Institut für Visualisierung und Interaktive Systeme
Abteilung Grafische Interaktive Systeme
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3557

Using Per-Pixel Linked Lists for Transparency Effects in Remote-Rendering

Sebastian Alexander Michael Daniel Jähne

Course of Study: Computer Science

Examiner: Prof. Dr. Thomas Ertl

Supervisor: Dipl.-Inf. Daniel Kauker

Commenced: September 09, 2013

Completed: March 11, 2014

CR-Classification: I.3.2, I.3.3

Abstract

Modern graphic cards are highly versatile because they allow the programmer to load custom code to execute onto them. This can be used to construct a structure called a per-pixel linked list, which contains all fragments composing the scene. However, with the need to render more and more complex geometry, even the most powerful hardware reaches its limit fast. To overcome this problem, the geometry is rendered on multiple systems instead of one, and finally put together for rendering. This is called remote rendering and works well for opaque scenes. The goal of this thesis is to conquer rendering of transparent objects remotely using per-pixel linked lists. Since rendering those objects requires a step called blending, standard approaches are incapable of displaying them. Three different methods are shown, compared and analyzed for their usability and performance. First, limiting the amount of depth layers is discussed. Second, identifying regions of visual change is used to reduce the amount of data to be sent. Finally, a way for reusing the previously sent fragments for the current frame is studied in detail.

Kurzfassung

Moderne Grafikkarten sind sehr flexibel aufgrund ihrer Fähigkeit, vom Programmierer verfassten Code auszuführen. Dies kann dazu genutzt werden, eine Datenstruktur names Per-Pixel Linked List zu erstellen. Diese enthält alle Fragmente der zu rendernden Szene. Da aber aufgrund der immer komplexer werdenden Geometrie die Anforderungen stark steigen, erreichen selbst leistungsfähige Grafikkarten schnell ihre Grenzen. Um dieses Problem zu lösen bietet sich das sogenannte Remote Rendering an, bei welchem die Rechenlast auf mehrere Computer im Netzwerk verteilt und am Ende die einzelnen Zwischenbilder zu einem Gesamtbild vereint werden. Für opake Szenen gibt es bereits funktionierende Lösungen. Das Ziel dieser Arbeit ist die Behandlung transparenter Geometrie im Kontext des Remote Renderings mit Hilfe von Per-Pixel Linked Lists. Da das Rendern transparenter Objekte eine Operation namens Blending erfordert, sind die bisherigen Algorithmen meist nicht geeignet. Es werden drei verschiedene Methoden vorgestellt, analysiert und ihre Brauchbarkeit und Geschwindigkeit verglichen. Als erstes wird ein Verfahren, welches die Anzahl an Tiefenebenen limitiert, beleuchtet. Das zweite Verfahren versucht anhand optischer Differenzen zwischen dem aktuellen und dem vorausgegangenen Bild diejenigen Bereiche zu ermitteln, welche für ein korrektes Endbild übertragen werden müssen. Als letzte Technik wird ein Ansatz vorgestellt, welcher die Verwendung von bereits übertragener Fragmente dazu benutzt, um Fragmente einzusparen indem diese wiederverwendet werden.

Contents

1. Introduction 11
   1.1. Related Work 13

2. Methods 15
   2.1. Data Structures 15
   2.2. Linked List Construction 18

3. Depth-limited Rendering 19

4. Visual Difference List 23

5. Bilateral Transformation 27

6. Results 39
   6.1. Test Sequences 39
   6.2. Discussion 42

7. Future Work 45

A. Appendix 47
   A.1. GLSL Shaders 47
      A.1.1. Color Fragment Creation 47
      A.1.2. Normal Fragment Creation 49
   A.2. Measurements 52
   A.3. Algorithms 67
      A.3.1. Fragment Sorting 67
      A.3.2. Depth-limited Rendering 68
      A.3.3. Visual Difference Rendering 69
      A.3.4. Bilateral Rendering 70
   A.4. Notation 72

Bibliography 73

List of Figures

1.1. Compositing . . . 12

2.1. Header and linked list construction . . . 16
2.2. Spherical coordinates . . . 18

3.1. Depth-limited scenarios . . . 20
3.2. Depth-limited rendering with 1 layer . . . 21
3.3. Depth-limited rendering with 2 layers . . . 21
3.4. Depth-limited rendering with 10 layers . . . 21

4.1. Number of rendered fragments for difference rendering . . . 24
4.2. Results for difference rendering . . . 24
4.3. Problems arising with difference rendering . . . 26

5.1. Object space visualization . . . 27
5.2. World space visualization . . . 29
5.3. Eye space visualization . . . 30
5.4. View frustum visualization . . . 31
5.5. Viewport visualization . . . 32
5.6. Normalized device coordinates visualization . . . 33
5.7. Transformation pipeline . . . 34
5.8. Visualization of errors in bilateral rendering . . . 36
5.9. Effects of a large threshold on bilateral rendering . . . 37
5.10. Difference fragments transfered in bilateral rendering . . . 37

6.1. Average memory saved due to usage of LZ4 . . . 40
6.2. Bad pixels for depth-limited rendering in relation to the number of depth layers 40
6.3. Average percent of saved fragments for the bilateral method compared to sending the complete linked list . . . 41

A.1. Percent of bad pixels with increasing number of depth layers for depth-limited rendering . . . 53
A.2. Network traffic using depth-limited rendering for the bunny model with different resolutions . . . 54
A.3. Network traffic using visual difference rendering for the bunny model with different resolutions . . . 55
A.4. Network traffic using bilateral rendering for the bunny model with different resolutions . . . 56

A.5. Network traffic using depth-limited rendering for the dragon model with different resolutions . . . 57
A.6. Network traffic using visual difference rendering for the dragon model with different resolutions . . . 58
A.7. Network traffic using bilateral rendering for the dragon model with different resolutions . . . 59
A.8. Network traffic using depth-limited rendering for the buddha model with different resolutions . . . 60
A.9. Network traffic using visual difference rendering for the buddha model with different resolutions . . . 61
A.10. Network traffic using bilateral rendering for the buddha model with different resolutions . . . 62
A.11. Average network times in milliseconds for the three methods . . . 63
A.12. Average frames per second for the three methods . . . 64
A.13. Percentaged number of difference fragments compared to the original number of fragments for the visual difference method . . . 65
A.14. Percentaged number of difference fragments compared to the original number of fragments for the bilateral method . . . 66

List of Tables

2.1. Fragment structures . . . 17
2.2. Memory requirements . . . 17

A.1. Comparison of uncompressed and LZ4 compressed sizes for different resolutions 52
A.2. Average time required for sending the fragments to the display node in milliseconds over a gigabit lan connection and actual frame rate for different resolutions and methods . . . 53

List of Algorithms

A.1. Sorting of fragments . . . 67
A.2. Depth-limited rendering remote node . . . 68
A.3. Depth-limited rendering display node . . . 68

A.4. Visual difference rendering remote node 69
A.5. Visual difference rendering display node 69
A.6. Bilateral rendering remote node 70
A.7. Bilateral rendering display node 71

1. Introduction

Transparency in 3D graphics is achieved by compositing the fragment colors into an final output color. In interactive applications like games, this is usually done by sorting the geometry, normally consisting of triangles, by depth. Opaque objects are rendered in a first render pass, and in a second render pass the remaining transparent geometry is rendered from back to front, accumulating the colors via a operation called blending, specifically the over operator. A structure called the z-buffer handles occlusion by saving the depth of the rendered pixels, and has to be disabled when rendering transparent objects. This is known as the Painter's algorithm [FDFH90]:

(1.1)

$$c_0 = c_a\alpha_a + c_b\alpha_b \cdot (1 - \alpha_a)$$

c_0 =: Resulting color

c_a =: Front color

c_b =: Back color

α_a =: Alpha value of front fragment

α_b =: Alpha value of back fragment

with the new alpha value for the next compositing step being $\alpha_a + \alpha_b \cdot (1 - \alpha_a)$.

Here is an exemplary calculation showing what happens if the fragments are composed in the wrong order:

Half transparent red rectangle over white background (figure 1.1 (A)):
$(1, 0, 0) \cdot 0.5 + (1, 1, 1) \cdot 1 \cdot (1 - 0.5) = (1, 0.5, 0.5)$ with new $\alpha = 1$.

Half transparent blue rectangle over white background (figure 1.1 (B)):
$(0, 0, 1) \cdot 0.5 + (1, 1, 1) \cdot 1 \cdot (1 - 0.5) = (0.5, 0.5, 1)$ with new $\alpha = 1$.

Half transparent red rectangle over blue rectangle with background (figure 1.1 (1)):
$(1, 0, 0) \cdot 0.5 + (0.5, 0.5, 1) \cdot 1 \cdot (1 - 0.5) = (0.75, 0.25, 0.5)$

Half transparent blue rectangle over red rectangle with background (figure 1.1 (2)):
$(0, 0, 1) \cdot 0.5 + (1, 0.5, 0.5) \cdot 1 \cdot (1 - 0.5) = (0.5, 0.25, 0.75)$

There are different strategies for efficiently sorting the geometry like depth-presorted triangle lists [CSN+12], but problems remain. For example, two intersecting triangles can't be sorted

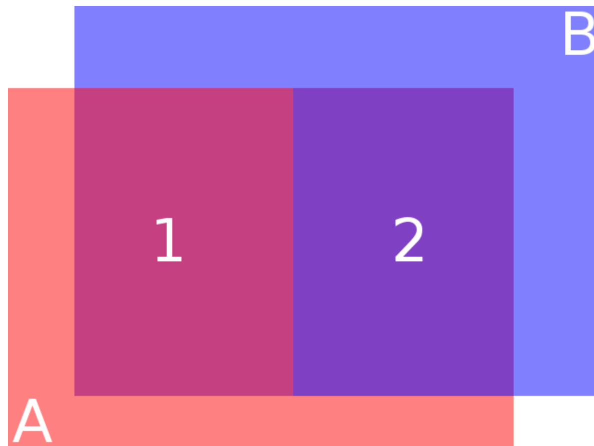

Figure 1.1.: Compositing using the over operator with the red rectangle (A) covering the blue rectangle (B). (1): right order; (2): wrong order

by depth, requiring a dissection of these triangles into new triangles. This segmentation can be done before entering the rendering loop or pre-computed into the model. If the triangles have no connectivity information, testing all triangles for intersections has a runtime of $\mathcal{O}(n^2)$. Also, any pre-computation has to be redone when the model changes and limits the freedom of the methods applied.

To avoid problems like this and for allowing arbitrary geometry, instead of sorting the individual primitves, the fragments created during rendering can be collected and rendered afterwards with a separate shader invocation. Modern shaders and hardware are capable of doing this interactively, simplifying and speeding up the rendering. Algorithms which don't need sorted geometry for correctly displaying transparent objects allow **order-independent transparency (OIT)**. The algorithm described above is realized by utilizing so-called A-buffers [YHGT10], a separate structure which stores the fragments.

If the creation and rendering of the fragments is done on the same machine, even a large amount of fragments does not pose a problem. On our systems, the average frames per second on a single machine were around 400 frames/s for the famous Stanford dragon model[1]. For remote rendering, sending millions of fragments over the network for every unit from a render node to a display node is not feasible. Since the resolution of displays tends to become even bigger, the amount of data produced also grows larger exponentially. For example, a standard resolution nowadays is 1920×1080 pixels. Even if just one fragment of 12 bytes would be transfered per pixel, for a fluent animation of 30 frames per seconds the amount of data would be 712 MiB/s. A gigabit ethernet card has a transfer rate of around 100 MB/s, which is far too less for this scenario. In this work, different methods for reducing the amount of fragment data to be sent over the network which is necessary for displaying an acceptable approximation are studied.

[1] https://graphics.stanford.edu/data/3Dscanrep/

This thesis evaluates three different approaches:

- Depth-limited rendering
- Visual difference rendering
- Bilateral transformation rendering

Note that compressing the fragment data prior to sending it over the network with a fast compression algorithm like LZ4[2], invented by Yann Collet, drastically reduces the amount of data.

1.1. Related Work

Order-independent transparency can be achieved by different means. A very common way to avoid sorting the geometry is to use a technique called depth peeling, invented by Everitt [Eve01]. The idea is to render the scene multiple times with different depth thresholds for each pixel. The previous render pass defines these thresholds and peels the foremost geometry away, exposing the next depth layer. While the geometry is peeled away, the current fragment layer is blended into the framebuffer.

Instead of peeling away a single layer per render pass, Liu et al. are peeling away multiple layers at the same time [LWX06]. This is done by disabling the z testing in OpenGL and performing this test in the fragment shader. Additionally, the created fragments are sorted by depth and the first n fragments are blended and peeled away. The z value of the backmost peeled fragment is stored into a texture and used for the z test in the next render pass. This way, a speedup of roughly the factor n can be achived.

In 2008, Bavoil and Myers adapted the idea of peeling away more than one layer per render pass by peeling away the foremost and the backmost layer at the same time, reducing the number of rendering passes needed by [Eve01] from n to $n/2 + 1$. For this method to work, they defined a min-max range for every render pass, setting it so that the previous front and back layers would be ignored.

