Transparent geometry rendering in deferred pipelines for real-time applications

Tomas Savickas
Supervisor: Prof. Stephen Pettifer

Department of Computer Science, University of Manchester
April, 2022

Abstract

Transparent objects are all around us in any relatively modern environment. They allow for interesting and pleasing designs and architectures, as well as simple matters of convenience, like seeing the contents of a box without opening one or providing invisible shelter from the rain. Even if transparent objects go consciously unnoticed in our everyday lives, they are still a large part of what makes up our reality. These objects are important if a computer-generated images are to be made more believable, which is one of the main goals of graphics programmers all around the globe.

Since the most visually pleasing visuals, among real-time applications, are produced in games, they are what we will be directing our focus at in this paper.

It has been said in recent years, that games have "solved" lighting. This can be in part attributed to more powerful hardware and in part to the rise of deferred rendering pipelines. In fact, a magnitude of games that heavily utilise inherently dark environments with a plethora of lights for the sake of a pleasant visual experience can be seen. However, even to this day very few games utilise intricate, multi-layered transparent geometries or transparency effects due to their cost. The games today are still very much stuck with using opaque bottles, simple dirty textures for windows or simply making the windows opaque altogether. We explore the reasons behind high-cost transparency effects and try to tackle the problem ourselves by exploring already existing algorithms and categorising them by usage. Finally, we present a novel "cluster collapse" method for combining these different methods so that only the most efficient method can be chosen for each situation instead of relying on a generic transparency method.

Acknowledgments

First and foremost, I would like to thank my family for supporting me when the tasks seemed insurmountable. Thank you for reminding me that work takes time, patience and perseverance more than anything else.

Next, I would like to thank all my friends for listening to my never-ending rants about how graphics programming is both phenomenally enjoyable and absolutely ghastly at the same time.

Finally, I would like to thank professor Stephen Pettifer for guidance throughout the project, as well as wonderful questions that lead to new insights.

Contents

List of Figures

Chapter 1

Introduction

1.1 Objectives

The project aims to explore various strategies for rendering transparent geometry in deferred pipelines in real-time and arrive at a conclusion of when each one should be used. This includes implementing and evaluating visual fidelity, memory and processing requirements, and limitations of the following algorithms:

- forward transparency;
- depth peeling;
- dual depth peeling;
- depth-weighted blended transparency;
- A-buffer transparency via per-pixel linked-lists (PPLL).

Furthermore, a novel "cluster collapse" method for mixing any of the aforementioned methods with the A-buffer transparency method is proposed.

1.2 Structure

This report is split into 8 parts. Firstly, this very introduction, followed by the software that was developed to enable the implementation of various graphical algorithms along with some abstractions that are crucial for any rendering software are covered. Next, the basics of the usual rendering pipelines are described, and why this project focuses on the deferred pipeline. Then, types of transparency are covered and the main issue of transparency in deferred pipelines is introduced. Afterward, the main topic – different transparency methods, their combinations in

deferred pipelines followed by the description of cluster collapse. Next, discussion about the performance and visual fidelity of the methods. Finally, the conclusion and potential future work are presented at the end.

Chapter 2

Renderer

To facilitate the implementation of various rendering algorithms a rendering software package was developed. This software is essentially a package containing a thin abstraction layer on top of OpenGL, as well as various visualisation, performance metric, and control tools. This was absolutely necessary, as managing the graphics processing unit (GPU) state via relatively low-level OpenGL function calls is not difficult but fairly error-prone and hard to debug. Furthermore, various methods covered in this paper require wildly different GPU state setups, having additional abstraction layers to simplify these states made the implementation of different methods much easier.

The whole software architecture will not be covered in thorough detail as it is a fairly large program at around 9 thousand lines of C++ code. Instead, only a very high-level overview of the key pipeline abstraction and its components will be given, to aid the reader in understanding the implementation details of the algorithms presented further on in this paper.

The test scene is also described to give some context to the images presented throughout this paper.

2.1 Shaders

Shaders are simply programs that run on the GPU. There are various flavours of shaders, of which the most important for us are:

- vertex shader – runs for each vertex of the model and applies any desired transformation to them;
- fragment shader – calculates a colour for each fragment (read – pixel) and outputs it to an output attachment;

- compute shader – a program that does not necessarily have a graphical output, often used for running highly parallelised code on GPU for speed reasons.

Historically, the first routines that did graphical shading were called shaders, they retained the name when the calculations were moved onto specialised GPUs. By the time the GPUs became general enough to run code that does not do graphical shading, the name was already solidified.

2.2 Attachments

Any data that can be passed into a shader or generated by a shader is an attachment: texture, cubemap, plain old data structure, array (of known size) of structures, etc. Most often attachments happen to be simple 2D textures.

2.3 Pass

2.3.1 Subpass

A subpass is a simple descriptor of: what objects need to be rendered (opaque, transparent, GUI, particles, debug geometry, etc.), what attachments the subpass will require for rendering, and finally, what shaders should be used. Subpasses are often referred to as passes.

2.3.2 Renderpass

A single renderpass can be thought of as a plain container of attachments and subpasses. Usually, the attachments belonging to a renderpass will only be used by the subpasses of the same renderpass. The only exception is an "output attachment" which can be used by other renderpasses.

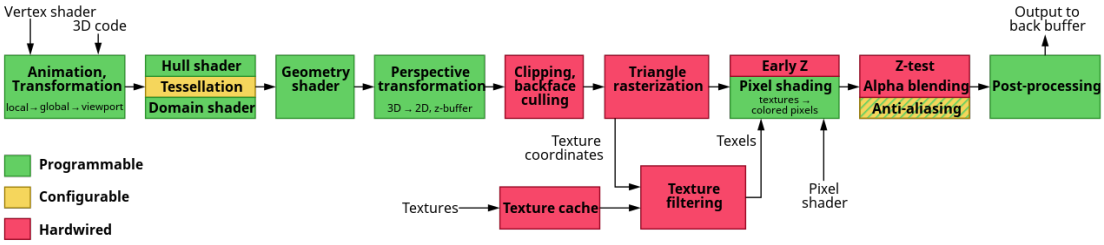
Renderpass is a useful grouping abstraction – different subpasses that achieve some common goal or output are grouped under a single renderpass. The goals can be generating either a visual output or some intermediate output used by other renderpasses: shadow map generation, light culling, final image generation, etc.

2.4 Pipeline

Pipeline is essentially a description of what steps need to be taken to render some data onto the screen. Most often this data is a combination of 3D models, textures and material properties. The pipeline can be described at various levels of

abstraction, and often one can find a conceptual model at the hardware level as seen in figure 2.1.

Figure 2.1: Full graphics pipeline.



However, a rather large chunk of this pipeline is not configurable by the programmer. Furthermore, many effects in graphics applications require multiple passes through this hardware pipeline. Hence, in the context of our renderer, a pipeline is the set of all the passes through the programmable bit in the hardware pipeline the program has to take to render the final image. In other words - the set of all renderpasses, that when executed in order will produce the final output of the renderer.

2.5 Test scene

For the test scene, the famous Dubrovnik Sponza palace is used, along with a few Stanford dragons (rather high quality model at 871,414 triangles), some simple spheres, two particle systems and 2^{14} point lights moving in an ellipse around the scene. Figure 2.2 shows a top-down plan of the scene. Different colours and shapes represent different objects:

- black rectangles – outlines of the Sponza palace;
- black ellipse – opaque Stanford dragon;
- hollow magenta ellipse – path the point lights move in;
- yellow square – smoke particle system that is lit by the lights;
- grey square – smoke particle system that is unlit by the lights;
- green, blue, red ellipses – transparent Stanford dragons;
- hollow green, blue, red circles – transparent hollow (important with the refractive effect) spheres;

- green, blue, red circles – transparent spheres.

Figure 2.2: Diagrammatic top-down view of the test scene.

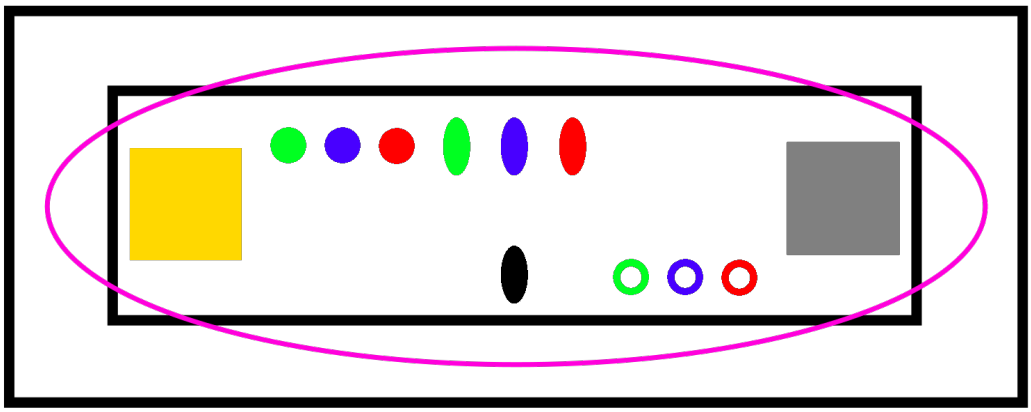
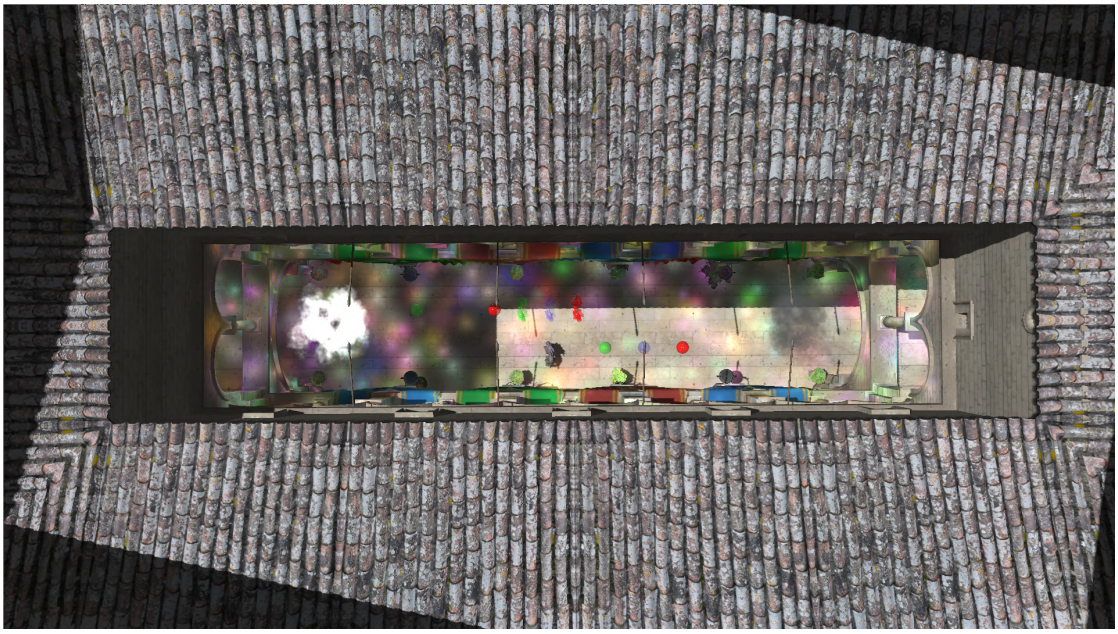


Figure 2.3: Same top-down view in the renderer.



