 (June 2014), pp. 603–613. ISSN: 0178-2789. URL: http://dx.doi.org/10.1007/s00371-014-0956-z.

[Kub15] C. Kubisch. *Life of a triangle - NVIDIA's logical pipeline*. https://developer.nvidia.com/content/life-triangle-nvidias-logical-pipeline. Accessed 28-June-2018. Mar. 2015.

[LWXW09] B. Liu, L. Y. Wei, Y. Q. Xu, and E. Wu. "Multi-layer depth peeling via fragment sort." In: *Proceedings of the IEEE International Conference on Computer-Aided Design and Computer Graphics*. 2009, pp. 452–456. DOI: 10.1109/CADCG.2009.5246861.

[LHLW09] F. Liu, M.-C. Huang, X.-H. Liu, and E.-H. Wu. "Efficient Depth Peeling via Bucket Sort." In: *Proceedings of the Conference on High Performance Graphics 2009*. HPG '09. New Orleans, Louisiana: ACM, 2009, pp. 51–57. ISBN: 978-1-60558-603-8. DOI: 10.1145/1572769.1572779. URL: http://doi.acm.org/10.1145/1572769.1572779.

[MCTB13] M. Maule, J. a. Comba, R. Torchelsen, and R. Bastos. "Hybrid Transparency." In: *Proceedings of the Symposium on Interactive 3D Graphics and Games*. I3D '13. Orlando, Florida: ACM, 2013, pp. 103–118. ISBN: 978-1-4503-1956-0. URL: http://doi.acm.org/10.1145/2448196.2448212.

[MCTB12] M. Maule, J. L. D. Comba, R. Torchelsen, and R. Bastos. "Memory-Efficient Order-Independent Transparency with Dynamic Fragment Buffer." In: *Proceedings of the SIBGRAPI Conference on Graphics, Patterns and Images*. SIBGRAPI '12. IEEE Computer Society, 2012, pp. 134–141. ISBN: 978-0-7695-4829-6. URL: http://dx.doi.org/10.1109/SIBGRAPI.2012.27.

[MB13] M. McGuire and L. Bavoil. "Weighted Blended Order-Independent Transparency." In: *Journal of Computer Graphics Techniques (JCGT)* 2.2 (2013), pp. 122–141. ISSN: 2331-7418. URL: http://jcgt.org/published/0002/02/09/.

[Mes07] H. Meshkin. "Sort-independent alpha blending." In: Perpetual Entertainment, 2007.

[Mor66] G. Morton. *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*. International Business Machines Company, 1966.

[MKKP18] C. Münstermann, S. Krumpen, R. Klein, and C. Peters. "Moment-Based Order-Independent Transparency." In: *Proceedings of Computer Graphics and Interactive Techniques* 1.1 (May 2018), 7:1–7:20.

[NVI] NVIDIA. *NVIDIA GeForce GTX 980 (whitepaper)*. URL: http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF.

[PD84] T. Porter and T. Duff. "Compositing Digital Images." In: *SIGGRAPH Computer Graphics* 18.3 (Jan. 1984), pp. 253–259. ISSN: 0097-8930. URL: http://doi.acm.org/10.1145/964965.808606.

[SML11] M. Salvi, J. Montgomery, and A. Lefohn. "Adaptive Transparency." In: *Proceedings of the Symposium on High Performance Graphics*. HPG '11. Vancouver, British Columbia, Canada: ACM, 2011, pp. 119–126. ISBN: 978-1-4503-0896-0. URL: http://doi.acm.org/10.1145/2018323.2018342.

[SV14] M. Salvi and K. Vaidyanathan. "Multi-layer Alpha Blending." In: *Proceedings of the Symposium on Interactive 3D Graphics and Games*. I3D '14. ACM, 2014, pp. 151–158. ISBN: 978-1-4503-2717-6. URL: http://doi.acm.org/10.1145/2556700.2556705.

[SA09] E. Sintorn and U. Assarsson. "Hair Self Shadowing and Transparency Depth Ordering Using Occupancy Maps." In: *Proceedings of the Symposium on Interactive 3D Graphics and Games*. I3D '09. ACM, 2009, pp. 67–74. ISBN: 978-1-60558-429-4. URL: http://doi.acm.org/10.1145/1507149.1507160.

[VF14] A. A. Vasilakis and I. Fudos. "K+-buffer: Fragment Synchronized K-buffer." In: *Proceedings of the Symposium on Interactive 3D Graphics and Games*. I3D '14. ACM, 2014, pp. 143–150. ISBN: 978-1-4503-2717-6. URL: http://doi.acm.org/10.1145/2556700.2556702.

[VVPM17] A.-A. Vasilakis, K. Vardis, G. Papaioannou, and K. Moustakas. "Variable k-buffer using Importance Maps." In: *EG 2017 - Short Papers*. Ed. by A. Peytavie and C. Bosch. The Eurographics Association, 2017. DOI: 10.2312/egsh.20171005.

[YHGT10] J. C. Yang, J. Hensley, H. Grün, and N. Thibieroz. "Real-time Concurrent Linked List Construction on the GPU." In: *Proceedings of the Eurographics Conference on Rendering*. EGSR'10. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2010, pp. 1297–1304. URL: http://dx.doi.org/10.1111/j.1467-8659.2010.01725.x.

[YK08] C. Yuksel and J. Keyser. "Deep Opacity Maps." In: *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2008)* 27.2 (2008), pp. 675–680. DOI: 10.1111/j.1467-8659.2008.01165.x.

A

APPENDIX

Table A.1: Naive insertion sort (Listing 4.11) with different dynamic index assignments (for line 9). Rendering times in ms on the NVIDIA GTX 1080.