The main disadvantage of depth peeling is the need to render the scene multiple times from the same view. To avoid this problem, fragment creation and blending have to be separated. Yang et al. showed a fast method which involves the use of two shader buffers and an atomic counter [YHGT10]. The first shader buffer contains the fragments of the scene, while the second buffer serves as an indexing table which relates screen positions to fragments. This concept is known as an A-buffer and was introduced by Carpenter in 1984 [Car84]. In the first step, the scene is rendered normally onto the screen. However, instead of blending the fragments, they are stored into the first buffer according to their screen position into a single linked list. In order to prevent two fragments at the same screen position and time to overwrite each others values in the linked list, the atomic counter is incremented for every entry in the linked list so that

[2] http://code.google.com/p/lz4/

no two fragments will have the same memory address. Also, the second buffer always points to the last fragment created at the according position. Every fragment contains a pointer to its predecessor, so that in the post-processing step for every screen position all fragments contributing to this position are known and can be sorted and blended.

The methods discussed in this thesis use the work of Yang et al. to create and render the fragments. The main goal is to reduce the amount of data needed to be sent over the network from the remote render node to the display node.

There are two scenarios for remote rendering. The first is when there is only one remote render node with high computational power and a weak display node. In this case, the renderer produces the final image and sends it compressed to the display node. Often the image data is encoded into a video stream ([CBPZ04], [PHE+11]) before sending it to the display node.

The second scenario is when there are multiple render nodes and a single display node which merges the data sent to it. Kauker et al. evaluated the use of per pixel linked lists in remote rendering [KKP+13], pointing out that this structure is better suited for remote rendering because only fragment data which is actually displayed is sent over the network. This feature comes from the fact that the per-pixel linked list is a tightly packed structure, and the amount of fragments can be easily figured out by simply reading the value of the atomic counter used during fragment creation.

It should be pointed out that a per-pixel linked list is not restricted to transparency, but other techniques like screen-space ambient occlusion are also possible with an A-buffer [BKKB]. Kerzner et al. speeded up the order-independent transparency with A-buffers by allocating memory per primitive and not per fragment [KWBG13]. Since the focus on this thesis lies on sending the least possible amount of data over the network, this technique is not suited for this work. There are also techniques which generate an approximation of the final image [SML11] with far less system requirements than the typical A-buffer method.

2. Methods

In order to understand the techniques discussed in this thesis, it is mandatory to know how per-pixel linked lists are constructed and stored. First, the data structures involved are presented. After that, the actual algorithm for creating the linked list is stated.

2.1. Data Structures

Storing the fragments in VRAM[1] is realized by the use of an header, a single linked list and atomic counters like in [YHGT10]. The header simply stores pointers to the last fragments at the corresponding display positions. For a screen resolution of 1920×1080 pixels, and assuming a computer word of 4 bytes, this results in a buffer object of around 8 MiB. Using 4 bytes for indices theoretically allows us to address fragments in 2071 layers with this resolution. Assuming a fragment data structure of 16 bytes, a buffer of 64 GiB would be needed, which is far more than any GPU can store these days.

Since the rasterization produces fragments in parallel and in a hardly predictable manner, the linked list incorporates these fragments in an unordered fashion. Each fragment consists of multiple variables, for example storing the color, the depth, and a pointer to the next fragment at the same screen position. For our purpose, we use the most common color and depth formats. The color is composed of four channels, with a separate channel for red, green, blue, and the alpha value. These four channels are stored in a single DWORD[2], resulting in 256 possible values per channel. The depth and the pointer are also stored in separate DWORDs, represented as floating point numbers. Instead of storing the color directly, the normal can be used. If represented by two half floats, it occupies the same amount of memory as the color fragments.

Because the OpenGL driver is allowed to expand the structure to a bigger size for performance reasons, the actual amount of memory per fragment might be bigger than 12 respectively 24 bytes. This is known as padding and has to be considered for memory transfers from or to the GPU. It also affects the network traffic unless the unneeded parts are pruned. Pruning requires knowledge on the memory layout on the GPU which may differ from graphic card vendor and model and prevents usage of fast memory operations like `memcpy`.

For the bilateral transformation to work, instead of sending the resulting color over the network, the normal has to be sent. The color can not be used because transforming the old

[1] Video Random Access Memory
[2] Double Word, usually equivalent to 4 byte

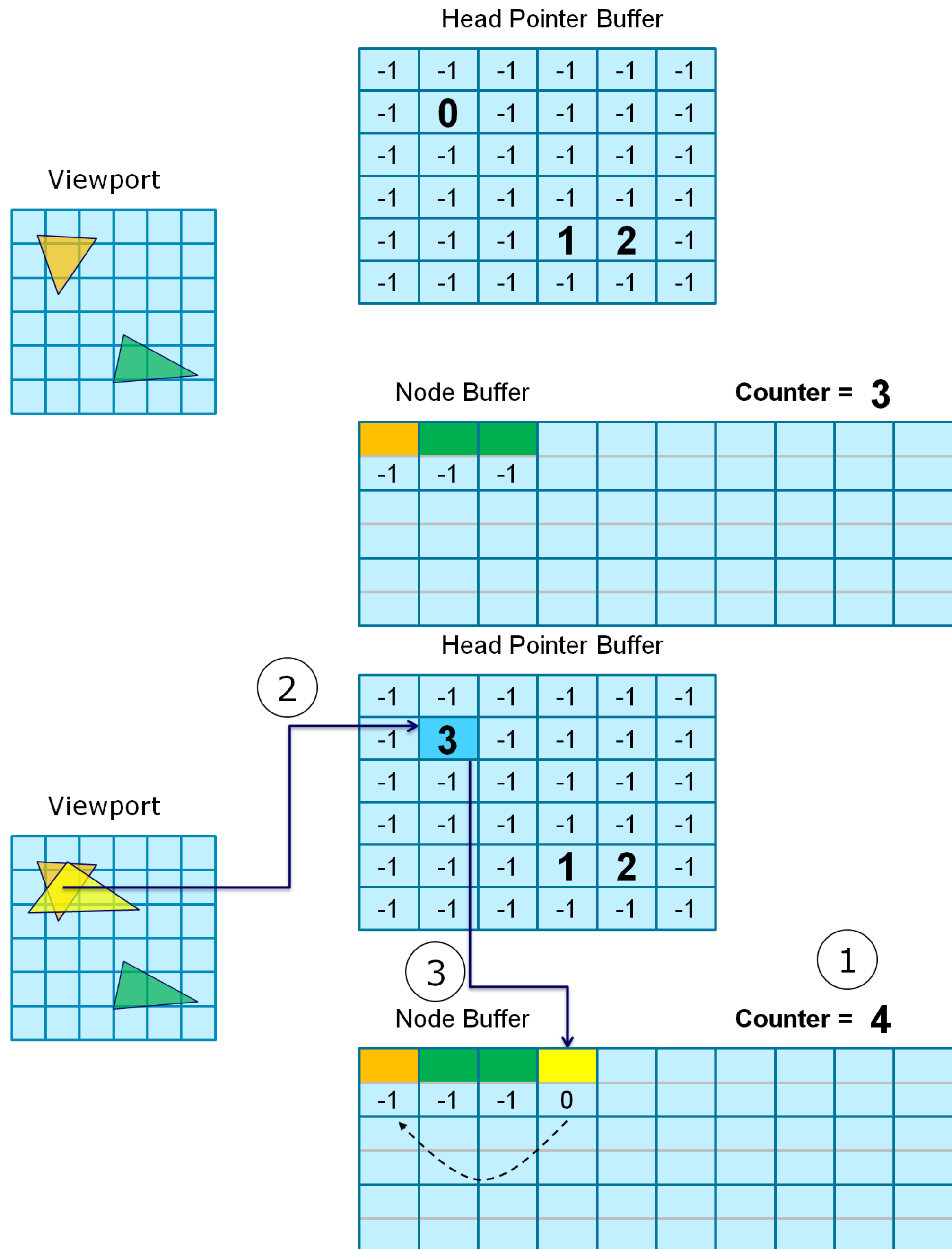

Figure 2.1.: Construction of header and linked list as performed by Yang et al. [YHGT10]. A head pointer buffer references the linked list of fragments created at a specific screen coordinate (2). An atomic counter (1) assures that new fragments created will have a unique storage index into the node buffer. Of course, the head pointer buffer has to support atomic read and write operations. The header will always point to the last fragment created at this position (3), which will reference to its predecessor. This way, the first fragment created will be the last one in the linked list.

Fragment (Color)		Fragment (Normal)	
Variable	Size in bytes	Variable	Size in bytes
Color[1]	4	UV coordinates[2]	4
Depth	4	Normal[2]	4
Next Pointer	4	X	4
		Y	4
		Depth	4
		Next Pointer	4
Total	12 (16)	Total	24 (32)

Table 2.1.: Fragment structures. The number in brackets denote the actual memory requirement due to padding. [1]: four channels (RGBA); [2]: saved as two half-floats

Resolution	1280×720	1920×1080
Header	3600	8100
Linked List (Color)	10800 (14400)	24300 (32400)
Linked List (Normal)	21600 (28800)	48600 (64800)

Table 2.2.: Memory requirements in kibibytes for different screen resolutions. The linked list sizes are given per layer.

fragments also changes the color of them, and computing the new color from the previous color is not possible. Transforming the normals along with the fragments instead results in correct rendering results. Since a normal has always length 1, it is sufficient to only store two components of the vector using spherical coordinates.

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \rightarrow \begin{pmatrix} \arctan\left(\frac{y}{x}\right) \\ \arccos\left(z\right) \end{pmatrix} = \begin{pmatrix} \theta \\ \varphi \end{pmatrix} \text{ with } \theta \in [0; 2\pi], \varphi \in [0; \pi], \sqrt{x^2 + y^2 + z^2} = 1 \tag{2.1}$$

θ denotes the azimuth, while φ represents the polar (inclination) angle, which is the angle originating from the zenith.

The inverse map for getting the original normal is given by

$$\begin{pmatrix} \theta \\ \varphi \end{pmatrix} \rightarrow \begin{pmatrix} \sin(\varphi)\cos(\theta) \\ \sin(\varphi)\sin(\theta) \\ \cos(\varphi) \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \text{ with } \theta \in [0; 2\pi], \varphi \in [0; \pi] \tag{2.2}$$

With these transformations, the normal can be stored into a single unsigned integer using the built-in functions `packHalf2x16` and `unpackHalf2x16` from GLSL without a disturbing visual impact [GP07].

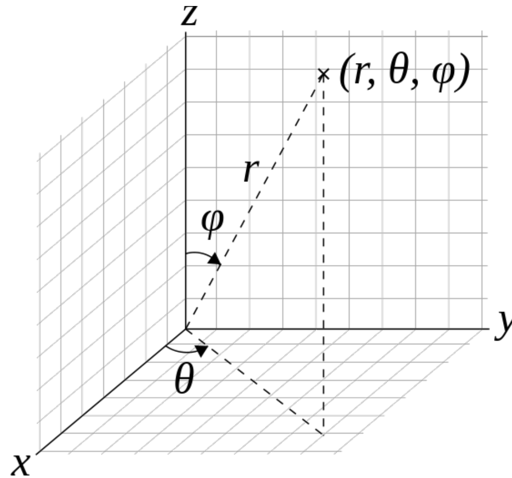

Figure 2.2.: Spherical coordinates often used in mathematics [Wik14]. For normals, r is always 1.

2.2. Linked List Construction

Constructing the linked list is straightforward. We use a vertex and a fragment shader for this purpose. The vertex shader simply passes the fragment position and, depending on the algorithm, either the color or the normal to the fragment shader. The fragment shader reads the current value of the atomic counter, increments the counter and uses its value for indexing the linked list. This assures that every fragment is stored at a different location in memory.

After a unique index has been requested, the new fragment data is written to the memory indicated by the index. To complete the linked list, the next pointer has to be set accordingly. This is done by reading the header value for this fragment atomically, setting the next pointer of the fragment to this value and overwriting the header with the current index. Using this method, the linked list is constructed backwards. The header value has to be read atomically, because two fragments at the same screen position can be rasterized at the same time by the GPU.

The header and linked list can now be sent over the network to the host application, or rendered locally. This list, even compressed, is usually too large to send over the network and achieve interactive frame rates, so we try to prevent sending most of the data by either reducing the number of fragments created or reusing existing fragments, thus preventing them to be sent.

For blending, this linked list alone is not sufficient. In order to apply the over operator, the linked list needs to be sorted back to front. To do this, we use a simple bubble sort on the GPU by drawing a quad over the entire screen and invoking the appropriate vertex and fragment shader (algorithm A.1). With the new compute shaders, this can also be done without invoking another vertex and fragment shader. Because shaders have to be invoked for the final rendering none the less, sorting can also be applied in the rendering stage without the need of pre-sorting the fragments.

3. Depth-limited Rendering

To reduce the amount of fragments created, multiple strategies are possible: The simplest way is to only allow a limited number of fragments to be created per screen position. If the opacity of the fragments is high enough, only a few number of fragments are needed to compute the final color of the corresponding pixel without an visible error. If the scene only contains opaque objects, obviously only one fragment per screen position has to be sent.