Chapter 3

Graphical pipelines

This chapter introduces the two most common ways of setting up a render pipeline, how they were derived and some problems that need to be solved along the way.

3.1 Forward rendering

Forward rendering pipeline is the simplest method to render objects on screen. It has been used since the inception of computer graphics and is still widely used today in many games and other real-time graphical applications.

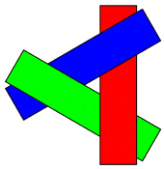
The idea is rather straightforward - simply render all of the objects on screen one by one. If there are any shaded pixels under the object currently being rendering, they get overwritten with the shaded pixels of a new object. After all the objects are rendered, one will be left with the final image. However, immediately an obvious problem arises: what happens if one renders an object close to the camera, and then an object that is further from the camera on top of the former object? With such an algorithm, a physically impossible phenomenon where a further object occludes a closer object, will be rendered. Fortunately, there are methods to remedy that.

3.1.1 Painter's algorithm

The most popular early method to fix the aforementioned problem was the painter's algorithm (sometimes called depth-sort algorithm). The idea is again, fairly obvious – before rendering all of the objects, they are simply sorted far-to-near with regards to distance from the camera. Generally, this solves the problem of more distant objects being occluded by closer objects, however, it does so in a per-object fashion.

There is no way to sort the objects in the case shown in figure 3.1. No matter

Figure 3.1: Fail case for painter's algorithm.



how objects are sorted, there will always be at least one rectangle that will have both of its ends occluded if rendered with painter's algorithm. Since at least one rectangle has to appear "last" in the sorted list; therefore, will be occluded by the other two rectangles.

One might try to apply this idea at a lower level – by sorting the primitives (triangles). However, depending on how large the primitives in the model are, similar impossible sorting issues will arise. Hence, this is not a bulletproof solution and would only work in some very specific cases.

3.1.2 Depth (Z) buffer

Fortunately for graphics programmers, the correct visual output can be achieved with a different strategy - leveraging the depth (or often called Z, as introduced by Catmull (1974)) buffer. A new, off-screen (one that will not be displayed to the user) texture is created, where at each pixel the distance to the camera (called depth), of the object that ends up rendered on said pixel, is stored. Now, the objects are rendered, but each pixel is only rendered on-screen if the depth of this new object is less than what already is in the Z-buffer. If this condition is satisfied, the depth in the Z-buffer is updated with the new – smaller depth. This essentially does object sorting at the pixel level and guarantees that each object is correctly occluded by any other objects that lie closer to the camera.

3.1.3 Shading and lights

Lighting objects in forward rendering pipelines is fairly straightforward as well. For each pixel of the rendered object we have to iterate through all the existing lights in the scene and calculate their effect on the said pixel. Conceptually, the cost of shading a scene with such a model has the complexity $\mathcal{O}(objects * lights)$. This does not sound bad at first, however, it can go horribly wrong when considering scenes with many light sources, e.g. a house with Christmas decoration lights, a room with many candles, a forest with many fireflies, etc. In the environments usually seen in modern games, this approach is not feasible if real-time frame-rates are to be achieved.

3.2 Overdraw and Z pre-pass

The problem of correct occlusion has been solved. However, another issue becomes more apparent now. With Z-buffer solution some objects are rendered, and then some other objects, that are closer to the camera, are rendered. This means that whatever pixels were rendered previously are now thrown away since something was rendered on top of it, overwriting the old pixels. This unproductive overwriting of the old pixel values is called overdraw. Having overdraw means that precious GPU cycles are wasted rendering objects that do not need to be rendered since they will not be visible in the final image.

One can solve this issue by carrying a so-called Z pre-pass. It is a simple subpass that does not produce any visual output, but instead renders all objects into the Z-buffer and only then does shading in a subsequent subpass. At first glance this might seem silly and expensive, after all, the geometry is rendered twice! However, doing Z pre-pass allows us to avoid overdraw in the second, expensive, subpass by carrying out an early-depth test. It quickly checks if the pixel of the current object would be occluded by any other object via the Z-buffer and if so, does not carry out any of the expensive lighting calculations.

To put it differently, Z pre-pass is simply a pass that caches depth information until the shading passes, when this depth information can be exploited to reduce overdraw.

3.3 Deferred rendering

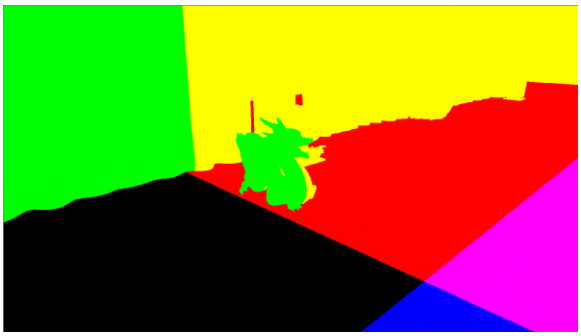
A natural progression is to exploit the idea of caching and reducing overdraw even further. Turns out, caching can be used to significantly reduce shading complexity.

3.3.1 G-buffer

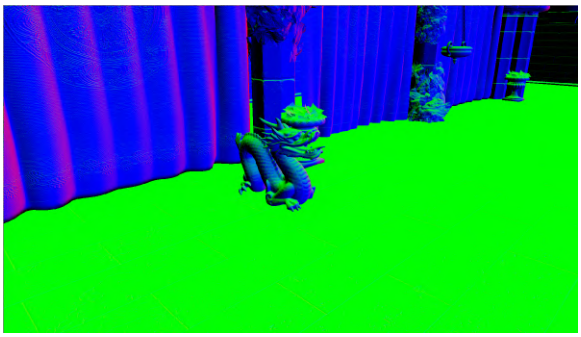
Deferred rendering caches much more data than a simple Z pre-pass. It generates a so-called geometry buffer (often G-buffer, as named by Saito and Takahashi (1990)) during a geometry pass, which contains multiple textures of the scene that describe various properties of the objects: position in the scene, normals, albedo colour, material properties, etc. There is no one "correct" way to form a G-buffer; it might be done differently depending on the specific needs of the program. Note that this buffer requires a lot of video memory (VRAM) since multiple viewport-sized textures have to be held in memory until the lighting pass. G-buffer further reduces the expensive overdraw since the geometry pass only renders properties of the objects are output to textures – no costly computations need to be done, so the overdraw that does occur is not very expensive.

Figure 3.2: Textures composing the G-buffer.

(a) Position in world space.



(b) Normals in world space.



(c) Albedo.

(d) Depth.

Unsurprisingly, this is the common runtime-memory trade-off. G-buffer saves us from overdraw, but requires a huge amount of video memory as each texture in the G-buffer is stored on the GPU. In the case of our renderer, only 4 textures are needed as seen in figure 3.2, whereas more sophisticated renderers routinely store much more textures for various properties, material IDs, etc. This VRAM usage is one of the reasons why deferred rendering only became popular around 2008 since earlier hardware was severely bottlenecked by VRAM size and speed. For reference, G-buffer in our renderer consumes $126.56MB$ of VRAM (at 1920×1080 resolution). Meanwhile, one of the most powerful GPUs at the time – GeForce 8800GT had $512MB$ of VRAM, meaning our G-buffer would consume nearly a quarter of all available memory! It must be noted, that there are ways to compress data in the G-buffer, however, these fall outside the scope of this paper. That being the case, the G-buffer in our renderer does not do any compression or smart data packing to save memory.

Finally, the G-buffer essentially decouples geometry processing from lighting calculations. This is crucial in lighting calculations as outlined next.

3.3.2 Lights

Lighting is where the most performance is to be gained. The same approach as with forward rendering can be used – simply sample the textures in G-buffer at each pixel and calculate the colour of said pixel by iterating through all lights. Some performance would be gained with this approach since overdraw was largely eliminated and so each pixel only needs lighting to be calculated once for it. Unfortunately, the major issue of multiplication in the complexity still remains.

There are various ways to do lighting better in deferred rendering. One of the earliest proposed methods is called deferred lighting (Deering et al., 1988). It simply renders the unlit objects using the information from the G-buffer, and then renders various invisible shapes, in a forward rendering manner – on top of what is already on-screen, for each light. The shape depends on the type of the light. Usually, there are three types of lights: point lights, spot lights and directional lights. Sphere, cone and full-screen quad shapes are used for each type, respectively. When the invisible shapes are rendered, the pixels that fall under the volume of said shapes get lit by the light. Again, all the calculations are supported by geometry information from the G-buffer. Finally, since many lights can overlap, additive blending is enabled to sum up the influence of the lights. This approach essentially reduces rendering complexity to $\mathcal{O}(objects + lights)$. Replacing multiplication with a sum is an immense improvement that allows us to have tens of thousands of light sources instead of tens or hundreds.

3.3.3 Tiled light culling

More sophisticated methods of doing lighting include tiled deferred rendering (Olsson and Assarsson, 2011), clustered deferred rendering (Olsson et al., 2012) and many more. Our renderer uses tiled deferred rendering, mainly for its good performance as well as a rather straightforward implementation complexity when compared to other methods. It is an extension of the deferred rendering algorithm and uses the G-buffer to cache data about on-screen objects, but does the lighting in a slightly different way.

The method works by splitting the viewport into a grid of small tiles. A tile is a rectangular region of the viewport, in figure 3.4 c the tiles can be seen as regions with the same continuous heatmap colour. For each tile, a new fictional camera frustum is constructed. Each frustum extends from the center of the main camera and passes through the corners of the tile it belongs to. Then every light is tested against every tile's frustum to see whether it overlaps the tile. If the light overlaps a tile's frustum, it is saved in the tile's "light list". This process is called tiled light culling. Finally, the final image is rendered to the screen in a similar fashion mentioned in the forward rendering pipeline. However, now instead of having to

carry out calculations for all lights for a given pixel, only those in the light grid are used.

Data structure

Data, about which lights fall under which tiles, has to be created and maintained on the GPU. This could be done on the CPU, however, the GPU can do this work for us in a very efficient way using compute shaders, since culling is embarrassingly parallel. Furthermore, by creating and maintaining the data structure on the GPU a hefty cost of sending data from CPU to GPU every frame is avoided.

Creating data structures on the GPU requires a bit of creativity since memory cannot be allocated and deallocated freely as on the CPU. On the GPU only fixed-size buffers and textures are allowed.

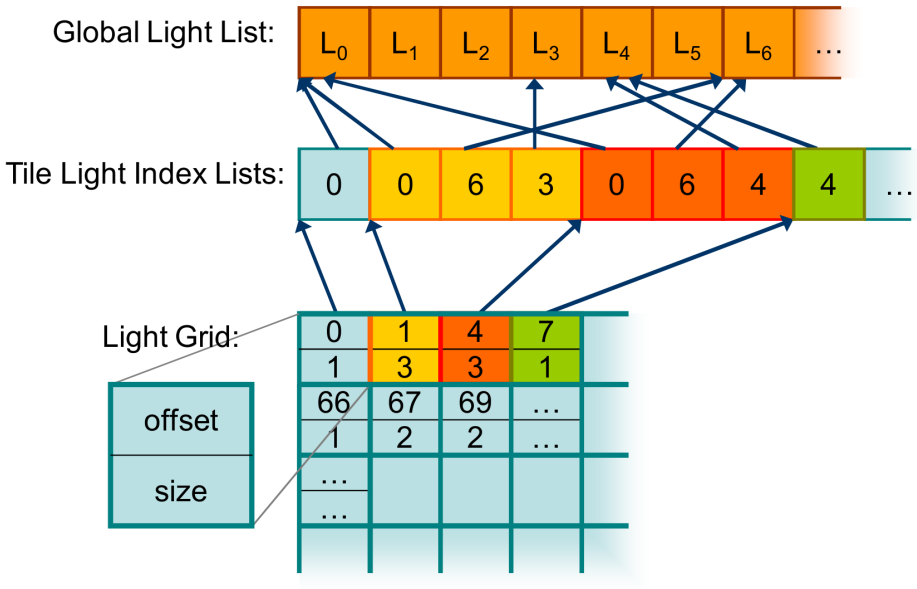
Firstly, regardless of what lighting methodology is used, a global light list that has been uploaded to the GPU from the scene description on the CPU. This list contains structs that describe the lights: positions, ranges, types of light source, colours, intensities, etc. This has to be uploaded from the CPU since the description of the scene is deserialised on the CPU. If the lights are not moving this light list will not be updated for the rest of the program's lifetime.