SAMPLES:	DIRECT	LOOP	SWITCH	IF ELSE
4	0.0584	**0.0573**	**0.0573**	**0.0573**
8	0.1116	**0.1065**	0.1096	0.1085
16	2.5999	**0.2068**	0.2410	0.2130
32	6.3600	**0.8335**	0.9789	1.0424

Table A.2: Insertion of a single element into a sorted sequence with and without early out (see Listing 4.8 and Listing 4.9). Rendering times in ms on the NVIDIA GTX 1080. MLAB was used and the insertion within the shader was repeated 128 times for the Village and 16 times for the Powerplant in order to measure a difference (the difference for one insertion was not measurable).

VILLAGE	NO EARLY	EARLY	PPLANT	NO EARLY	EARLY
4	**5.089**	6.047	4	**15.59**	19.79
8	8.424	**6.338**	8	**22.44**	27.77
12	9.304	**7.405**	12	**29.28**	38.02
16	18.47	**7.463**	16	51.38	**43.21**
20	22.26	**8.185**	20	59.04	**47.17**
24	31.32	**9.624**	24	85.38	**64.53**
28	37.08	**10.58**	28	100.4	**71.98**
32	48.81	**13.27**	32	145.4	**103.0**

Listing A.1: Insertion with two moves

```
// array[1] to array[SIZE-1] are sorted
array[0] = newElement;
for(int i = 0; i < SIZE - 1; ++i){
   if(newElement > array[i+1]) {
      // swap
      array[i] = array[i + 1];
      array[i + 1] = newElement;
   }
}
```

Table A.3: Bubble sort with and without early out (see Listing 4.10). Rendering times in ms on the NVIDIA GTX 1080. Note, that the first two times are almost identical and the version with early out is better in general.

SAMPLES:	NO EARLY OUT	EARLY OUT
4	**0.0527**	0.0528
8	**0.0983**	0.0984
12	0.1502	**0.1482**
16	0.2396	**0.2391**
20	0.3127	**0.2961**
24	0.3836	**0.3420**
28	0.4456	**0.4211**
32	1.1093	**0.5507**

Table A.4: Rendering times in ms for MLAB with and without stencil test (NVIDIA GTX 1080 Village Scene).

SAMPLES:	NO STENCIL	STENCIL
FULLSCREEN PASS:		
4	0.0543	**0.0410**
8	0.1034	**0.0727**
16	0.1956	**0.1352**
32	0.3901	**0.2273**
OVERALL:		
4	**1.2483**	1.2780
8	**1.6906**	1.7285
16	**2.4279**	2.4740
32	4.3837	**3.9752**

Table A.5: Rendering times in ms for UMLAB with and without stencil test (NVIDIA GTX 1080 Village Scene).

SAMPLES:	NO STENCIL	STENCIL
4	**1.1551**	1.1970
8	1.3855	**1.3783**
16	1.6916	**1.6404**
32	2.8426	**2.6829**

A.1 HAT ERROR WITH TWO UNDERESTIMATIONS

This section describes how the error for two succeeding underestimations at one point (z_0) is calculated by HAT and compares the calculated error with the actual error. This example extends Figure 6.3 by one fragment ((z_2, α_2)) with $z_2 > z_1 > z_0$.

E_1 describes the visible error after one underestimation at z_0. The exact same value will be computed by HAT as well.

E_2 describes the visible error after the second underestimation at z_0.

E_{computed} describes the error that is computed by HAT for the second underestimation.

$\text{vis}_{c1}(z)$ and $\text{vis}_{c2}(z)$ describe the visibility function after the first and the second underestimation respectively.

$$\begin{aligned}
\overline{\alpha_i} &:= 1 - \alpha_i \\
E_1 &= \alpha_1(\text{vis}_t(z_1) - \text{vis}_{c1}(z_1)) \\
&= h\alpha_1(\overline{\alpha_0} - \overline{\alpha_0}\,\overline{\alpha_1}) \\
E_2 &= \alpha_1(\text{vis}_t(z_1) - \text{vis}_{c2}(z_1)) + \alpha_2(\text{vis}_t(z_2) - \text{vis}_{c2}(z_2)) \\
&= h\alpha_1(\overline{\alpha_0} - \overline{\alpha_0}\,\overline{\alpha_1}\,\overline{\alpha_2}) + h\alpha_2(\overline{\alpha_0}\,\overline{\alpha_1} - \overline{\alpha_0}\,\overline{\alpha_1}\,\overline{\alpha_2}) \\
E_{\text{computed}} &= \left(1 - \frac{\text{vis}_{c1}(z_2)}{\text{vis}_{c1}(z_0)}\right)(\text{vis}_{c1}(z_0) - \text{vis}_{c1}(z_2)) \\
&= \left(1 - \frac{h\overline{\alpha_0}\,\overline{\alpha_1}\,\overline{\alpha_2}}{h\overline{\alpha_0}\,\overline{\alpha_1}}\right) h(\overline{\alpha_0}\,\overline{\alpha_1} - \overline{\alpha_0}\,\overline{\alpha_1}\,\overline{\alpha_2}) \\
&= h\alpha_2(\overline{\alpha_0}\,\overline{\alpha_1} - \overline{\alpha_0}\,\overline{\alpha_1}\,\overline{\alpha_2}) \\
&\Rightarrow E_{\text{computed}} < E_2
\end{aligned}$$

Unfortunately, the error computed by HAT is smaller than the actual error. Thus, the visibility function is probably compressed more often at z_0 than it should be. Therefore, it is likely that the visible error is not kept minimal after multiple compressions.