This technique can also be applied in an adaptive manner. After sorting the fragments, the opacity of the corresponding pixel using only n fragments can be calculated. If the opacity exceeds a certain threshold, the fragments are omitted. For this purpose, a second header, linked list and atomic counter have to be allocated on the GPU. Since the buffer is read back to system memory as a whole block, limited by the atomic counter, there is no other way to remove the fragments from the linked list. Ohterwise, the data would have to be processed sequentially on the CPU, becoming the bottleneck of the whole application.

If only the final color is sent over the network for each pixel, displaying multiple objects rendered on different clients which overlap each other is not possible since blending them requires the depth of the fragments. Figure 3.1 shows a case where sending a singe pre-blended fragment over the network yields false results. One way to prevent false results is to send the whole fragment list. If the problematic sections are known, all fragments in the section can be sent, while only one fragment per screen position is sent in non-occluding areas.

Finding the occluding areas is not a trivial task. A simple approach would be to render a binary image on each render node, sending this image to the display node and summing all images up. If a pixel in the summed up image exceeds 1, the display node sends requests for the whole fragment list for the specific screen positions to the render nodes, and applies the merging of these fragments itself. While this is a possible approach, other approaches which only require one network transfer per frame seem to be more appropriate.

Since the goal is to support multiple render nodes, pre-merging the fragments is not an option. Instead, the whole fragment list is sent over the network, but the amount of fragments per pixel is limited. Given this limit, it is easy to give limits for the number of fragments:

(3.1)

$$n_{min} = 0$$

(3.2)

$$n_{max} = s_x \cdot s_y \cdot d_{max}$$

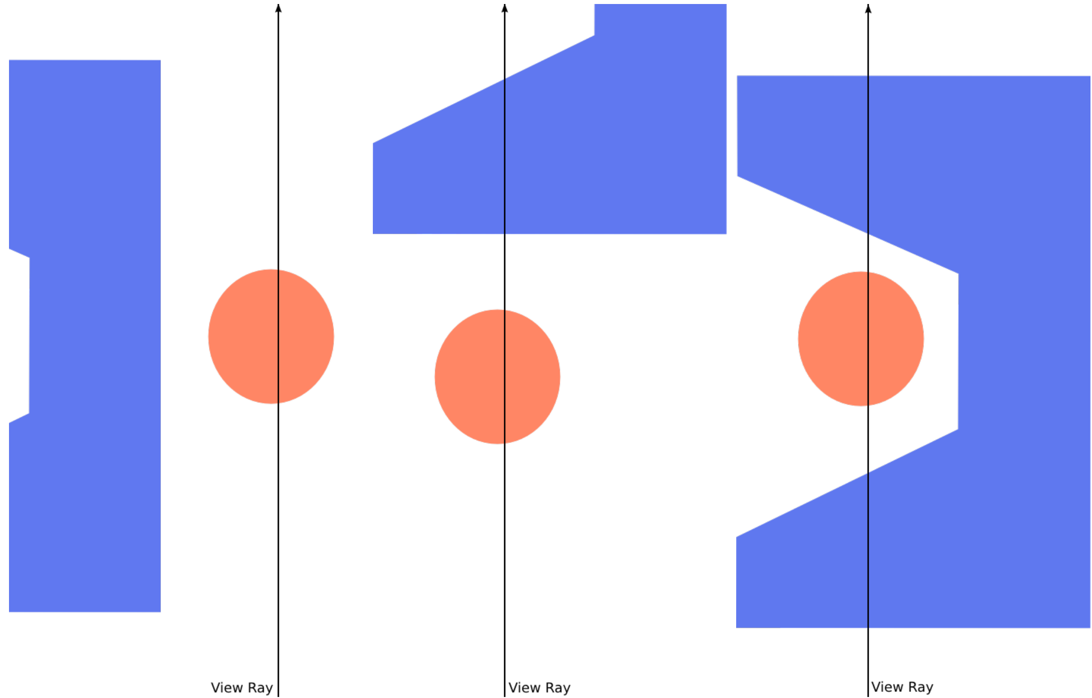

Figure 3.1.: Three different scenarios: The left scenario does only require a final color per fragment. The middle case can be rendered when a depth map is also sent over the network. The right image shows a case where a depthmap is not enough. Instead, all fragments have to be sent including their depth.

with s_x and s_y being the screen resolution in x- and y-direction and d_{max} specifying the maximum of depth layers. Given the fact that a fragment consists of 12 bytes without padding and the header pointers are 4 bytes long, the limits for the uncompressed network flow are

$$f_{min} = s_x \cdot s_y \cdot 4 + n_{min} \cdot 12 = s_x \cdot s_y \cdot 4 \tag{3.3}$$

$$f_{max} = s_x \cdot s_y \cdot 4 + n_{max} \cdot 12 \tag{3.4}$$

For example, assuming a resolution of 1920 × 1080, and a limit of 5 layers, the resulting maximum network flow is:

$$f_{max} = 2073600 \cdot (4 + 12 \cdot 5) = 132710400 \text{ bytes/frame} \equiv 129600 \text{ KiB/frame}$$

In a worst-case scenario, if the desired framerate is 30 frames/s, this would mean a network traffic of 3796 MiB/s. Algorithm A.2 and A.3 summarize the steps performed by the display node and render nodes, while figure 3.2, 3.3 and 3.4 show the visual results for different limits for the depth-limited rendering.

Figure 3.2.: Depth-limited rendering with 1 layer. Left: shaded; Right: number of fragments

Figure 3.3.: Depth-limited rendering with 2 layers. Left: shaded; Right: number of fragments

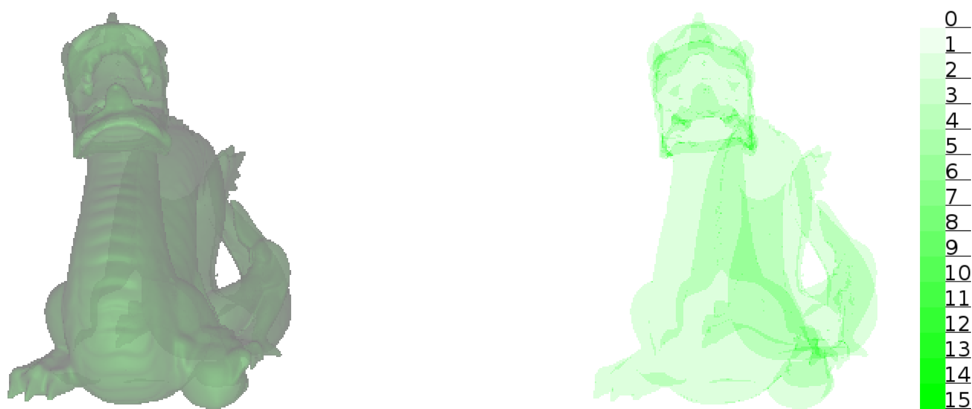

Figure 3.4.: Depth-limited rendering with 10 layers. Left: shaded; Right: number of fragments

4. Visual Difference List

A standard rendering scenario is the rendering of objects in a scene and a camera moving at relatively low speed through it. This means that there are usually very limited visual changes from frame to frame. This fact can be used to further reduce the amount of data which is sent over the network by only transmitting fragments which produce a change in the next frame.

To accomplish this, a form of double buffering is applied. When a new frame is rendered on the client side, the resulting image is compared to the previous rendered image. Should a pixel change its color, all fragments contributing to this new pixel will be transmitted to the host. In order to indicate where changes in the image occured, only a difference header has to be sent to the host. When a pixel does not undergo visual changes, the difference header will point to 0 for this position. Otherwise, it will point to the corresponding linked list of fragments. Of course, this approach can also be combined with depth limitation to further shrink the amount of data.

The fragments are created as usual on the renderer node. If they are created for the first time, the header and linked list are sent unmodified to the display node. They are also blended into a texture on the render node and saved for the next frame. For the subsequent frames, the render node creates the fragments like before, but the blended result is saved into another texture. Using the two textures, the render node can determine visual differences in the images and collect fragments which produce a change in the final image (figure 4.2). The collected fragments are saved and sent to the display node along with the corresponding header. For the next frame, the older of the two textures is overwritten by the blended result, so that these two textures always contain the current and the previous frame. The display node has to simply replace the fragments in its own header and linked list by the indicated fragments. If fragments need to be deleted, this is done by sending a fully transparent fragment for the corresponding position. This can be seen in figure 4.1.

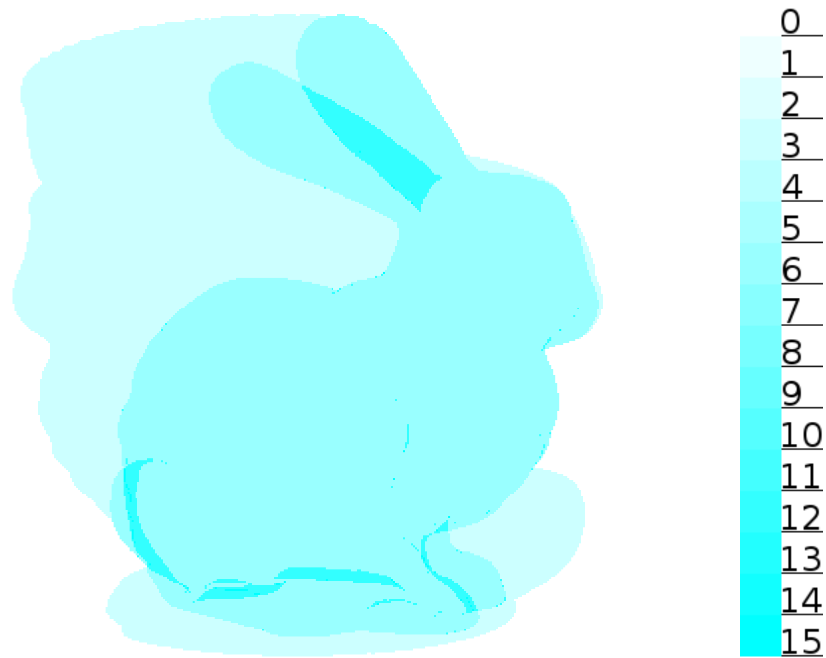

Figure 4.1.: Image showing the number of fragments for each pixel. The header will not point to 0 at positions where fragments were rendered before but to completely transparent fragments instead. Thus the number of fragments rendered on the display node is slightly higher than it is on the render node.

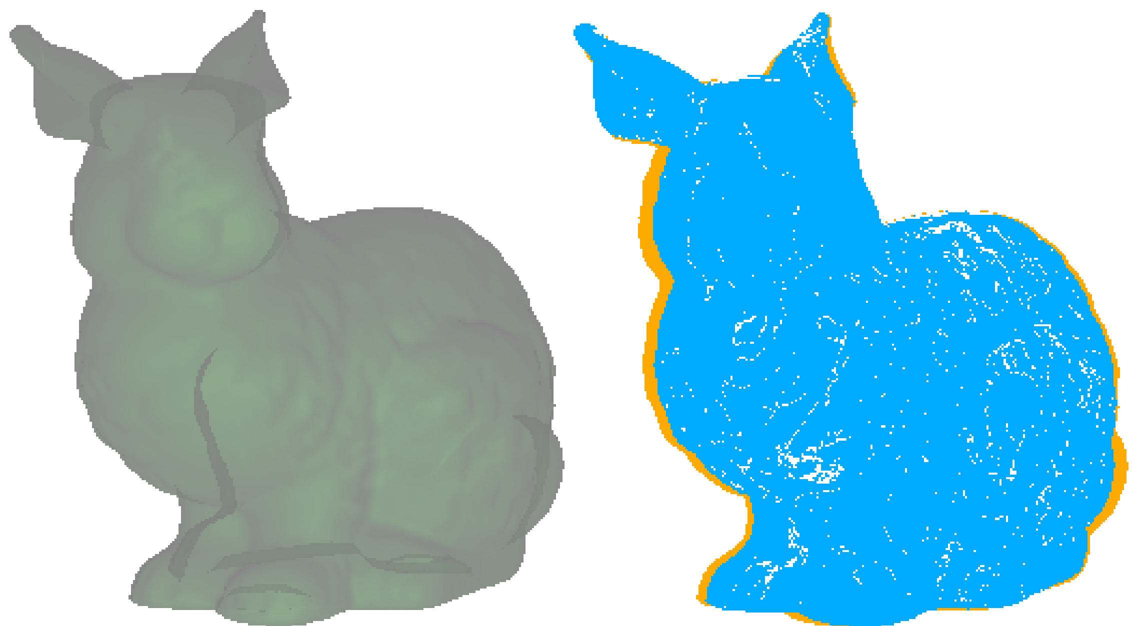

Figure 4.2.: Stanford bunny rendered with difference list. Left: shaded result; Right: difference image where orange indicates removed parts and blue shows changed fragment lists.

While this method works in many cases, there are problems when changes in fragments are only updated partially. Like in the previous method, objects overlapping each other can produce false results. Assume an object moving towards the viewer in the scene, initially lying behind another object. As soon as the moving object breaks through the occluding object, some pixels will not change because the render node does not know about the other objects in the scene (figure 4.3). Overcoming this problem is not easy because the display node would need to keep track of the depths of the fragments for the different render nodes and inform them if a change in occlusion happens.