Specifically for tiled light culling, a 2D array called a "light grid" is needed. It represents on-screen tiles, as well as a tile light index array. Tile light index array stores indices to the lights in the global light list. The light grid stores two values for each tile: size and offset. These two values describe a sub-array in the tile light index array, which contains all the indices of the lights belonging to the said tile. The situation can be summarised more briefly and accurately with a diagram seen in figure 3.3.

Min-max depth optimisation

In its simplest form, tile frustums consist only of top, bottom, left and right planes. However, that means each frustum is essentially infinitely deep. In many cases, this infinite depth amplitude will fail to cull many obviously irrelevant lights. For example, if the camera is looking at a wall, the culling algorithm will not cull any lights that are behind the wall, since the depth information of the scene is ignored. The number of lights culled can be improved when near and far planes to each frustum are introduced. One does so by finding the minimum and maximum depth values of all pixels that fall under the tile at hand. Then the near and far planes are set to min and max depths respectively. Now, if faced with the same example as before — camera facing a wall with many lights behind it — the lights behind the wall will be culled away. Couple more examples can be seen in figure 3.4: in the top-down view the roof blocks most of the lights; in the other case the

Figure 3.3: Tiled light data structure.



columns block many lights.

However, this optimisation uncovers another problem. The tiles that have large min-max depth amplitudes still have to consider many lights. The most obvious case of this are the red tiles on the edge of the column on the right. The tiles include both the column and a wall very far back. This is of course necessary to ensure that lights illuminating the nearby column, as well as lights illuminating the faraway wall are considered, for the correct visual output to be ensured. Unsurprisingly, there are other algorithms that tackle this large depth amplitude issue, for example, the aforementioned clustered deferred rendering (Olsson et al., 2012). However, these techniques are much more complicated and falls outside the scope of this paper and will not be explored here.

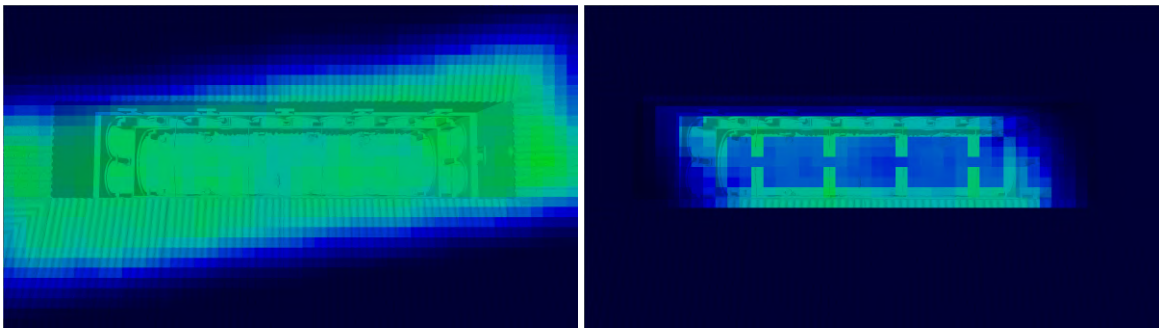
3.3.4 Overlapping lights

It is important to notice, that none of the techniques mentioned will perform well if many lights influence a single pixel. In such cases, regardless of the method used the renderer will struggle, since to produce a correct output all the lights will *have* to be considered for a pixel. This essentially brings us back to $\mathcal{O}(objects * lights)$. Both deferred lighting and tiled lights assume that only a few lights illuminate each pixel and attempt to avoid processing the lights that do not.

Figure 3.4: Light heatmaps with and without min-max depth optimisation.

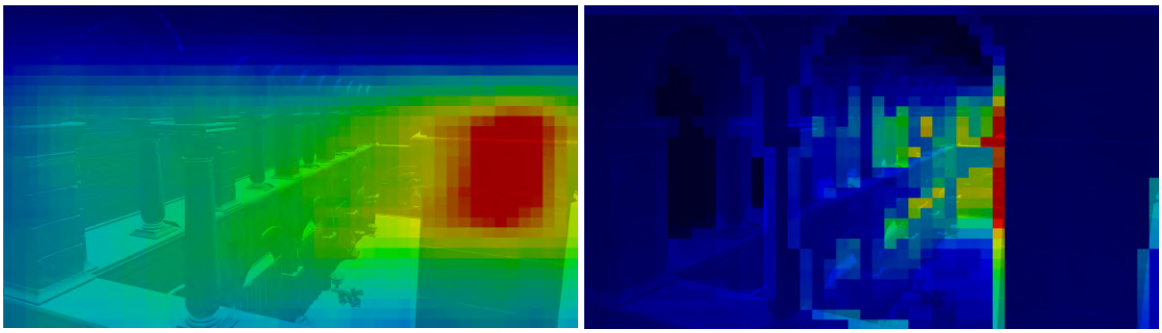
(a) Top-down view of the scene

(b) Top-down view of the scene, with min-max depth optimisation



(c) Random location in the scene

(d) Random location in the scene, with min-max depth optimisation



3.3.5 Renderers used nowadays

Deferred rendering pipelines dominate when it comes to modern games due to their ability to support tens of thousands of light sources in real-time. This allows game developers to create fairly realistic environments with pleasant lighting setups with many light sources. For example, the two most popular publicly available game engines Unreal Engine 5 and Unity both use a deferred rendering pipeline by default. For this reason, only deferred rendering pipelines will be considered further in this paper.

Chapter 4

Transparency

4.1 Layers

When talking about transparency, it is important to recognise that the more intricate transparent objects have layers. For example, a wine glass has multiple layers. Depending on the exact definition one could say that the first layer is the glass surface closest to the viewpoint, and the second layer is the surface that is seen through the first layer and is farther from the viewpoint than the first. An alternative definition would define four layers: one for the closest surface, another for the back-side of the closest surface and then two more for the surface farther away.

Layer concept is important as it allows us to reason easier about transparent objects and it appears multiple times in the various algorithms.

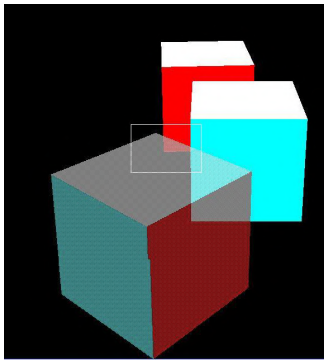
4.2 Types of transparency

It is worth separating the general "transparency" into a few different types, due to their inherent properties which influence how the objects have to be rendered.

4.2.1 Screen-door transparency

This type of transparency is currently most often used in games, when an object gets rendered in between the player model and the camera in such a way, that the player model would end up occluded. We do not deal with this type of transparency in this paper, but it is without a doubt the easiest one to implement. The object would simply be rendered with a conceptual stipple pattern overlaid on top, which informs which pixels need to be discarded so that the object behind can be seen. An example of how such an effect might look can be seen in figure 4.1, nevertheless,

Figure 4.1: Screen-door transparency. The closest cube is stippled.



there are multitudes of different ways of achieving transparency by discarding pixels.

4.2.2 Particles

Billboarded particles

These are the most simple particles that use elementary rectangular shapes that are always oriented to face the camera. The example in figure 4.2 showcases how such particles might look in our renderer. Each rectangle is rendered with a smoke texture where the rectangle is made transparent depending on the transparency level in the texture.

Other kinds of particles

Particle systems are not limited to billboarded particles. Objects of usual geometry can also make up particle systems. However, we are not exploring such particle systems separately, as they are simply bigger clusters of objects of the following type.

4.2.3 Transparent geometry

Simple transparency

Any object that has a well-defined geometry can be rendered as transparent. Simply transparent objects do not express any intricate transparency effects and simply have a colour or texture applied to them. For example, here are three monocoloured transparent spheres in our test scene figure 4.3.

Figure 4.2: Example billboarded particle system in our renderer.

(a) Particles in our renderer.

(b) Smoke particle texture used.

Volumetrics

This is a type of transparency that considers how light behaves inside some volume (geometry). Most often this would be something like "god-rays", realistic fluids, refractive materials, fog, etc. Usually, the process of rendering this kind of an object requires more information than simple transparency – usually data about each layer of the object is desired. E.g. if a refractive sphere is to be rendered correctly, information about the face that is facing the camera, as well as the information of the face at the back of the sphere (facing the same direction camera) has to be known. This information is required to be able to correctly calculate how the light rays travel through the sphere, not to mention that the level of transparency depends on the thickness of the object, as well as a plethora of other factors. Example in figure 4.4 shows refractive, tinted glass objects.

Figure 4.3: Simple transparency.

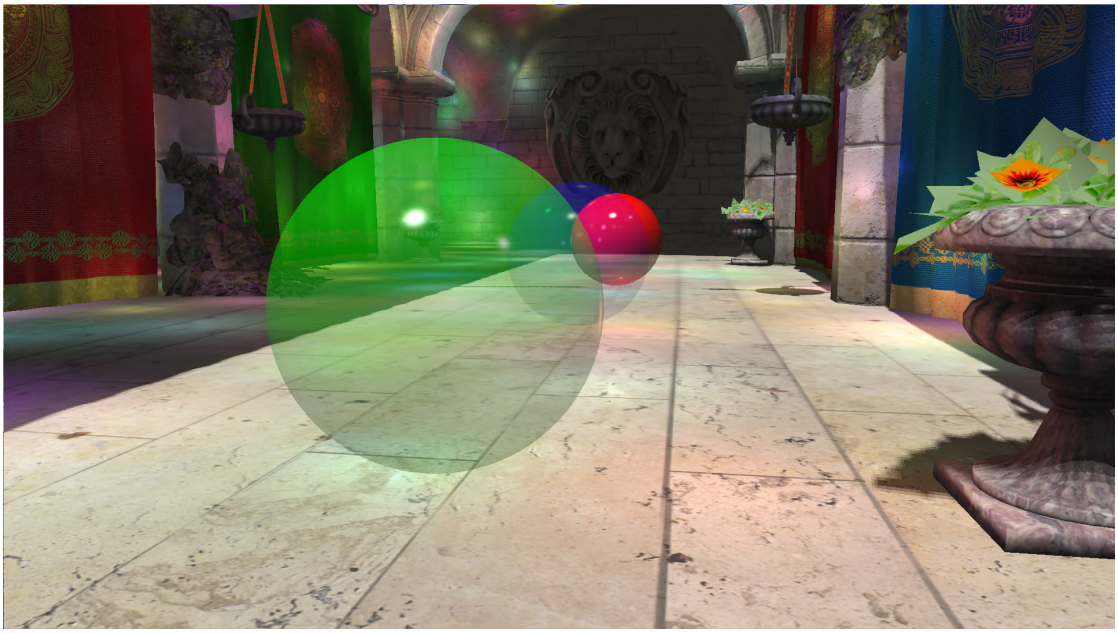


Figure 4.4: Refractive objects.

4.2.4 Compositing objects

Finally, let's assume that a way to render a transparent object is achieved. To achieve the correct final image, a way of overlapping these objects one over another is needed. This compositing process is called alpha blending and has been well established by Porter and Duff (1984). They proposed a computationally simple process known as the "over" operator, which takes a background image and composites a transparent image on top of it. Crucially, this process only allows laying a transparent image on top of another opaque or transparent object – in other words, it is not commutative. We have no method of *correctly* inserting a transparent image somewhere in the middle of other transparent objects. This requirement of strict ordering is one of the main reasons why transparency is difficult to render.

Once all transparent objects are rendered, they can simply be overlaid one over another using the "over" operator in a strict back-to-front fashion to achieve the final image.

4.3 G-buffer and transparency contradiction

There is an inherent contradiction between transparent objects and the G-buffer. For rendering transparent objects information about each transparency layer has to be known, so that they can be blended one "over" another. However, the cornerstone idea of G-buffer is the exact opposite of that – only store a single layer of data, since multiple layers cause overdraw. G-buffer only stores a single layer of information, when multiple layers are needed to be able to do transparency. In the next chapter various methods to solving this issue are explored.

Chapter 5

Transparent geometry rendering in deferred rendering pipeline

5.1 Proxy geometry for light culling

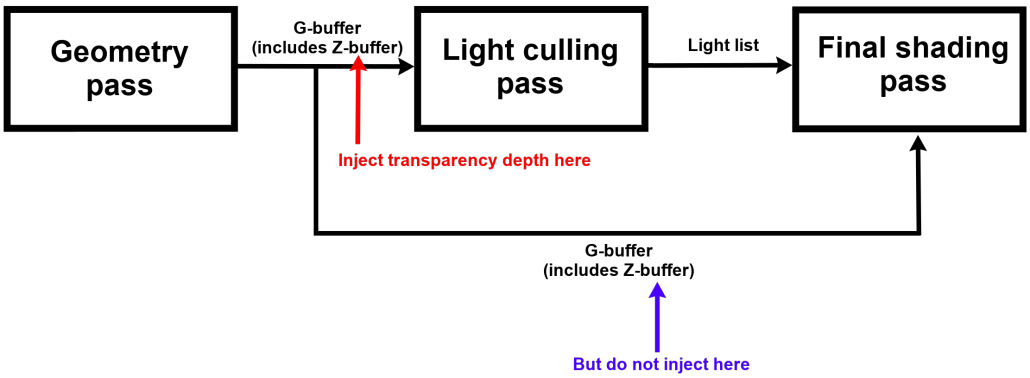
To achieve correct lighting effect, all the lights that might illuminate the objects need to be considered. However, for most of the upcoming algorithms writes to the Z-buffer are disabled. This is done to not discard the overlapping transparent objects. This poses an issue, because the tiled lighting culling with min-max depth optimisation uses Z-buffer to determine which lights should be considered for each tile. By disabling Z-buffer writes transparent geometry was essentially made invisible to the light culling algorithm. Hence, an incorrect light list for each tile occupied by a transparent object will be produced.

The light culling algorithm has to be made aware of the depth of transparent objects somehow. Howbeit, depth of transparent objects cannot be written to the usual Z-buffer. This is due to the fact that the depth buffer is generated during geometry pass and later used in the final shading stage, so writing to it would cause problems in the final image. This situation is summarised in figure 5.1.

The problem can be solved by introducing a completely separate Z-buffer only for depth of transparent objects. This secondary transparent object Z-buffer is used in addition to the usual Z-buffer in light culling pass to achieve the correct list of lights per each tile even when a transparent object occurs in a tile. It is important to note, that the objects rendered to this buffer are not visible in the final image, it is only rendered to the secondary Z-buffer. That being the case, one might not want to waste a lot of effort rendering the highest quality version of the object.

One might choose to use proxy objects that approximately represent original transparent objects in the secondary Z-buffer can be used. As is usual with graph-

Figure 5.1: Transparent geometry depth injection issue.



ics, a tradeoff is to be made when choosing what kind of proxy geometry to use. If bounding boxes are used as proxy objects, then there is risk that tiles that do not contain proxied object will think the object overlaps it. A middle-ground solution, between a bounding box and the object itself, is to use level-of-detail (LOD) models. LODs are incrementally simplified models of the original. By using LODs, the best render cost to tile occupancy accuracy ratio will be achieved. However, there is also an authoring cost to be paid here, as good LODs are often made by hand since automated processes do not guarantee a low-poly model that still retains the original idea and "feel" of the object. Due to this authoring cost, we opted to use bounding boxes in our renderer. For the sake of clarity of how these proxy objects might look, a visualisation of proxy geometry as cyan boxes is presented in figure 5.2.

Most of light culling algorithms require depth information. Thus, regardless of light culling algorithm, proxy geometry or some similar solution will have to be introduced in order for the lights to be culled correctly.

5.2 Forward transparency

The idea is identical to that of forward rendering, just adapted for transparency. First, the the background consisting only of the opaque objects is rendered using the deferred rendering pipeline. Then, all the transparent objects are rendered with alpha blending on top of that, as if switched to forward rendering. Important

Figure 5.2: Visualised proxy geometry (in cyan) for correct tiled light culling.

to note that Z-buffer writes are disabled during transparent object rendering so that none of the transparent objects are discarded. This obviously causes an issue, as alpha blending requires the objects to be rendered in a strict near-to-far or far-to-near order. Not satisfying this condition causes impossible visual output as seen in figure 5.3.

5.2.1 Painter's algorithm

The solution to the problem is again, identical to that previously seen with forward rendering. The painter's algorithm is simply employed – the transparent objects are sorted before rendering. As expected, this corrects the visual artifacts, now that alpha blending works correctly on sorted objects, as seen in figure 5.4.

However, one cannot forget that painter's algorithm does not do per-primitive sorting. One might expect that incorrectly sorted triangles in a single transparent object might blend together, especially if the whole object is monocoloured. Unfortunately, depending on the exact geometry the artifacts can still be very visible. A particularly visible case can be seen in the Stanford dragon model in figure 5.5, right down the middle of the model. One half of the object has its triangles ordered near-to-far and the other – far-to-near with a particularly visible transition between the two right in the middle.

Figure 5.3: Paradoxical view of further object occluding a closer one.

Figure 5.4: Forward transparency without and with sorting.

(a) Unsorted objects.

(b) Sorted objects.

Figure 5.5: Unsorted primitive issue.

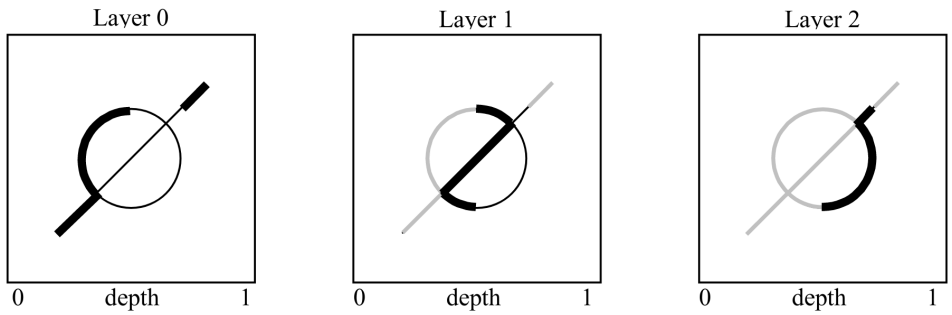
This approach might work in cases where the object is not self-overlapping (has a single layer) and the primitive sorting issue will not be apparent. It might also work with simple volumetric effects that do not have a very well defined geometry: god-rays or fog. This is because these effects are usually defined by very simple geometry (e.g. a box), where the effect is generated throughout the whole volume of the object, without actually rendering the object itself.

5.3 Depth peeling

Forward transparency has the issue of incorrect primitive blending. This problem can be tackled with the most straightforward solution – sorting all the primitives in an object. That means that one would have to be constantly sending the new, sorted, vertex data from the CPU to the GPU which is extremely costly, especially if the geometry is very complex. To avoid this, the GPU is employed to do the sorting. The following way of doing the sorting is called depth peeling and was first introduced by Everitt (2001).

It works by re-rendering the transparent object multiple times and each time only a single transparency layer of the object is kept. Every consecutive pass ignores the previous layer. Hence the depth peeling name – each layer of transparency is peeled away with the help of multiple depth buffers.

Figure 5.6: Depth peeling process over multiple passes. The conceptual camera is on the left, looking right.



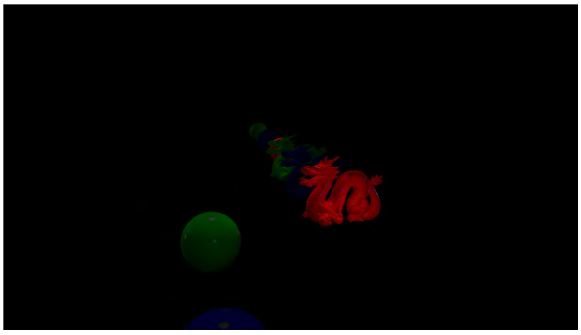
To achieve this depth peeling feature, auxiliary depth buffers are needed, let's call them "depthA" and "depthB". These two buffers are used in a "ping-pong" fashion. During the first pass the transparent geometry is rendered using depthA as the usual depth buffer, let's call this "working depth buffer". Meanwhile depthB will be assigned as the "minimal depth buffer". For every consecutive pass, depthA and depthB are swapped in terms of which is used as a working depth buffer and which as a minimal depth buffer. Furthermore, before every pass the working depth buffer is cleared, but minimal depth buffer retains data from the previous pass.

During the first pass, depthA is used as the working depth buffer, depthB is irrelevant at this stage since it is empty. After this pass, depthA will contain the depth information of the first layer of the transparent object. Throughout the second pass depthA will be used as minimal depth buffer, its purpose is to discard any pixels that have lesser depth – are closer or at the same distance to the camera than the first layer. This essentially forces the second pass to only render the second layer of the transparent object.

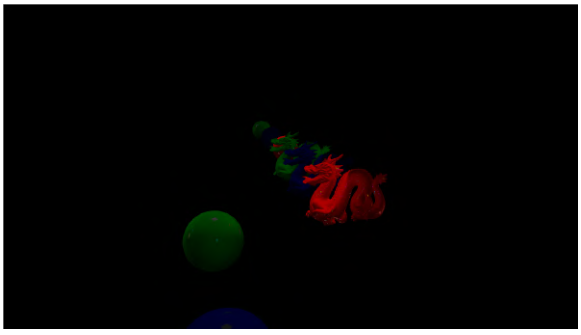
We continue doing as many of such passes as the time allows us to. Figure 5.6 from Everitt (2001) summarises this process very well in a conceptual diagram. The bold black line represents the layer rendered at each pass, everything to the left of this line is discarded due to the "minimal depth buffer" and everything to the right is to be rendered in future passes. Figure 5.7 showcases real outputs generated by the different depth peeling passes in our renderer.