The visual difference method stores color and depth, along with the pointer to the next fragment into the fragments. Since the render node has to illuminate its own scene the same way as the display node in order to identify changing regions, there would be no benefit in sending the normals instead of the colors. This would just increase the workload on the display node, since the scene would have to be illuminated twice.

The major drawback of this method is the need of keeping a header and linked list for every render node in the display node. If just one header and linked list for all render nodes would be used, the display node is not able to replace only the fragments from the corresponding render node without removing the fragments of the other nodes. But because 4 bytes are unused in the fragment due to padding, this can be used for storing a unique id for every render node. Using this id, the render node can replace the fragments of the corresponding render node and does not need to keep separate headers and linked lists in memory.

The steps performed by the display and render nodes are shown in algorithm A.4 and A.5.

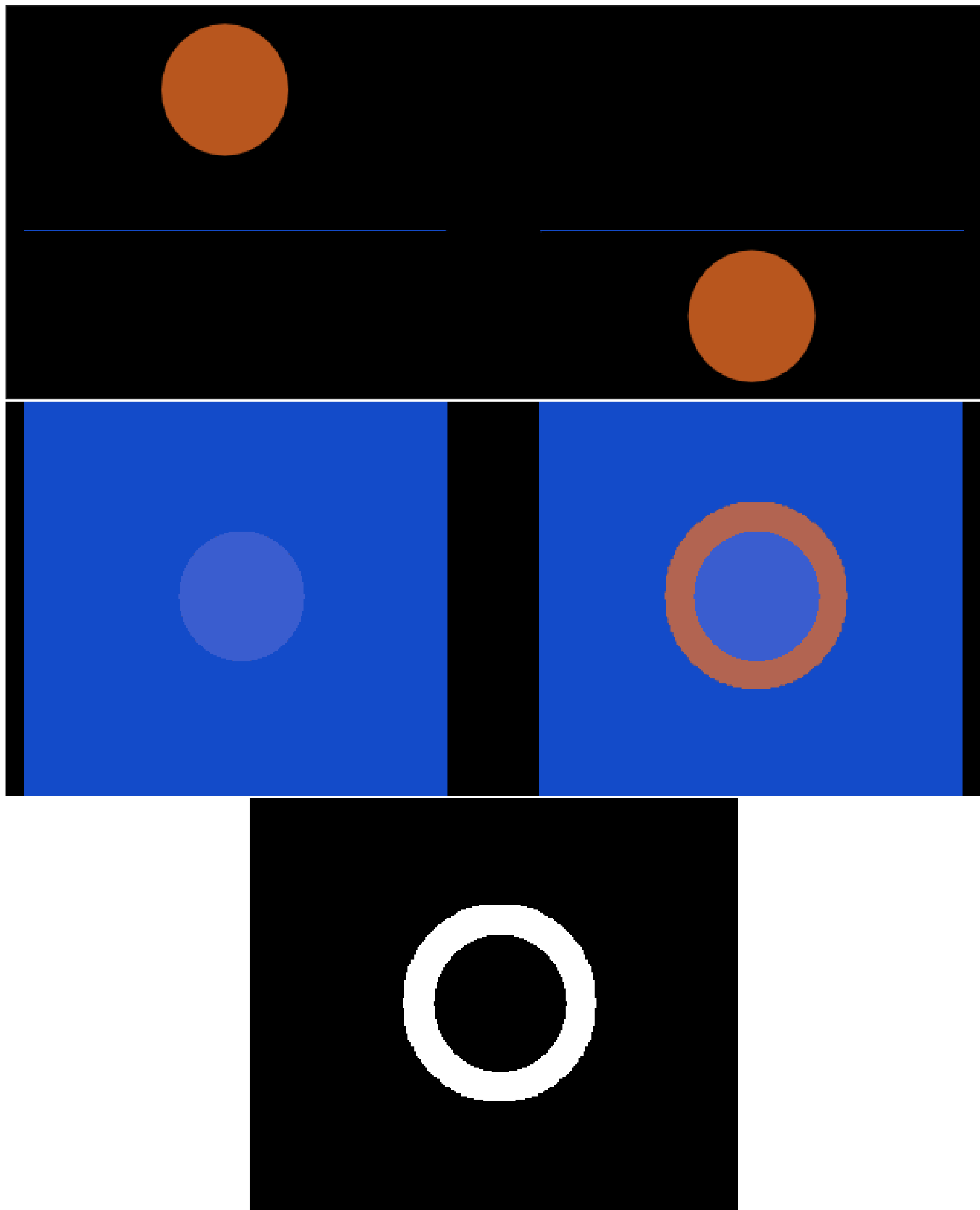

Figure 4.3.: Example case where visual change recognition on the render node alone is not enough for correct visualization. Top image: Top view on two consecutive scenes. Middle image: Resulting image on the display node. Bottom image: Difference as seen by the render node.

5. Bilateral Transformation

In order to overcome the previous problems, one can make use of the model-view matrix. If the previous fragments are transformed with the model-view matrix, there is a chance that they coincide with the new fragments. But since the screen is discretized with pixels, there will hardly be a perfect match.

For understanding the transformation steps, knowledge about the different spaces in OpenGL is crucial. First of all, there is the object space, sometimes called model space (figure 5.1). Every object of the scene has its own object space. The vertices of the object are declared in this space relative to an origin.

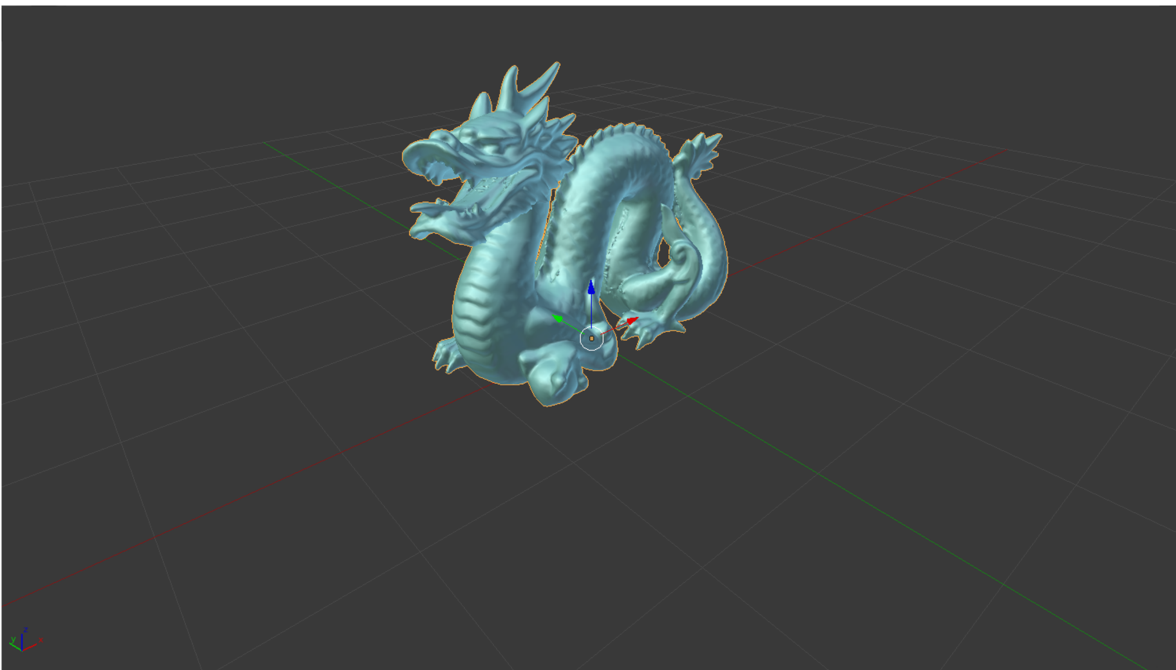

Figure 5.1.: Object space for the stanford dragon[1] in blender[2]

[2]https://graphics.stanford.edu/data/3Dscanrep/
[2]http://www.blender.org/

The objects need to be placed in the scene. For this, a new space is created, called world space (figure 5.2). In order to transform an object into world space, the objects vertices have to be multiplied with the model matrix **M**:

(5.1)

$$\begin{aligned} \vec{\mathbf{p}}_w &= \mathbf{M} \cdot \vec{\mathbf{p}}_o \\ \mathbb{R}^4 \ni \vec{\mathbf{p}}_w &=: \text{position in world space} \\ \mathbb{R}^4 \times \mathbb{R}^4 \ni \mathbf{M} &=: \text{model matrix} \\ \mathbb{R}^4 \ni \vec{\mathbf{p}}_o &=: \text{position in object space} \end{aligned}$$

The model matrix is constructed by multiplying the translation, rotation and scaling matrix in the right order. Since in OpenGL matrices are multiplied from the left side to the vectors, and scaling comes before rotation and rotation comes before translation, the model matrix looks like

(5.2) $\mathbf{M} = \mathbf{T} \cdot \mathbf{R} \cdot \mathbf{S}$

with the translation matrix **T**, the rotation matrix **R** and the scaling matrix **S**:

$$\mathbf{T} = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{R} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(r_x) & -\sin(r_x) & 0 \\ 0 & \sin(r_x) & \cos(r_x) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \cos(r_z) & -\sin(r_z) & 0 & 0 \\ \sin(r_z) & \cos(r_z) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \cos(r_y) & 0 & \sin(r_y) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(r_y) & 0 & \cos(r_y) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{S} = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Notice that the order of rotation matrices is not necessarily the same as shown above. Also, instead of constructing the rotation matrix from euler angles, the rotation matrix is usually aquired from quaternions. This way, an effect called gimbal lock, where in certain situations a rotation will not take effect because two rotation planes are coplanar, can be prevented.

Missing from the scene is information about the view. Instead of placing the camera into the scene, the vertices are transformed into the eye space (figure 5.3), also known as camera space,

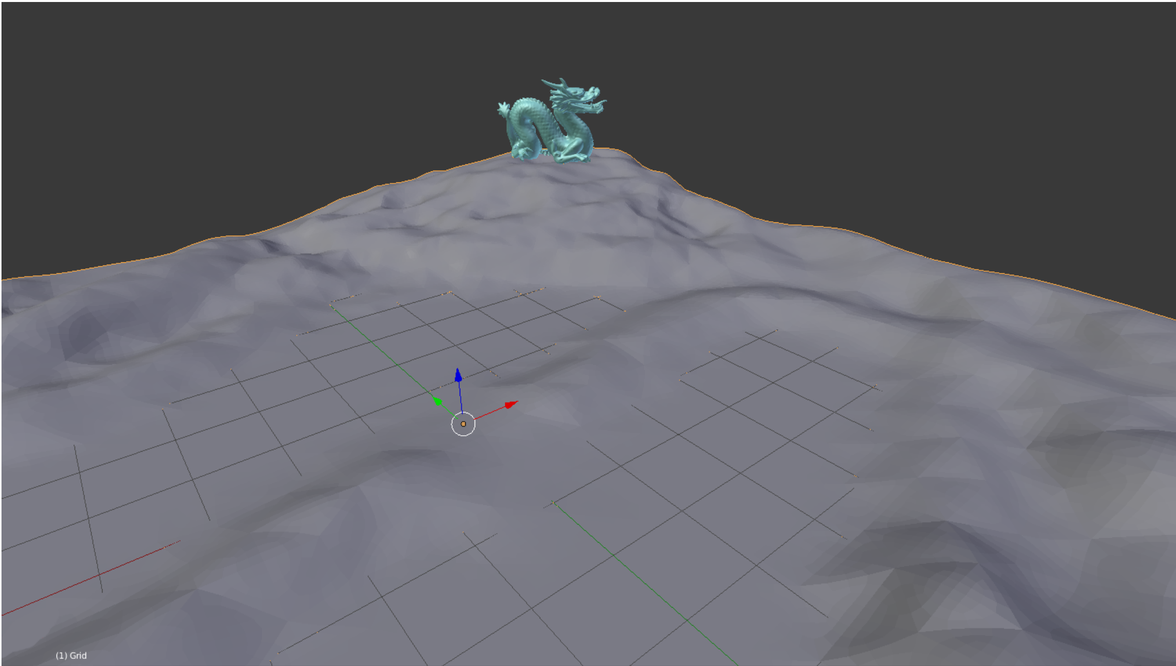

Figure 5.2.: Example for a world space with landscape and model

with the camera at the origin. Given the matrices needed for transforming the camera into world space, a view matrix $\mathbf{V}$ can be constructed from them by simply inverting the product of these matrices. The view matrix is then applied to every vertex in the world with equation (5.3).

$$\vec{\mathbf{p}}_e = \mathbf{V} \cdot \vec{\mathbf{p}}_w \tag{5.3}$$

$$\begin{aligned} \mathbb{R}^4 \ni \vec{\mathbf{p}}_e &=: \text{position in eye space} \\ \mathbb{R}^4 \times \mathbb{R}^4 \ni \mathbf{V} &=: \text{view matrix} \\ \mathbb{R}^4 \ni \vec{\mathbf{p}}_w &=: \text{position in world space} \end{aligned}$$

The vertices now lie relative to the camera. For a realistic impression of the scene, a perspective transformation has to be applied. This transformation takes care of the phenomenon of points distant to the viewer being nearer to each other than points close to the viewer. Equation (5.4) realizes this transformation.