Figure 5.7: Depth peeling outputs with artificially increased saturation for visibility purposes.

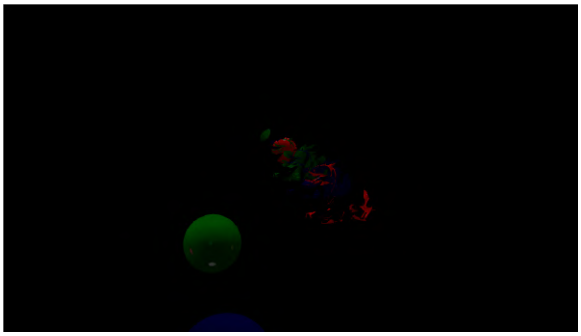
(a) 1st pass output.



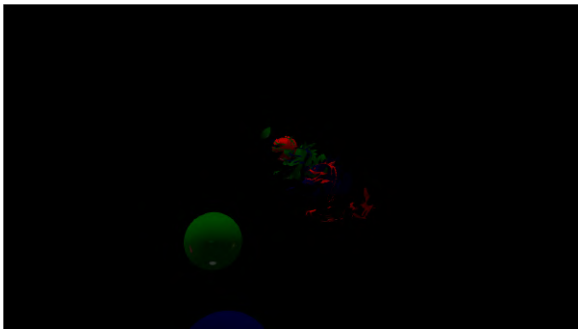
(b) 2nd pass output.



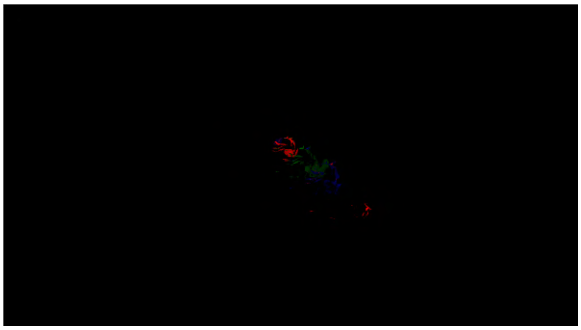
(c) 3rd pass output.



(d) 4th pass output.



(e) 5th pass output.



(f) 6th pass output.

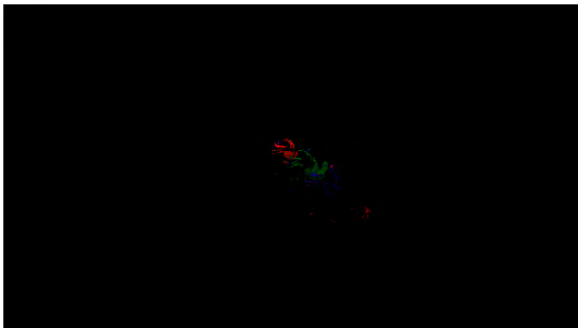


Figure 5.8: Visual artifacts of insufficient depth peeling layer count. Red sphere is invisible when looking through both green and blue spheres.

5.3.1 Compositing

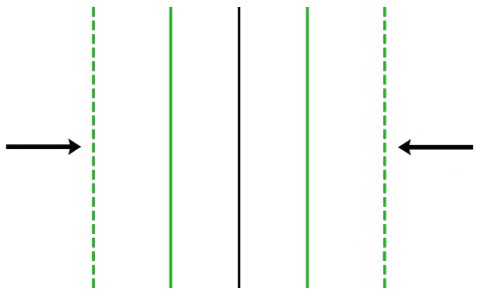
During each pass a new texture is created, where the shaded transparency colour is deposited. After all the depth peeling passes are done, these textures can be rendered in a forward transparency manner – each layer rendered on top of the opaque objects in a strict order, with alpha blending turned on.

5.3.2 Multiple pass implications

Depth peeling has a glaring downside – to achieve a correct final image, the method might have to perform an unbounded number of passes. This can be extremely expensive as an output texture for each pass is kept. Moreover, each pass re-renders the whole object which is very wasteful and very expensive. Especially, if the geometry is complex, like in the case of Stanford dragon used in the test scene.

Some of these issues can be mitigated by limiting the number of maximum passes. As more and more transparent objects are layered atop one another, each subsequent layer contributes less and less to the final image. Hence, the technique can be limited to a small number of passes, for example 6 or so. However, one must be aware that doing so will produce undesired artifacts, especially with low-transparency objects as seen in figure 5.8. This maximum layer count has to be

Figure 5.9: Advancing fronts of dual depth peeling.



carefully selected, depending on the specific scene that is expected to be rendered, in order to avoid aforementioned artifacts.

Regardless of the pass limiting, this method still suffers when complex geometry is rendered due to repetitive re-renders of objects. It is also not suitable for particle systems as it would require many passes to capture all the layers in the system. However, it might be a very good alternative for rather simple transparent objects that will have unsorted fragment issues when rendered with forward transparency.

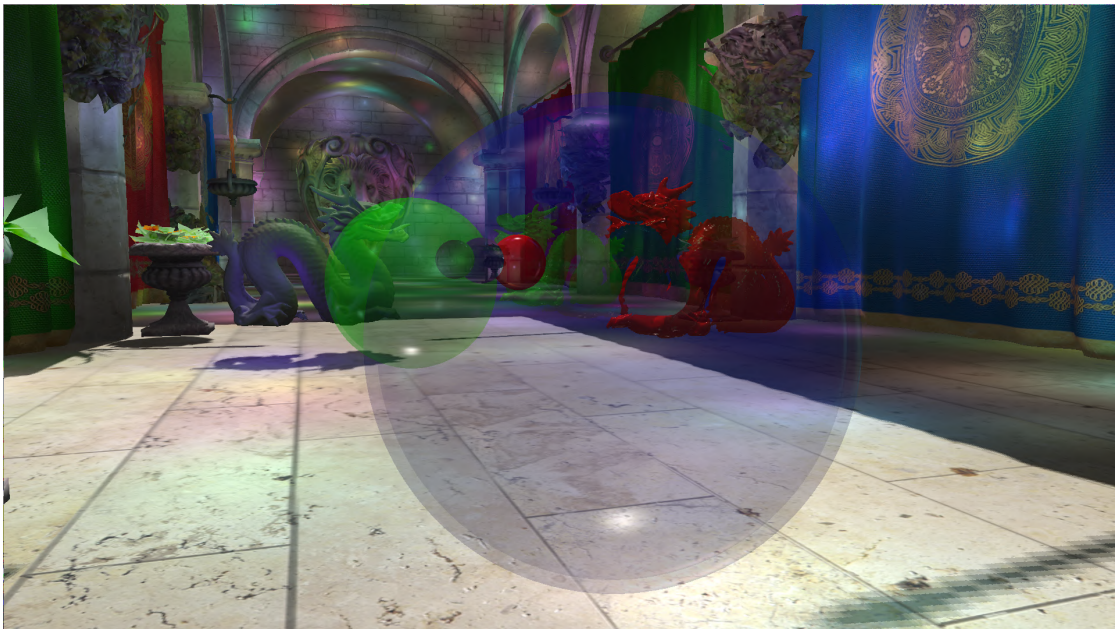
5.4 Dual depth peeling

A few years later after the original depth peeling method was published, an optimised version was proposed by Bavoil and Myers (2008), called dual depth peeling. As can be expected from the title, it piggy-backs off the original depth peeling method. The creators were aiming to reduce the number of times the geometry had to be re-rendered and came up with a method where instead of a single peel per pass, two layers are peeled. This approach achieves the same visual output with fewer passes and so requires fewer re-renderings of the geometry. In fact, it requires exactly half as many passes as depth peeling. Hence, achieving higher performance.

The method works by advancing two fronts – one from the front and another from the back. On the first pass only the depths of front-most and back-most layers are stored. On consecutive passes, the layers indicated by the depths in the depth buffer are peeled (dashed green line in figure 5.9) and the depths for next layer (solid green line in figure 5.9) are saved. There is a small issue with the layer right in the middle being processed twice (black line in figure 5.9). Fortunately, the remedy is very simple, just test if front and back layers have the same depth and if so, process it as a front layer.

As can be expected, this method suffers from the same unbounded pass count issue. And the solution here is the same – limiting the maximum pass count. Unsurprisingly, this produces a similar issue where information about layers is

Figure 5.10: Visual artifacts of insufficient dual depth peeling layer count.



missing. However, due to a different peeling strategy, the information about the layers in the middle of transparent objects are now missing, instead of at the back as was with the depth peeling method. The artifact arising from this issue can be seen in figure 5.10.

Depending on the limit of the number of layers, this method can use fewer output buffers than traditional depth peeling. It opts to blend the output into four separate textures – two "ping-pong" textures for the layers peeled from the front, another two "ping-pong" textures for layers peeled from the back. These two buffers are blended on-the-fly, however a final pass is necessary to blend the texture for the front layers with the one for the back layers.

Similarly to simple depth peeling, the method is still unsuitable for very complex geometries or particle systems. However, it might be a good fit for medium-complexity geometries.

5.5 Order-independent transparency (OIT)

This class of methods takes slightly different path than previous methods. It avoids sorting completely and tries approaching the problem from a different angle. There are a few families that this class can be further separated into, however, only a single family, that tries to redefine the compositing operator such that it

is commutative, will be investigated. This enables rendering the objects in any order. As will be seen later, these methods are just approximations, but they are good enough to look physically plausible.

The first to introduce a commuting blending operator called "weighted sum" was Meshkin (2007). This operator was achieved by taking the original "over" operator, expanding its mathematical formula, by including coefficients for many layers, and noticing that there are two separate parts: order-dependent and order-independent. Author then omitted the order dependent part of the formula to arrive at the weighted sum operator. Unfortunately, it had the downside of producing overly darkened/brightened images for objects with opacity levels higher than around 30%.

Only a year later, this was improved by Bavoil and Myers (2008) with the "weighted average" operator. Authors acknowledged that the order-independent part cannot be simply omitted, so they chose another approximation. They selected to take the average colour of all the layers and weigh them depending on the transparency level. This method produced a slightly more plausible result, yet it still had artifacts which looked as if the layers further from the camera have equal or more influence than the closer layers.

McGuire and Bavoil (2013) introduced weighted blended OIT method and immediately extended it to depth-weighted blended OIT which addressed this depth ignorance issue. It did so by using a depth-weight. In this section the focus will be directed to this latest method only, as it is a direct improvement on the previous ones and has the best visual result.

Finally, these methods are important to cover as they have a significant advantage over the other approaches since they only require a single pass. Furthermore, they require no work (such as sorting in the case of forward transparency) on the CPU. Lastly, very few additional textures or buffers need to be allocated, thus reducing VRAM load.

5.5.1 Depth-weighted blended OIT

Depth-weighted blended OIT's advantage over other OIT methods is that it additionally uses depth information. Weighted sum, weighted average and weighted blended OIT discarded depth information, therefore all pixels are treated as equal, regardless of the distance from the camera. This fairly obviously produces incorrect results when there is a less transparent object behind other transparent objects. Ending up with a weird result where further objects appeared as if they had too much influence, as seen in figure 5.11.

The exact depth weights highly depend on the scene. Authors provide multiple generic depth weight functions but note, that they themselves used a specifically

Figure 5.11: Comparison between weighted blended and depth-weighted blended OIT methods.

(a) Weighted blended. Far away spheres appear to have too much influence in the final colour.

(b) Depth-weighted blended. Correctly weighted colours.