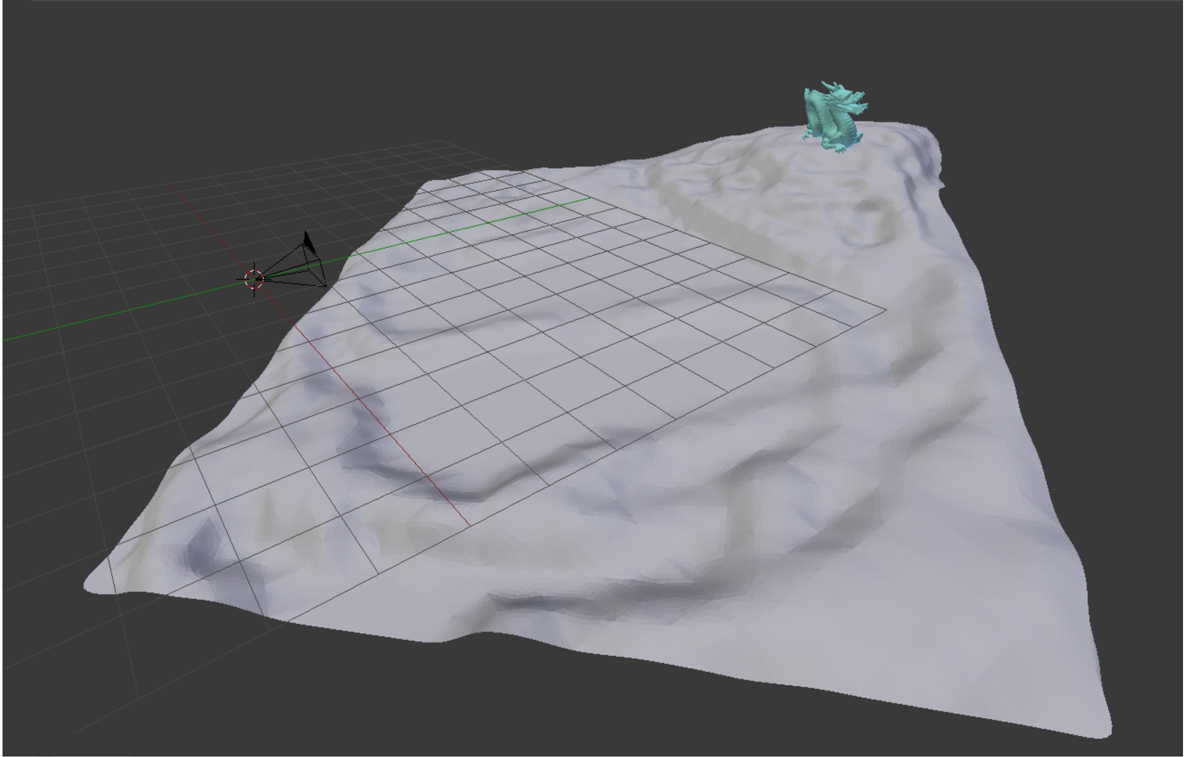

Figure 5.3.: The scene transformed into eye space for a specific camera

(5.4)

$$\vec{\mathbf{p}}_c = \mathbf{P} \cdot \vec{\mathbf{p}}_e$$

$$\begin{aligned} \mathbb{R}^4 \ni \vec{\mathbf{p}}_c &=: \text{position in clipping space} \\ \mathbb{R}^4 \times \mathbb{R}^4 \ni \mathbf{P} &=: \text{projection matrix} \\ \mathbb{R}^4 \ni \vec{\mathbf{p}}_e &=: \text{position in eye space} \end{aligned}$$

Given the view frustum, geometry not seen by the viewer can be clipped, which is why this space is called clipping space, and the operation is called frustum clipping. To get the final screen coordinates, the vertices are then transformed into normalized device coordinates (figure 5.6) by dividing the clip coordinates with the w coordinate of the vector. This perspective division is applied by equation (5.5).

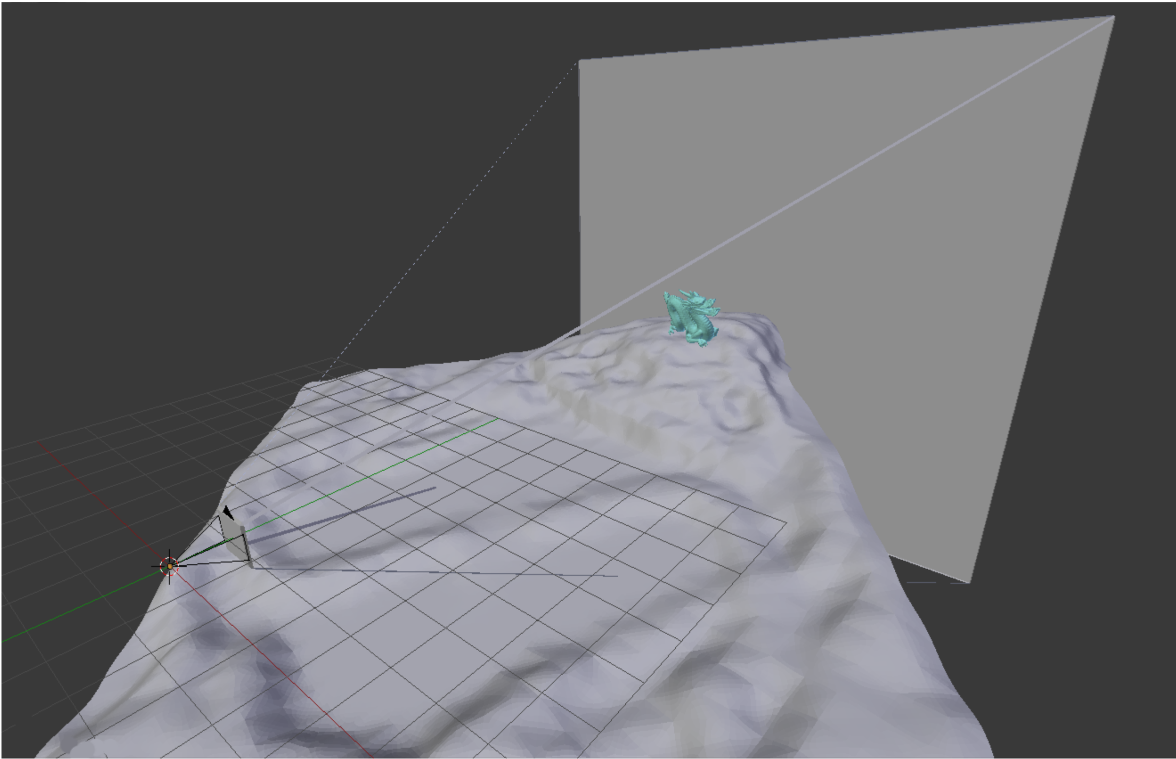

Figure 5.4.: Example for a view frustum

(5.5)
$$\vec{\mathbf{p}}_{ndc} = \frac{1}{\vec{\mathbf{p}}_{c_w}} \cdot \vec{\mathbf{p}}_c$$

$$[-1;1]^3 \times \{1\} \ni \vec{\mathbf{p}}_{ndc} =: \text{position in normalized device coordinates}$$
$$\mathbb{R}^4 \ni \vec{\mathbf{p}}_c =: \text{position in clip coordinates}$$

This simplyfies the last transformation step, called viewport transformation. The viewport defines the actual screen coordinates with a rectangle (figure 5.5). The transformation is done by the simple linear mapping in equation (5.6).

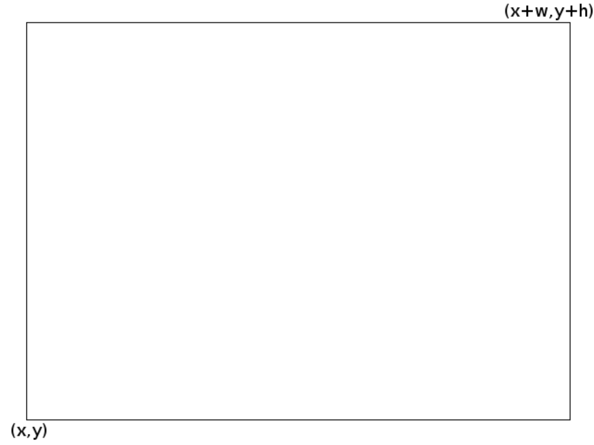

Figure 5.5.: General setup for a viewport

(5.6)

$$\vec{\mathbf{p}}_s = \begin{pmatrix} \frac{w}{2} & 0 & 0 & x + \frac{w}{2} \\ 0 & \frac{h}{2} & 0 & y + \frac{h}{2} \\ 0 & 0 & \frac{f-n}{2} & \frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \vec{\mathbf{p}}_{ndc}$$

$$\begin{aligned} [x; x+w] \times [y; y+h] \times [n; f] \times \{1\} \ni \vec{\mathbf{p}}_s &=: \text{position in screen coordinates} \\ [-1; 1]^3 \times \{1\} \ni \vec{\mathbf{p}}_{ndc} &=: \text{position in normalized device coordinates} \\ x &=: \text{lower left x-position of the screen} \\ y &=: \text{lower left y-position of the screen} \\ w &=: \text{width of window} \\ h &=: \text{height of window} \\ n &=: \text{distance of near clipping plane to camera} \\ f &=: \text{distance of far clipping plane to camera} \end{aligned}$$

The fragments are now in screen space and can be renderer. In order to be able to apply the inverse transformation required for the bilateral transformation, and since the only matrices which change from frame to frame are the model matrices and the view matrix, one has to know the positions of the fragments in either eye space, clip space, normalized device space or screen space. To achieve higher accuracy for the transformation by performing less matrix multiplications and avoiding normalization, x, y and the depth value are stored in eye space. As long as the coordinates are stored in the same space, and given the previous and current transformation matrices, they can be projected into every space of the transformation pipeline, including the screen space for blending. Note that since the normals are stored in object space,

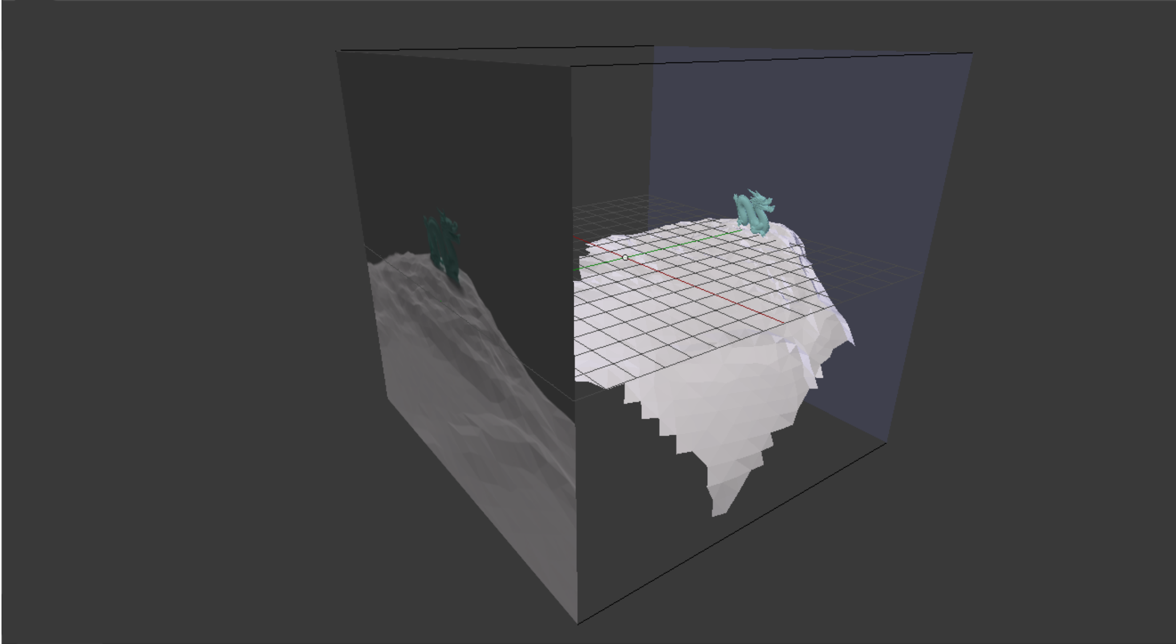

Figure 5.6.: Normalized device coordinates with clipped geometry

no inverse transformation has to be applied to them. To transform the normal into eye space, it has to be multiplied with the transposed inverse model-view matrix. Equation (5.7) shows the correct way to transform normals.

$$\vec{\mathbf{n}}_e = \left((\mathbf{V} \cdot \mathbf{M})^{-1}\right)^\top \cdot \vec{\mathbf{n}}_o \tag{5.7}$$

Given a position in object space, the vertex shader usually applies the matrix multiplications in equation (5.8) prior to the fragment shader:

$$\vec{\mathbf{p}}_c = \mathbf{P} \cdot \mathbf{V} \cdot \mathbf{M} \cdot \vec{\mathbf{p}}_o \tag{5.8}$$

with the model matrix $\mathbf{M}$, the view matrix $\mathbf{V}$, the projection matrix $\mathbf{P}$, the position in object space $\vec{\mathbf{p}}_o$ and the resulting position in clip space $\vec{\mathbf{p}}_c$. OpenGL takes care of the clipping and normalization for us before sending the coordinate to the fragment shader.

Instead of storing the fragments position given by `gl_FragCoord` in the fragment shader, the position is saved in eye space $\vec{\mathbf{p}}_e$ with equation (5.9).

$$\vec{\mathbf{p}}_e = \mathbf{V} \cdot \mathbf{M} \cdot \vec{\mathbf{p}}_o \tag{5.9}$$

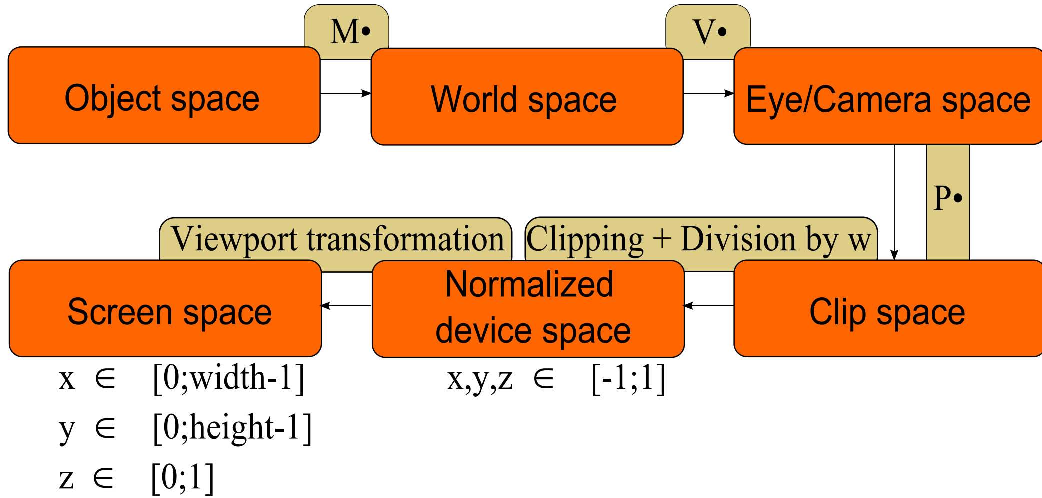

Figure 5.7.: Transformations applied to a vertex.

If the position from the previous frame is known, and given the old transformation matrices $\mathbf{M}_p$ and $\mathbf{V}_p$, one can calculate the previous positions by multiplying the stored positions $\vec{\mathbf{p}}_{e_p}$ with the inverse matrices $\mathbf{M}_p^{-1}$ and $\mathbf{V}_p^{-1}$ using equation (5.10).

$$\vec{\mathbf{p}}_o = \mathbf{M}_p^{-1} \cdot \mathbf{V}_p^{-1} \cdot \vec{\mathbf{p}}_{e_p} \quad (5.10)$$

The original positions are then transformed the same way as in equation (5.9). In order to get the x and y positions in screenspace for storing the fragments with the correct index for the header, this position has to be multiplied by the projection matrix of the current frame and normalized in the same way as in equations (5.4) and (5.5).

The normalized coordinates lie in a unit cube ranging from -1 to 1 in each direction. To get the x and y position in screen space, equation (5.6) is used.

With the fragments from the previous frame and the current frame, the render node has to identify the fragments which are missing from the transformation or become invalid after transformation. To do this, both fragment lists are first sorted from front to back. Then, the shader walks through both lists at the same time, comparing the difference of depths of the fragments. If the difference exceeds a depth threshold, depending on wether the transformed fragment lies before the new fragment, different actions have to take place. Otherwise, the transformed fragment is accepted, without the need to send it over the network.

When the difference in depth of two fragments is too big, it will be inserted into the difference list. Additionally, the fragments contain a flag which marks them for insertion or removal.

The render node has to insert and remove the fragments into the transformed fragment list like the display node in order to construct the correct difference list. Hence, the render node has to keep two headers and linked list in memory, one representing the previous frame, the other one storing the current frame. This is the biggest drawback of this method. For every object, and thus for every render node, the display node has to keep a header, a linked list and transformation matrices representing the previous frame in memory.

The render node constructs a difference list by transforming the fragments from the previous frame as described above. These transformed fragments are then compared to the fragments of the new frame, and then are either marked for removal, new insertion or put on an ignore list. All fragments which are to be removed or inserted are put into the difference list, since the fragments which should be ignored are implicitly defined by not being in the difference list. These fragments are actually the fragments which don't need to be sent over the network, saving traffic. The render node stores the fragments which are to be ignored and inserted in its own list in order to know the configuration on the display node.

Applying the differences computed by the render node to the display node is done by a specific merging strategy. Depending on the flag set by the render node, different actions take place. If the difference fragment is marked for insertion, nothing special has to be done. If however the fragment should be removed from the transformed list, all transformed fragments which occupy the same pixel as itself are compared by depth value to the difference fragment, and the nearest one is chosen for deletion. Depending on the threshold for depth comparison, some fragments will be deleted which shouldn't be deleted while others stay in the transformed list depending on the order the difference fragments are processed. Hence, the merging of the fragments did not yield perfect results on the display node (figure 5.8), while the render node has all information it needs to reconstruct the perfect representation.

The whole process is summarized in algorithm A.6 and A.7.

Figure 5.8.: Visual differences between images. Top left: image on display node. Top right: image on render node. Bottom: bad pixels which are different between the render node and the display node

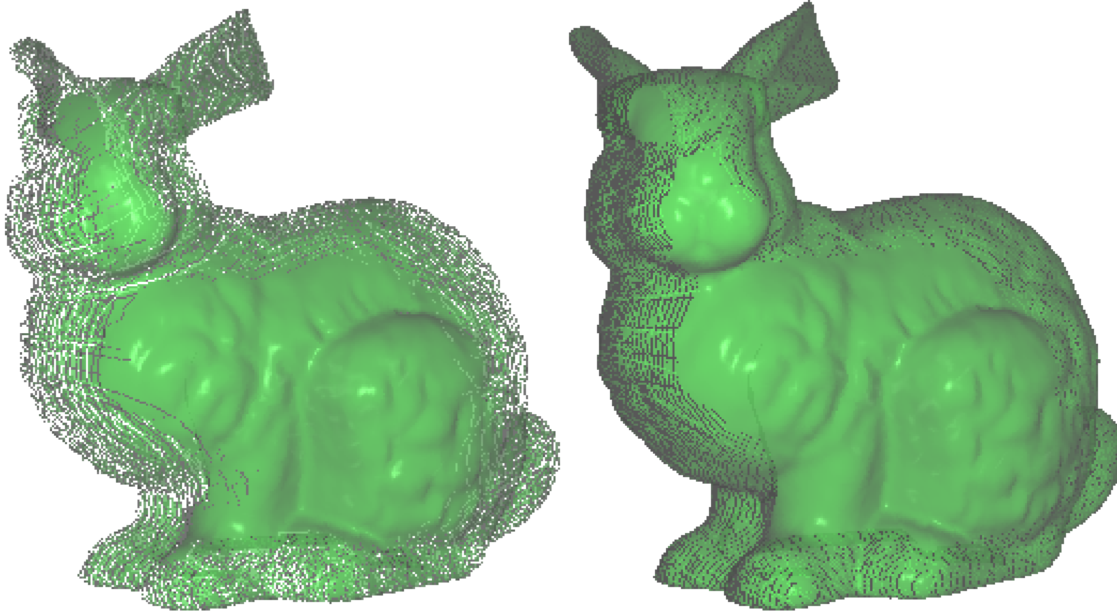

Figure 5.9.: Effect of a threshold for depth comparison which is too big. The bigger the threshold is, the more wrong fragments will be removed.

Figure 5.10.: Bilateral difference fragments, Orange framents are to be removed, blue fragments are to be inserted.

6. Results

Here is a summary of the tests performed and the corresponding results. The collected data is presented in section A.2.

6.1. Test Sequences

The tests were performed between two systems: The display node (Intel Core i7 3820, Nvidia Titan 6GB) and the render node (Intel Core i7 3820, Nvidia GTX 680 4GB), connected by a gigabit ethernet connection. Three different models, the stanford bunny, dragon and buddha model[1], were rotated 5 degrees per frame around their local y-axis with different resolutions for every method. The main focus was placed on the possible reduction of fragment data needed for remote rendering. Also, the effects of LZ4 on the total amount of data was of interest.

It was found that LZ4 compression has a big impact on the data size. Using compressed data instead of uncompressed data resulted in a total reduction from around 25-45%, depending on the resolution (figure 6.1).

The fastest method was the depth-limited rendering, since no additional buffers were required on either the render nor the display node. Both nodes only performed a single shader invocation, so the limiting factor was given solely by the network. The amount needed to be sent over the network is dependant on the actual transparency of the objects. For objects which are hardly transparent, very few layers have to be sent. The objects for the test sequences always had a constant alpha value of 0.5. Using this value, the amount of bad pixels was determined in conjunction with the amount of depth layers (figure 6.2). Sending more than 6 layers of fragments didn't produce a noticeable visual change in the test sequences.

The difference method was very stable considering the number of difference fragments as the test sequences progressed. The amount of transfered fragments was around 47% compared to the complete linked list and didn't change under different resolutions or for different models. The render node has to invoke two shaders, one for fragment creation and the other one for identifying parts in the image which differ compared to the previous frame. Also, another header and linked list are needed for storing this information, in addition to two textures which store the previous and the current frame. The display node has to have three headers and linked lists. The first header and linked list store the fragments from the previous frame, the second pair receives the difference from the network, and the third header and linked list

[1] https://graphics.stanford.edu/data/3Dscanrep/

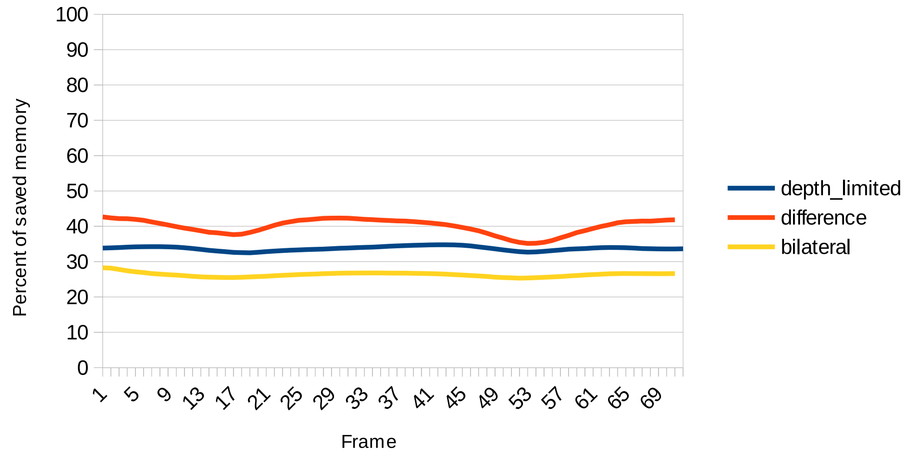

Figure 6.1.: Average memory saved due to usage of LZ4

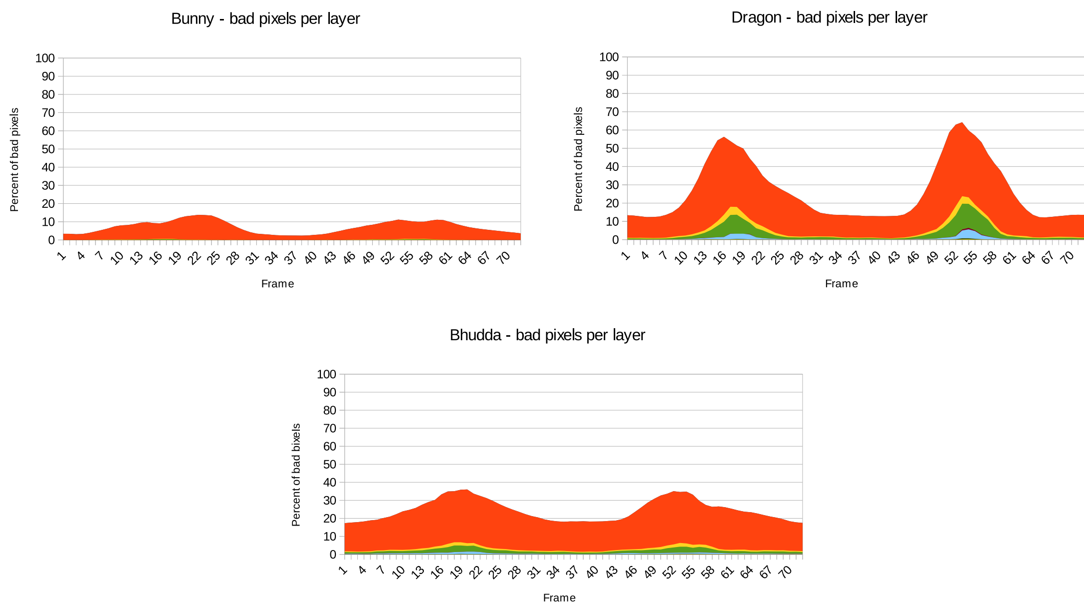

Figure 6.2.: Bad pixels for depth-limited rendering in relation to the number of depth layers

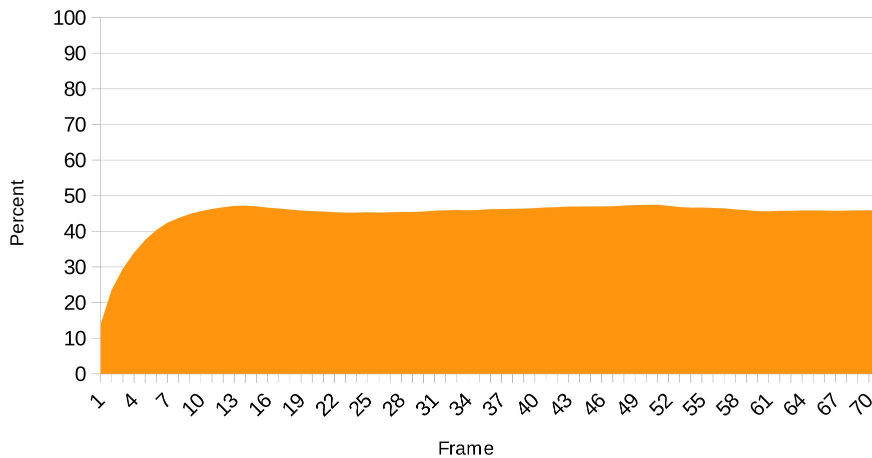

Figure 6.3.: Average percent of saved fragments for the bilateral method compared to sending the complete linked list

are used for merging the differences with the previous fragments. Merging and displaying can be done in a single shader. The program used for creating the results presented in this thesis however used separate shaders for merging and displaying.