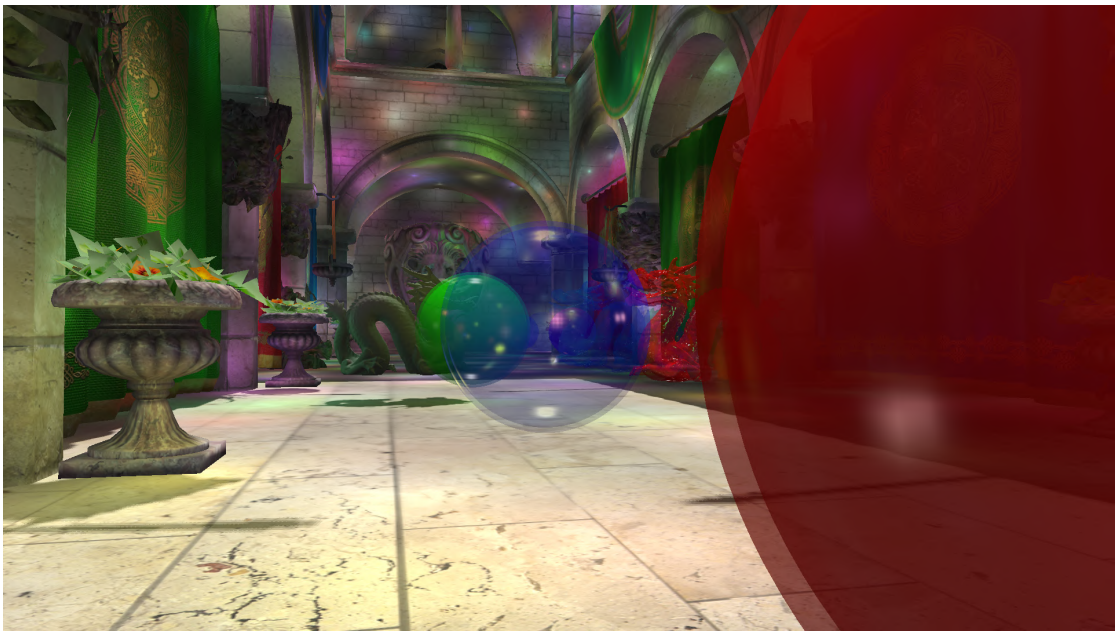
tweaked version for their scenes. We, however, opted to use one of the provided generic depth functions as they seemed to produce a fairly believable output.

However, this solution expects that either the layers have similar colours or are relatively evenly distributed throughout the depth range. Meaning, that if there are multiple clusters of close-by objects, that are far from one another, this method will produce unrealistic results. An example of incorrect output due to inaccurate depth weight can be seen in figure 5.12. That being the case, a conclusion can be drawn, that this method is particularly suitable for similar looking clustered objects, such as particle systems.

5.6 A-buffer transparency

A-buffer (accumulation buffer) idea was first introduced by Carpenter (1984) and was mainly intended as an anti-aliasing mechanism. In the original paper, A-buffer was essentially defined as a list of datapoints for each pixel, where each datapoint described a sub-pixel. Similar to multi-sampling nowadays, the sub-pixel values in the list were then averaged using their areas as weights to calculate the colour of the pixel. Back in the day, this was done on the CPU and it most certainly was not fit for any real-time applications. After the rise of GPUs, A-buffers were attempted, but it soon became apparent that it is too costly in terms of runtime and VRAM, not to mention the difficulty of implementing complicated data structures on such a constrained piece of hardware, when compared to a CPU. However, nowadays, very advanced GPU hardware that allows implementing various data structures rather easily, due buffers that allow arbitrary reads and writes. As well as the luxury of multi-gigabyte VRAM modules.

Figure 5.12: Depth-weighted blended OIT producing overly blue Stanford dragons behind the blue sphere.



A-buffer idea can be used in conjunction with the G-buffer idea. G-buffer tries to store the data needed to shade a single pixel, meanwhile, A-buffer allows us to store multiple pieces of data for a single pixel. An A-buffer backed G-buffer can be implemented for transparency data only, where each piece of per-pixel data will describe a transparency layer in our scene.

5.6.1 Per-pixel linked-list (PPLL)

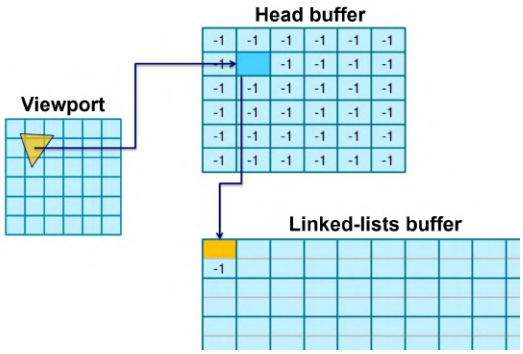
One of the possible ways to implement the A-buffer is to use a per-pixel linked-list. Multiple papers suggesting very similar implementations arose around 2010-2011: Crassin (2010); Barta et al. (2011); McKee (2011).

Each entry in this linked-list will represent a transparency layer for the pixel the list is assigned to. However, this data structure has to be kept on the GPU only. It would be far too expensive to generate this data on the CPU and upload it every frame to the GPU. To achieve this linked-list structure on the GPU a very similar approach to that used for lights can be employed.

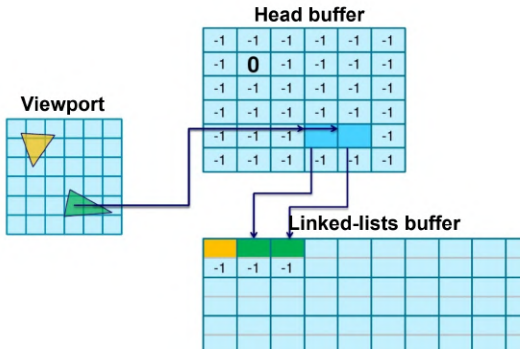
For this linked-list structure two buffers are needed: one texture of the size of the viewport called “head buffer” and a buffer that will contain actual linked-lists. As indicated by the name, the head buffer contains indices into the linked-

Figure 5.13: A-buffer implementation via a per-pixel linked-list.

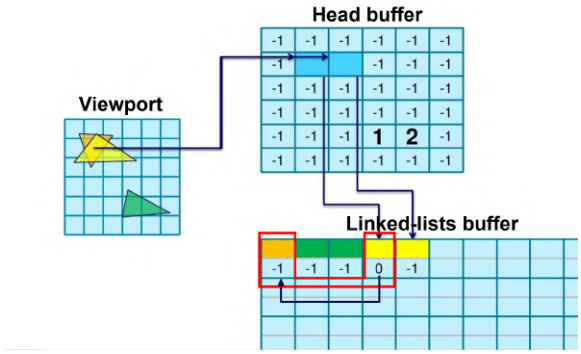
(a) Rendering first orange transparent triangle on one pixel.



(b) Rendering another green transparent triangle on two different, unrelated, pixels.



(c) Rendering a yellow triangle on top of the first, orange triangle, as well as one more pixel. A linked-list outlined in red is formed as a result of this.



lists buffer at each pixel position. The linked-lists buffer itself stores data about transparency layers, in a very similar fashion to how G-buffer stores data about a single layer of opaque objects, as well as an index to the next element of the linked-list of the pixel. In figure 5.13 we present how these buffers look, along with an example of drawing a few transparent triangles and how the data about those triangles is inserted into the relevant buffers.

We will be referring to the above described data structure as A-buffer for brevity henceforth.

The A-buffer is filled in when rendering the objects. Instead of drawing the objects on screen, similarly to G-buffer, all the important data is cached and used in the final shading later on. When a pixel is rendered, it is inserted somewhere

the linked-lists buffer and its index is stored in the head buffer. If before inserting the pixel there already was a linked-list present at the pixel location, this old pixel index is simply set as the "next" pointer in the pixel just inserted. Such a scenario can be seen in figure 5.13 c, as indicated by the blue arrow in the linked-list outlined in red.

5.6.2 Sorting

With this approach, the objects are not sorted before rendering them. That, in conjunction with the fact that all the data is stored on the GPU, means pixels need to be sorted on the GPU. Fortunately, compute shaders allow us to do that fairly simply.

For each pixel in the head buffer the whole linked-list of that pixel is retrieved, sorted and stored back to the linked-lists buffer. There are many different sorting algorithms that can be used here. In addition to the usual suspects: quick sort, merge sort, etc., there are specialised sorting algorithms that are designed to work better on a highly parallelised hardware like the GPU: bitonic merge sort, odd-even merge sort as presented by Pharr and Fernando (2005). However, it is rare to have hundreds or thousands of transparency layers. That being the case, simpler $\mathcal{O}(n^2)$ algorithms like insertion sort may be utilised. There will not be a big performance gap between different performing algorithms at very small array sizes. Nevertheless, the cost of sorting the linked-lists is not negligible, especially as the screen area of transparent objects increases. Unfortunately, buffers that allow both, reading from and writing to are very slow. This is why the cost of this step would remain very high, even when using a faster sorting algorithm, as the main overhead is reading and writing to the A-buffer.

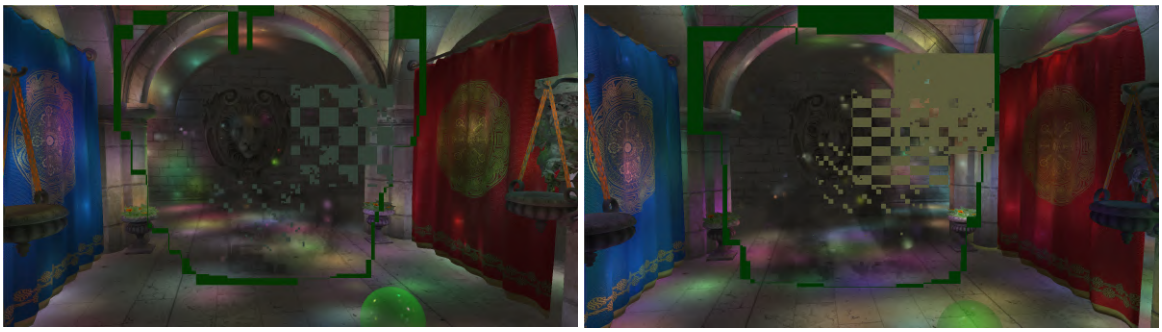
5.6.3 Layer limitation and self-balancing

This approach, similarly to depth peeling, can suffer from insufficient memory to store all the layers. Thus, it is not suitable for applications like particle systems, where hundreds of transparent particles might overlap. Figure 5.14 showcases what artifacts running out of memory might result in.

However, the data structure is in a sense self balancing layer-wise. Let's assume the renderer allocates enough memory for the linked-lists buffer to store 2 layers at each pixel. If the transparent objects only appear on a 4th of the viewport (so on a 4th of all pixels), then each of them can now have 8 layers, as the linked-lists buffer has enough space for that. This is a very desirable feature if rendering more intricate transparent objects like glasses and bottles are desired, as they rarely take up a lot of screen space.

Figure 5.14: A-buffer running out of memory due to too many transparency layers.

(a) (b)



5.6.4 No proxy geometry for light culling

Fortunately for us, A-buffer already has all of the depth data of the transparent objects. Therefore, there is no need to render any proxy geometry like it had to be done for all of the previous methods. Moreover, this depth buffer is made from the actual geometry and not the approximated proxy geometry, meaning the light culling will be more accurate. Hence, allowing the culling algorithm to cull more lights around the edges of the transparent objects, resulting in a performance boost.