For the bilateral method, the number of difference fragments started low and increased over time until it reached a maximum at around 48% reduction compared to the actual linked list size (figure 6.3). The low number at the beginning results from the small deviations of transformed fragments at the beginning of the rendering. As time increases, more and more small aberrations due to the limited floating point precision accumulate, increasing the number of difference fragments significantly. In addition, this method needs even more buffers then the vidual difference method. The render node needs two additional buffers for the transformed fragments, which together with the buffers for fragment creation and difference fragments gives a total of 6 buffers. The situation is even worse on the display node, because it has to keep a header and linked list for every object of the scene since they are transformed differently. While two buffers are enough to receive the difference lists, and two more are needed for merging the transformed fragments with the difference buffers, the number of buffers kept by the display node increases by 2 with each separate object in the scene.

As already mentioned in chapter 5, the merging of difference fragments on the display node was not perfect for the bilateral method, resulting in small artifacts. These artifacts are noticeable upon direct compare between the images produced by the display and the render node, but are usually not interfering much with the final visual result.

6.2. Discussion

The three methods discussed in this thesis differ in memory requirement, computational workload, network traffic and visual results. The fastest method is depth-limited rendering. It involves only one shader invocation per node and needs only two buffers. The fragments produced on different render nodes can simply be inserted in the fragment list on the display node. By specifying a depth layer threshold, balancing the visual results with the network traffic is easy and straightforward.

The visual difference method is a fast alternative, which can reduce the amount of data sent over the network by 50% when only small changes occure. It is easy to implement, and does not require a lot of additional memory on either node. It produces perfect visual results when there is no overlapping geometry in the scene. The two additional textures used for storing the previous and the current frame can be neglected compared to the buffers used for fragment storage. There is no need to introduce an additional threshold, but if desired this method can also be depth-limited.

The most complex method is the bilateral transformation. It involves transforming the previous fragments with the corresponding inverse transformation matrices, and requires a careful choosing of the threshold used for comparing the depth of the transformed with the difference fragments. If the threshold is too large, there will be many falsely deleted fragments. If the threshold is too small, the render node will send a difference list of twice the size of the actual fragment list. Every transformed fragment will be deleted, and every fragment of the current frame will be inserted into the difference list. Using the merge strategy in this thesis did not produce perfect results, but the visual impact was small. The memory requirements for this method are high, as well as the workload on the GPU. The render node has to invoke three different shaders: The standard fragment creation, a shader which transforms the previous fragments and finally a shader which identifies fragments which are to be removed from or inserted into the transformation list. Also, a flag has to be stored for each fragment in order to identify wether it will be inserted or removed along with the x and y coordinate of this fragment in eye space for transformations. Due to padding, this resulted in a twice as large fragment size compared to the other two methods.

Compressing the data with a very fast algorithm like LZ4 is highly advisable because it drastically reduces the network load with almost no cost to the framerate. Especially for fragment structures which provoke padding by the OpenGL driver, compression reduces the additional unnecessary transfers per frame due to the low entropy present in the padded data.

As represented in figure A.4, A.7 and A.10, different resolutions have a direct effect on how efficiently the difference list can be compressed. Despite the fact that at a resolution of 1280×720 the number of differences fragments was lower than at the other resolutions, the compressed data size was not the smallest. This comes from the fact that far more transformed fragments were identified by the remote renderer to be wrong and were replaced by new ones compared to the other resolutions. Due to the labeling of fragments for removal or insertion, this resulted in a linked list which was not well suited for compression.

From all the methods, the depth-limited approach was the fastest while needing less memory then the other two techniques. For objects which offer only a limited amount of transparency, this method seems like the most favorable solution. If there is more memory available on the system, the visual difference method is another good way to conquer transparent objects, even more so because the visual results are the best.

The bilateral transformation method is difficult to handle, especially debugging can be a tedious task. Since the transformation is usually not perfect due to the limited precission of floating point numbers, one has to take care of the erroneous fragments. Additionally, the depth threshold used for comparing the transformed fragments with the newly created ones depends on the view and objects which are to be rendered. Also the memory requirements and GPU load are high, so this method is only suitable for a setup which provides large amounts of memory and fast graphic processors.

All three methods are very slow compared to a local order-independent transparency implementation on a single system. The test system created and rendered the per-pixel linked list with around 400-500 frames/s locally compared to the average 5 frames/s achieved by the depth-limited method, showing that the memory transfer is the critical step for remote rendering.

7. Future Work

There are many improvements to the algorithms described in this thesis which can save memory, computation time and improve quality. The most important and obvious one is to replace the discussed merging strategies with more sophisticated ones. For the bilateral transformation method, the fragment data could be presorted on the render nodes. Using an already sorted linked list, merging could be simplified and speed would increase. A very tempting idea is to blend as many fragments on the render nodes as possible, since this would lower the network traffic as well as speed up the time consuming merging process on the display node. If objects in the scene are not overlapping, it is legit to blend these fragments prior to sending them. It would also lower the memory requirements for the display node since not every fragment visible would have to be stored temporarily for merging, allowing even bigger models for rendering.

If there are overlapping objects in the scene, the situation becomes more complicated. How should the render node know which fragments can be blended without creating artifacts in the final image? Decompositing the scene by depth may be a way out of this dilemma. If the different render nodes would not render one object per node, but instead render a specific depth range, blending the fragments prior to sending them is obviously feasible and adviced. This poses the problem that each render node has to know all object lying in this depth range. Since the camera is normally not fixed in the scene, the objects which each render node will process will change over time, so some form of subdivision of the scene has to be applied, for example octrees. If the geometry of the scene is stored in the nodes of the octree, these nodes can be efficiently distributed onto the render nodes, and blending them beforehand can be performed. This is especially interesting for static scenes, where generating the octree once is enough and has not to be updated for the rest of the time.

If the bilateral transformation method is used, it makes sense to store the previous fragments of all render nodes into a single linked list while still keeping a header for every render node in order to save memory. Making use of the unused bytes in the fragment due to padding like in the visual difference method, they can be marked to be a part of a specific model, so that the correct transformation matrices can be applied to them as well as the correct merging.

In order for the merging to be efficient and correct, it has to be restricted to the fragments of the same object as in the difference list. Since the difference lists are read sequentially from the render nodes, and the display node keeps headers for every object, this restriction is possible if the merging of fragments is performed by separate shader invocations. The time used by these invocations can be used to read the next fragment list from another render node, making use of the fact that the graphic card and the CPU can work in parallel.

In this thesis, the test system was restricted to one display node and one render node. The main goal was to proof the feasibility of the discussed method, and so optimizations and

support for multiple render nodes was omitted. It is clear that a more sophisticated and balanced implementation which facilitates more than one render node are required for a working environment.

For the visual difference approach, only a few improvements are possible. There could be a user defined threshold for tolerable visual differences on the screen, reducing the size of the difference list. When the camera is not moving any longer, the omitted differences could be sent to the display node since no other work has to be done anyway. Also, the problem of overlapping objects and incorrect results due to the identification of differing portions on the screen still exists. In order to solve this, the render nodes have to know more about the scene, particularly of objects passing through them. If the display node keeps a record about the depths of the fragments sent by the render nodes, he could identify possible bogus screen positions and request an update from the render nodes involved.

The image quality of the depth-limited method could be improved by using different amount of depth layers per object. An object which is only slightly transparent obviously does not need many depth layers before fragments lying behind the foremost fragments do not contribute to the final image.

Remote rendering of transparent objects remains a nontrivial task. Especially the need to intermediately store the fragments of the per-pixel linked list is a major problem, which justifies the use of other methods like depth-peeling if memory is only sparsely available on the system.

A. Appendix

A.1. GLSL Shaders

A.1.1. Color Fragment Creation

```
// ------------------------------- VERTEX SHADER ------------------------------
#version 430 core

// Interpolated vertex data
layout(location = 0) in vec3 vertexPosition;
layout(location = 1) in vec4 inColor;
// Add additional inputs if necessary (like normals, uv-coordinates...)

out vec4 fColor;

// Model-View-Projection matrix
uniform mat4 MVP;
// Model matrix
uniform mat4 M;
// View matrix
uniform mat4 V;

void main()
{
  gl_Position = MVP * vec4(vertexPosition, 1);
  fColor = inColor;
}

// ------------------------------ FRAGMENT SHADER -----------------------------
#version 430 core

#extension ARB_shader_image_load_store : require

// Atomic counter
layout (binding = 0, offset = 0) uniform atomic_uint ac;

// Header (2D)
layout (std430, binding = 0) buffer Header
{
  uint header[];
};
struct LinkedListEntry
{
  vec4 color;
  uint depth;
```

```
  uint next;
  uint dummy;
};

// Linked List
layout (std430, binding = 1) buffer LinkedList
{
  //uvec4 linkedList[];
  LinkedListEntry linkedList[];
};

// If this is not set, gl_FragCoord will contain (0.5, 0.5) values
layout (pixel_center_integer) in vec3 gl_FragCoord;

out vec4 gl_FragColor;

uniform uvec2 windowSize;
uniform int maxDepthLayers = 15;

// Color from vertex shader
in vec4 fColor;

void main(void)
{
  int id = int(int(gl_FragCoord.x + gl_FragCoord.y * windowSize.x));

  // Compute the fragment color and depth
  // ...

  // Get new index for linked list
  uint index = atomicCounterIncrement(ac);

  // Copy old header from header and set the new one
  uint oldHeader = atomicExchange(header[int(id)], index);

  LinkedListEntry item;
  item.next = oldHeader;
  item.color = fragColor;
  item.depth = floatBitsToUint(gl_FragCoord.z);

  linkedList[int(index)] = item;

  // Writing fragment is mandatory
  gl_FragColor = vec4(0.0f, 0.0f, 0.0f, 0.0f);
}
```