5.6.5 Video memory implications

As mentioned previously, G-buffer stores quite a lot of data. Since essentially a G-buffer for transparent objects was implemented, it also consumes a *very* large amount of memory. Even more so than the opaque G-buffer since memory was allocated for multiple layers. This is the biggest downside of this approach. It must be mentioned, that it is possible to pack the data much more efficiently than we have. Moreover, data can be compressed or outright omitted and recalculated, e.g. world position can be calculated from on-screen coordinates and in-world depth. However, G-buffer optimisations fall outside the scope of this paper.

5.6.6 Volumetric effects

The biggest advantage of this approach though is that it stores all the relevant data about transparency layers. This allows us to implement a plethora of more intricate effects. For example, the thickness of the object can be calculated from the depth data, and then used to implement an effect approximating refraction without employing a more complex algorithm like ray tracing. As a proof of

Figure 5.15: Refraction approximation using transparent layer data in the A-buffer.

(a) (b)

concept, one such refraction approximation implementation can be seen in figure 5.15.

5.7 Mixing methods with A-buffer as a backbone

As noted in previous section, different algorithms are well suited for different types of transparency. Usually games and other real-time applications feature most of these different types of transparency. Meaning that some way of using the different methods for objects of different transparency types, as well as a method for compositing outputs together, is needed. As a proof of concept, depth-weighted blended OIT used for particles and A-buffer method used for complex Stanford dragon objects will be combined.

5.7.1 Additional depth buffer

The most naive way to combine these two methods would perhaps be to render the transparent objects with A-buffer transparency and then blend the depth-weighted blended particles on top. However, this completely ignores the depth-weights of the objects rendered with A-buffer method, resulting in the particles appearing as if they were always rendered on top of other transparent objects. This is remedied by altering the depth-weighted blended OIT method slightly. A new depth buffer is added to store the minimal depth of the particles rendered. When compositing, particle system depth can be compared with objects in the A-buffer and blended into a final image correctly. A comparison can be seen in figure 5.16.

Figure 5.16: Refraction approximation using transparent layer data in the PPLL.

(a) Naive blending.

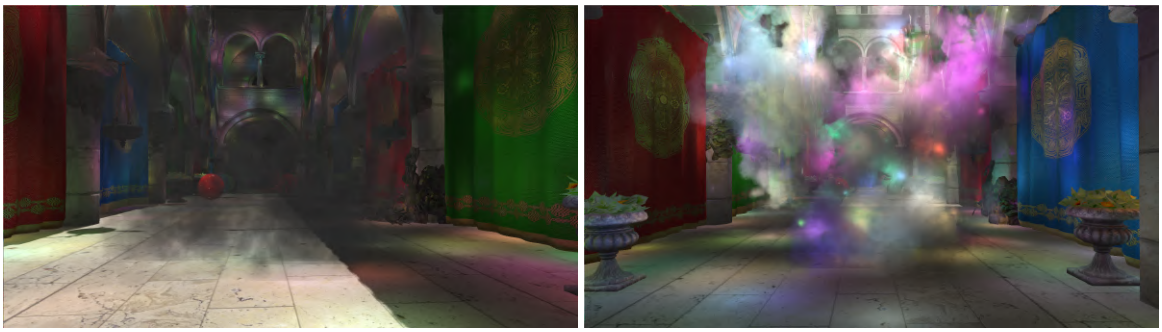
(b) Blending with additional depth buffer.

(c) Naive blending.

(d) Blending with additional depth buffer.

Figure 5.17: Single additional buffer is sufficient for blending.

(a) (b)



This has an obvious downside, if there are multiple particle systems with transparent objects rendered with A-buffer in-between artifacts will occur. As only a single depth buffer is used for all the particle systems, only the depth of the closest system will be stored. This will make it look like the particles further from the camera (and behind A-buffer rendered objects) are in front of the A-buffer objects. However, this is not always a significant issue, especially with particle systems that result in a low transparency value, as the A-buffer objects behind have little influence over the final image anyways. Examples of this can be seen in figure 5.17. The dragons are fairly strongly occluded by the nearby smoke, so it is hard to tell that the far-away particles are rendered on top of the dragons.

Unfortunately, incorrect blending can be seen, when using a single depth buffer for mixing the methods, if certain conditions are met. Most often, when the objects rendered with depth-weighted blended OIT of high transparency are in front of A-buffer rendered objects. An example of such a jarring artifact can be seen in figure 5.18 a. Furthermore, this mixing method suffers from pop-in when the camera is traversing a transparent object, which might often be the case with particle systems like smoke. This is due to the fact, that only a single additional depth buffer is used and there is no smooth transition – the particles are either behind or in front of the A-buffer objects without no in-between.

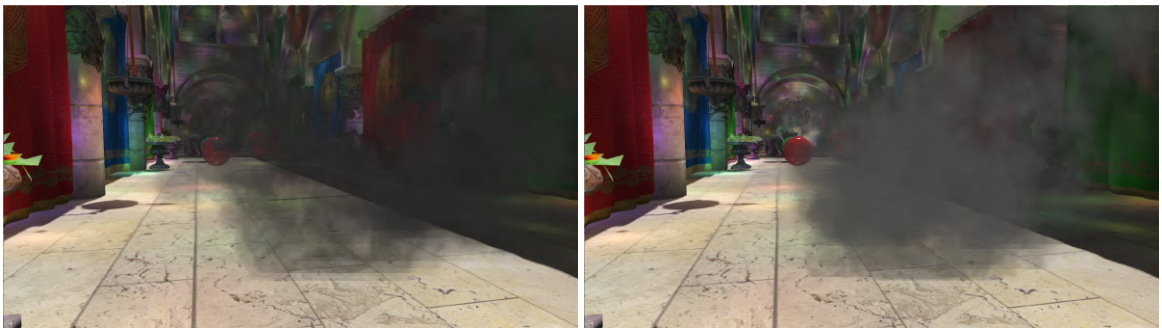
5.7.2 Cluster collapse

We propose a novel method that allows mixing other transparency methods with A-buffer transparency. The idea is rather simplistic – render clusters of nearby objects with any transparency method. Then add the cluster into the A-buffer as if it were a single transparent object. From then on, continue rendering using the A-buffer technique. The transparency layers simply get sorted on the GPU

Figure 5.18: Mixing via single depth buffer versus mixing via cluster collapse.

(a) Further smoke appears too significantly on the red sphere on the left when using a single depth buffer.

(b) Correct blending using our method.



and a correct image is achieved using the ordinary "over" operator. At first, the prerequisite of assigning objects to clusters might sound restrictive. However, consider the fact that A-buffer has the most difficulty with the use cases where objects are inherently clustered – particle systems. Usually, there should be no need to do any separate clustering on the CPU – all particles in a system can be assigned to a single cluster.

This mixing method has multiple upsides. Firstly, it eliminates artifacts from incorrect blending as seen in figure 5.18 b. Secondly, it completely removes pop-in as additional depth information about the particle system is stored in the A-buffer. Finally, the cluster collapse is not sensitive to what method is used for transparency. As long as the end result can be stored in the A-buffer, cluster collapse will work and produce a valid result. Meaning that a multitude of different transparency methods can now be employed, based on the specific objects being rendered, all the while without having to worry about how it all has to be composited together.

Of course, this is not without a cost. This method requires rendering a cluster and then activating a different shader to write the result into the A-buffer. Then it all needs to be repeated for the next cluster, and the one after until all the clusters are dumped into the A-buffer. This constant switching of shaders is costly. However, in the cases where there are only a handful of clusters visible at once, this is rather negligible, especially considering the flexibility offered by the method.

Chapter 6

Discussion

We present performance metrics in terms of frametimes and memory consumption. Using this data as well as artifacts (or lack of) of each method, described in the previous chapter, we formulate a shortlist under what circumstances each of the techniques should be used.

6.1 Frametimes

To rate the different methods in terms of frametimes, the averaged frametimes of different methods in few scenarios are presented.

Frametimes highlighted in red mean that the method results in severe visual artifacts and should not be used without significant modifications. Frametimes highlighted in yellow mean that the method results in slight visual artifacts and can only be used in certain conditions. Said artifacts have been covered in the respective sections of the previous chapter.

	Many lights (2^{14})		Few lights (2^{12})	
	With particles	No particles	With particles	No particles
Forward transparency	21.625 ms	14.756 ms	8.331 ms	6.134 ms
Depth peeling (6 iterations)	78.373 ms	42.014 ms	32.488 ms	21.989 ms
Dual depth peeling (3 iterations)	46.076 ms	23.173 ms	23.742 ms	14.093 ms
Depth-weighted blended OIT	21.311 ms	13.734 ms	8.034 ms	6.224 ms
A-buffer	27.245 ms	13.333 ms	15.478 ms	7.364 ms
A-buffer + depth-weighted blended OIT via cluster collapse	21.477 ms	13.921 ms	10.081 ms	7.792 ms

It can immediately be seen that depth peeling and dual depth peeling have the largest costs. The rest of the methods differ only slightly and their usage will come down to visual performance as well as memory usage.

6.2 Video memory usage

To evaluate the memory cost of each method, theoretical memory usage of additional buffers (not counting G-buffer, light list, etc.) used by each method is presented. It is calculated by simply multiplying the pixel amount (tested using standard 1920×1080 resolution) in a buffer by the data type size used for the pixels. Buffer sizes gathered from GPU captures via RenderDoc (industry-standard graphics debug tool) are presented as well. The differences in these two numbers can be explained by the graphics driver carrying out various internal transformations of the buffers – changing the actual underlying data types, padding the buffers and so on.

	Sum of the theoretical buffer sizes	Sum of the buffer sizes from a RenderDoc capture
Forward transparency	0.0 MB	0.0 MB
Depth peeling (maximum of 6 layers)	205.66 MB	205.66 MB
Dual depth peeling (maximum of 6 layers)	189.84 MB	189.84 MB
Depth-weighted blended OIT	39.55 MB	39.55 MB
A-buffer (maximum of 3 layers if all pixels are covered by transparent objects)	387.6 MB	395.51 MB
A-buffer (maximum of 3 layers if all pixels are covered by transparent objects) + depth-weighted blended OIT via cluster collapse	435.06 MB	466.7 MB

As outlined in the previous chapter the A-buffer (as well as cluster collapse utilising A-buffer) method is by far the most expensive in terms of memory usage. Depth peeling and dual depth peeling consume around half as much memory, however, that is still very much. Especially, considering their strict layer limitation and potential for artifacts due to this limitation. Dual depth peeling should always be considered over original depth peeling, as it has much better performance and uses only slightly more memory.

6.3 Usage shortlist

All in all, we can summarise the findings in a shortlist of when each technique should be used.

If the environment features some combination of the following transparency types, cluster collapse could be used to combine the methods if the VRAM budget allows for it. Otherwise, visual fidelity or the environment will have to be simplified to hide the artifacts. Alternatively, there might be some very-case specific solutions as well.

6.3.1 Simple transparency

Simple geometry

If the environment uses only extremely simplistic geometry with little to no overlapping (e.g. glass panes), forward transparency should be used. Since it produces the correct visual output with minimal overhead both, in terms of frametime and memory. If the objects that are expected to overlap are similar in colour or are spatially very close to one another, depth-weighted OIT could be used as well.

Complex geometry

However, if the environment utilises more complex transparency either dual depth peeling, depth-weighted blended OIT or A-buffer should be used. If it is expected that a lot of objects are going to overlap, A-buffer solution is preferable. If not, and the objects vary wildly in color, world position or both, dual depth peeling should be used. If the colors are similar and the objects are placed in close proximity, depth-weighted OIT could be used.

Forward transparency does not fit this use case due to its failure in primitive sorting, and simple depth peeling has worse performance due to repetitive re-rendering of the object.

6.3.2 Volumetric effects

For vague volumetric effects, like god-rays, forward rendering might be used. However, for well-defined volumetrics, like the refractive Stanford dragons we have presented, either dual depth peeling (not classic depth peeling, due to better performance) or A-buffer can be used, since both methods provide layer data needed for volumetric effects. If geometry is complex, A-buffer method should be preferred.

6.3.3 Particles

Particles only

If the particle system is closely packed in terms of volume and is not expected to contain many very well defined or differently-coloured particles then depth-blended weighted OIT should be used due to its performance, as well as very low memory cost.

Particles overlapping other transparency geometry

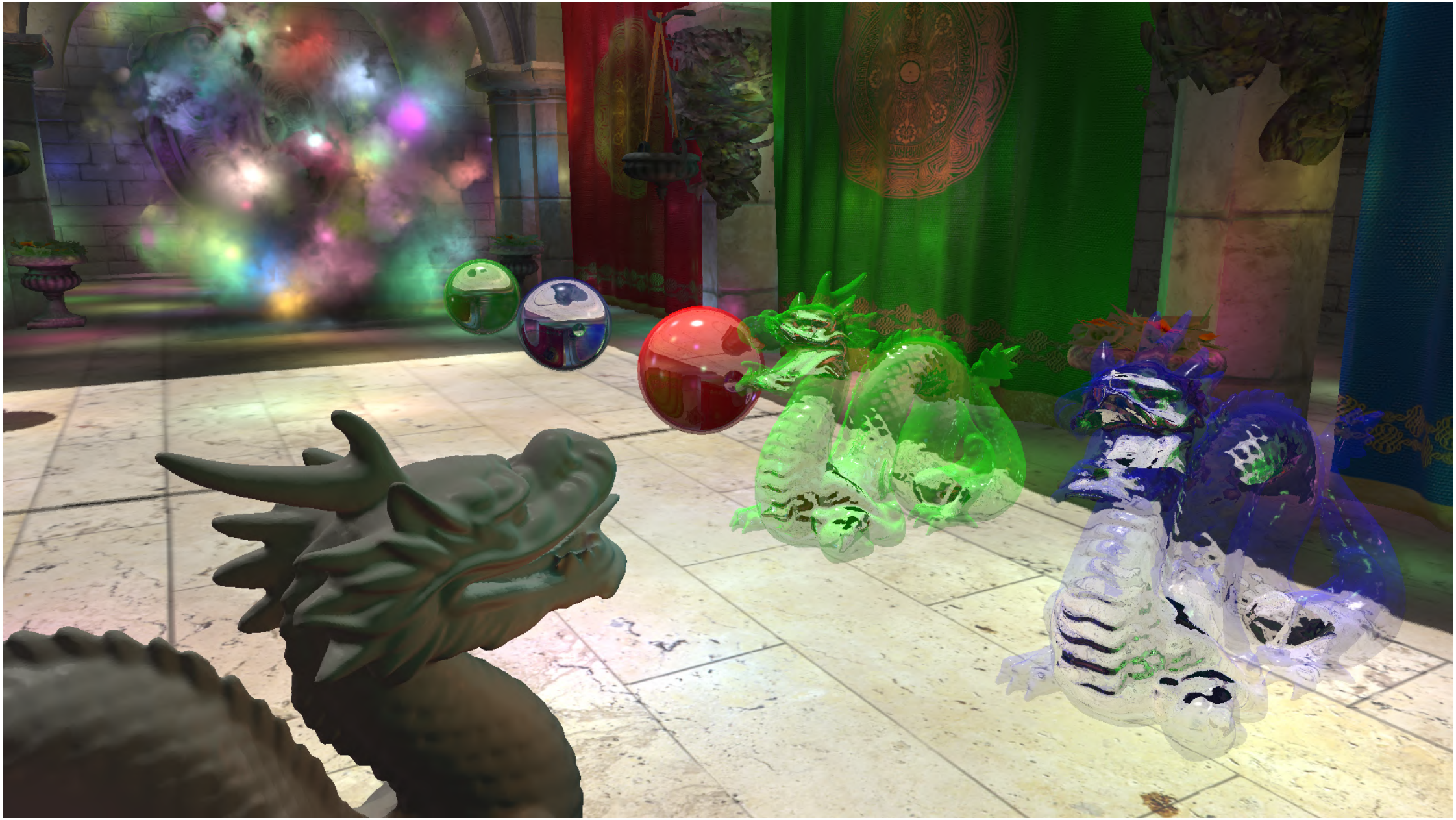
If there are non-particle transparent objects in between particle systems forward transparency might be used under the condition the in-between objects are geometrically simple. However, if the in-between objects are geometrically complex, either the cluster collapse method should be employed or some other case-specific method.

Chapter 7

Conclusion

There is an inherent contradiction between how rendering is done nowadays and transparency as an effect. In spite of that, there are multitude of ways to achieve transparency effects. Simple transparency effects like tinted/textured glass panes can be achieved by using the most primitive forward transparency. However, a much more hefty price needs to be paid in order to be able to *plausibly* render either more complicated geometry or one that has a more life-like appearance (e.g. refractive glass). This price is the main reason why games shy away from intricate transparency effects. We attempt to make these transparency effects more achievable by proposing a novel method of combining various different transparent methods (e.g. figure 7.1). Having such a method allows for the the most performant method to be selected for each object instead of using a single generic method. Thus reducing the overall cost of transparency effects.

Figure 7.1: Cluster collapse: A-buffer for geometry + depth-weighted blended OIT for particles.



Chapter 8

Further work

There is a very large number of methods not covered by this paper. All of them have various trade-offs, use cases and performance profiles that need to be explored.

Most obviously, due to recent advances in hardware, it might now be possible to use various ray-tracing flavours for transparency in addition to the current uses of shadows and reflections. As explored by Wang and Zhang (2021), the transparency effect can be achieved with ray-tracing at acceptable framerates, however, not quite the 60 frames-per-second as expected of games nowadays. Additionally, some older techniques like stochastic transparency (Enderton et al., 2010) were left out. They might exhibit better performance and visual outcomes than the methods covered in this paper. These unexplored techniques might also offer an unrelated way of combining techniques all the while being more performant than cluster collapse. Some recent work, like Friederichs et al. (2021), that builds upon the covered methods also have not been taken into account.

Moreover, how these transparent objects might interact with various shadow techniques was not touched upon. This is likely not a straightforward matter, as many shadow methods, similarly to deferred rendering, assume a single opaque layer of geometry to be casting shadows. Furthermore, coloured shadows and interaction between differently coloured shadows are not taken into account. These might occur when a light is blocked by tinted transparent objects. These types of shadows would be integral for a believable appearance of transparent objects.

Finally, more testing needs to be done in order to confirm that the cluster collapse method does not introduce any visible artifacts when used with objects that are more well-defined or more spatially sparse, unlike a simple smoke particle system, e.g. sparse boids (Reynolds, 1987) of creatures like fish, hair and the like. Furthermore, clustered deferred shading (Olsson et al., 2012) should be investigated as it might have the benefit of automatically clustering transparent objects as a side effect of light clustering, hence assisting cluster collapse.

References

Barta, P., Kovács, B., Szécsi, S. L. and Szirmay-kalos, L. (2011), 'Order independent transparency with per-pixel linked lists'.
URL: *http://jcgt.org/published/0002/02/09/*

Bavoil, L. and Myers, K. (2008), 'Order independent transparency with dual depth peeling'.
URL: *https://my.eng.utah.edu/ cs5610/handouts/DualDepthPeeling.pdf*

Carpenter, L. (1984), 'The a-buffer, an antialiased hidden surface method', *SIGGRAPH Comput. Graph.* **18**(3), 103–108.
URL: *https://doi.org/10.1145/964965.808585*

Catmull, E. E. (1974), A Subdivision Algorithm for Computer Display of Curved Surfaces., PhD thesis. AAI7504786.

Crassin, C. (2010), 'Opengl 4.0+ abuffer v2.0: Linked lists of fragment pages'.
URL: *https://blog.icare3d.org/2010/07/opengl-40-abuffer-v20-linked-lists-of.html*

Deering, M., Winner, S., Schediwy, B., Duffy, C. and Hunt, N. (1988), 'The triangle processor and normal vector shader: A vlsi system for high performance graphics', *SIGGRAPH Comput. Graph.* **22**(4), 21–30.
URL: *https://doi.org/10.1145/378456.378468*

Enderton, E., Sintorn, E., Shirley, P. and Luebke, D. (2010), Stochastic transparency, *in* 'I3D '10: Proceedings of the 2010 symposium on Interactive 3D graphics and games', New York, NY, USA, pp. 157–164.

Everitt, C. (2001), 'Interactive order-independent transparency', *Technical report* .
URL: *https://jcgt.org/published/0002/02/09/paper.pdf*

Friederichs, F., Eisemann, M. and Eisemann, E. (2021), Layered weighted blended order-independent transparency, *in* 'Proceedings of Graphics Interface 2021', GI 2021, Canadian Information Processing Society, pp. 196 – 202.

McGuire, M. and Bavoil, L. (2013), 'Weighted blended order-independent transparency', *Journal of Computer Graphics Techniques (JCGT)* **2**(2), 122–141.
URL: *http://jcgt.org/published/0002/02/09/*

McKee, J. (2011), Real-time concurrent linked list construction on the gpu, AMD, AMD Fusion 11 Developer Summit.
URL: *http://developer.amd.com/wordpress/media/2013/06/2041_final.pdf*

Meshkin, H. (2007), Sort-independent alpha blending, Perpetual Entertainment, GDC 2007.
URL: *https://ubm-twvideo01.s3.amazonaws.com/o1/vault/gdc07/slides/S3721i1.pdf*

Olsson, O. and Assarsson, U. (2011), 'Tiled shading', *Journal of Graphics, GPU, and Game Tools* **15**(4), 235–251.
URL: *https://doi.org/10.1080/2151237X.2011.621761*

Olsson, O., Billeter, M. and Assarsson, U. (2012), Clustered deferred and forward shading, *in* C. Dachsbacher, J. Munkberg and J. Pantaleoni, eds, 'Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics', The Eurographics Association.

Pharr, M. and Fernando, R. (2005), *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*, Addison-Wesley Professional.

Porter, T. and Duff, T. (1984), 'Compositing digital images', *SIGGRAPH Comput. Graph.* **18**(3), 253–259.
URL: *https://doi.org/10.1145/964965.808606*

Reynolds, C. W. (1987), 'Flocks, herds and schools: A distributed behavioral model', *SIGGRAPH Comput. Graph.* **21**(4), 25–34.
URL: *https://doi.org/10.1145/37402.37406*

Saito, T. and Takahashi, T. (1990), 'Comprehensible rendering of 3-d shapes', *SIGGRAPH Comput. Graph.* **24**(4), 197–206.
URL: *https://doi.org/10.1145/97880.97901*

Wang, X. and Zhang, R. (2021), 'Rendering transparent objects with caustics using real-time ray tracing', *Computers & Graphics* **96**, 36–47.
URL: *https://www.sciencedirect.com/science/article/pii/S009784932100039X*