A.1.2. Normal Fragment Creation

```
// ------------------------------ VERTEX SHADER -----------------------------
#version 430 core

// Interpolated vertex data
layout(location = 0) in vec3 positionModelSpace;
layout(location = 1) in vec4 vertexColor;
layout(location = 2) in vec2 vertexUV;
layout(location = 3) in vec3 vertexNormal;

// Output data; will be interpolated for each fragment.
// Color from vertex shader
out vec4 fragmentColor;

out vec3 positionWorldSpace;
// Normals
out vec3 normalModelSpace;

// Values that stay constant for the whole mesh.
uniform mat4 MVP;
uniform mat4 M;

void main()
{
  // Standard vertex transformation
  gl_Position = MVP * vec4(positionModelSpace, 1);

  // Calculate position in world space
  positionWorldSpace = (M * vec4(positionModelSpace, 1)).xyz;

  // Normal in model space
  normalModelSpace = normalize(vertexNormal);

  fragmentColor = vertexColor;
}

// ----------------------------- FRAGMENT SHADER ----------------------------
#version 430 core

#extension GL_ARB_shader_image_load_store : require

// Atomic counter
layout (binding = 0, offset = 0) uniform atomic_uint ac;

struct LinkedListEntry
{
  uint flag;
  uint normal;
  uint x;
  uint y;
  uint depth;
  uint next;
  uint dummy1;
  uint dummy2;
```

```
};

// Header (2D)
layout (std430, binding = 0) buffer Header
{
  uint header[];
};

// Linked List
layout (std430, binding = 1) buffer LinkedList
{
  LinkedListEntry linkedList[];
};

// If this is not set, gl_FragCoord will contain (0.5, 0.5) values
layout (pixel_center_integer) in vec3 gl_FragCoord;

out vec4 gl_FragColor;

uniform uvec2 windowSize;
uniform int maxDepthLayers = 15;
uniform mat4 V;

in vec4 fragmentColor;

in vec3 positionWorldSpace;
// Normals
in vec3 normalModelSpace;

#define NORMAL_EPSILON 0.01f
#define NORMAL_UP 0x8FFFFFFF
#define NORMAL_DOWN 0xFFFFFFFF

// Returns a normal packed into an uint, saved in spherical coordinates (phi, theta)
uint packNormal(vec3 normal)
{
  if(abs(normal.x) < NORMAL_EPSILON && abs(normal.y) < NORMAL_EPSILON)
  {
    if(sign(normal.z) > 0)
    {
      return NORMAL_UP;
    }
    else
    {
      return NORMAL_DOWN;
    }
  }
  else
  {
    // Theta, Phi
    vec2 angles = vec2(atan(normal.y , normal.x), acos (normal.z));
    return packHalf2x16(angles);
  }
}
```

```
// Unpacks a previously packed normal
vec3 unpackNormal(uint normal)
{
  if(normal == NORMAL_UP)
  {
    return vec3(0, 0, 1);
  }
  else if (normal == NORMAL_DOWN)
  {
    return vec3(0, 0, -1);
  }
  else
  {
    vec2 tempNormal = unpackHalf2x16(normal);
    float sinPhi = sin(tempNormal.y);

    return vec3(sinPhi * cos(tempNormal.x),
          sinPhi * sin(tempNormal.x),
          cos(tempNormal.y));
  }
}

void main(void)
{
  int id = int(gl_FragCoord.x + gl_FragCoord.y * windowSize.x);

  // Get new index for linked list
  uint index = atomicCounterIncrement(ac);

  // Copy old header from header and set the new one
  uint oldHeader = atomicExchange(header[int(id)], index);

  vec4 positionCameraSpace = (V * vec4(positionWorldSpace, 1));

  LinkedListEntry item;
  item.next = oldHeader;
  item.flag = 0;
  item.normal = packNormal(normalModelSpace);
  item.x = floatBitsToUint(positionCameraSpace.x);
  item.y = floatBitsToUint(positionCameraSpace.y);
  item.depth = floatBitsToUint(positionCameraSpace.z);

  linkedList[int(index)] = item;

  gl_FragColor = vec4(0.0f, 0.0f, 0.0f, 0.0f);
}
```

A.2. Measurements

Model	Depth-limited					
	1024 × 768		1280 × 720		1600 × 900	
	Uncompressed	Compressed	Uncompressed	Compressed	Uncompressed	Compressed
Bunny	3660	2569 (70.2%)	3217	2057 (63.9%)	5027	3203 (63.7%)
Dragon	3745	2503 (66.8%)	3291	2214 (67.3%)	5143	3398 (66.1%)
Buddha	3441	2292 (66.6%)	3025	2002 (66.2%)	4726	3099 (65.6%)

Model	Difference					
	1024 × 768		1280 × 720		1600 × 900	
	Uncompressed	Compressed	Uncompressed	Compressed	Uncompressed	Compressed
Bunny	7318	2096 (28.6%)	6432	1649 (25.6%)	10049	2935 (29.2%)
Dragon	7039	1859 (29.5%)	6187	1594 (29.5%)	9667	2702 (29.0%)
Buddha	7663	2264 (26.4%)	6735	1988 (25.8%)	10523	3051 (28.0%)

Model	Bilateral					
	1024 × 768		1280 × 720		1600 × 900	
	Uncompressed	Compressed	Uncompressed	Compressed	Uncompressed	Compressed
Bunny	7328	2115 (28.9%)	6440	2598 (40.3%)	10063	2799 (27.8%)
Dragon	7849	2327 (29.6%)	6897	2792 (40.5%)	10777	3038 (28.2%)
Buddha	7190	2202 (30.6%)	6319	2635 (41.7%)	9874	2875 (29.1%)

Table A.1.: Comparison of uncompressed and LZ4 compressed sizes (in KiB) for different resolutions

Model	Depth-limited						
	1024 × 768	1280 × 720	1600 × 900		1024 × 768	1280 × 720	1600 × 900
Bunny	52	145	63		5.8	5.7	3.8
Dragon	60	98	63		4.5	5.1	4
Buddha	110	111	82		5.6	5.7	4.6

Model	Difference						
	1024 × 768	1280 × 720	1600 × 900		1024 × 768	1280 × 720	1600 × 900
Bunny	69	88	96		5	4.8	3.7
Dragon	111	68	135		4.3	5	3.2
Buddha	95	85	119		4.5	4.8	3.6

Model	Bilateral						
	1024 × 768	1280 × 720	1600 × 900		1024 × 768	1280 × 720	1600 × 900
Bunny	111	196	145		2.1	1.9	1.7
Dragon	145	105	151		1.9	2.2	1.5
Buddha	112	118	131		2.1	2.3	1.7

Table A.2.: Average time required for sending the fragments to the display node in milliseconds over a gigabit lan connection (left) and actual frame rate (right) for different resolutions and methods

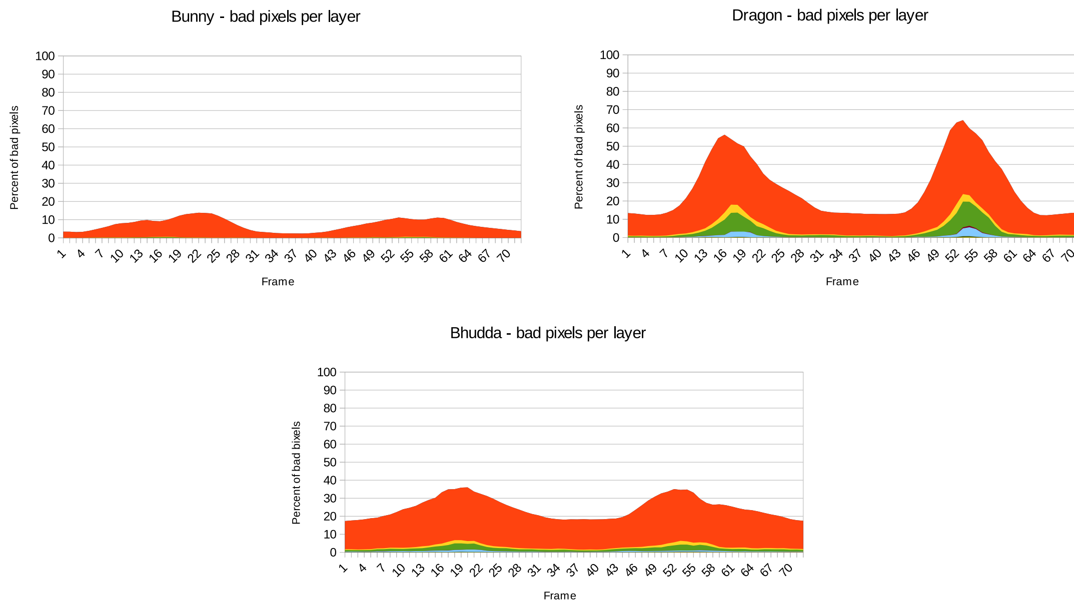

Figure A.1.: Percent of bad pixels with increasing number of depth layers for depth-limited rendering

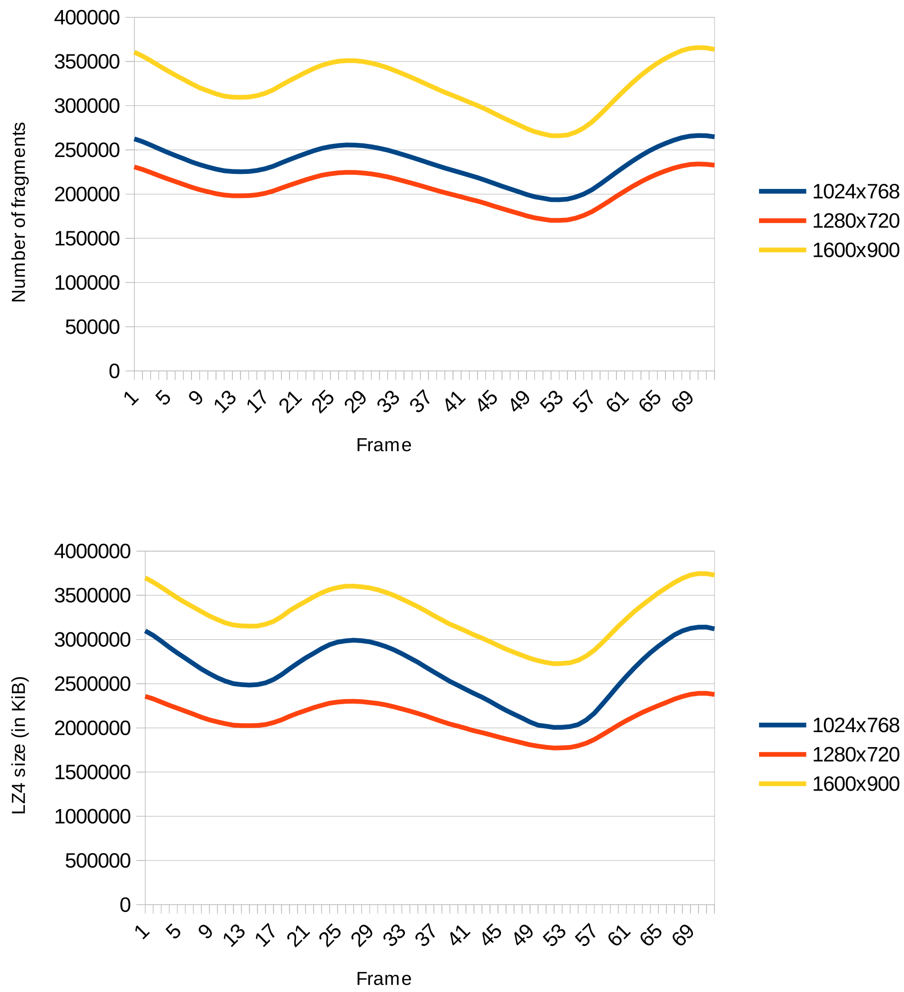

Figure A.2.: Network traffic using depth-limited rendering for the bunny model with different resolutions

Figure A.3.: Network traffic using visual difference rendering for the bunny model with different resolutions

Figure A.4.: Network traffic using bilateral rendering for the bunny model with different resolutions

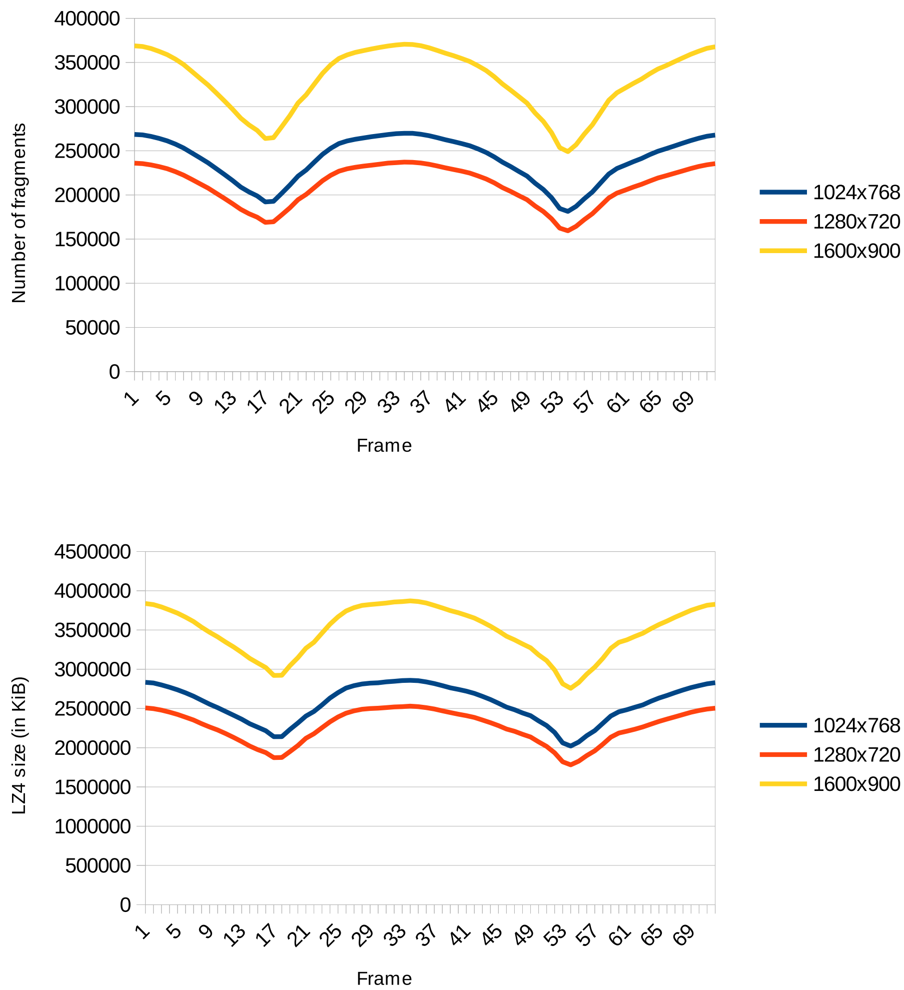

Figure A.5.: Network traffic using depth-limited rendering for the dragon model with different resolutions

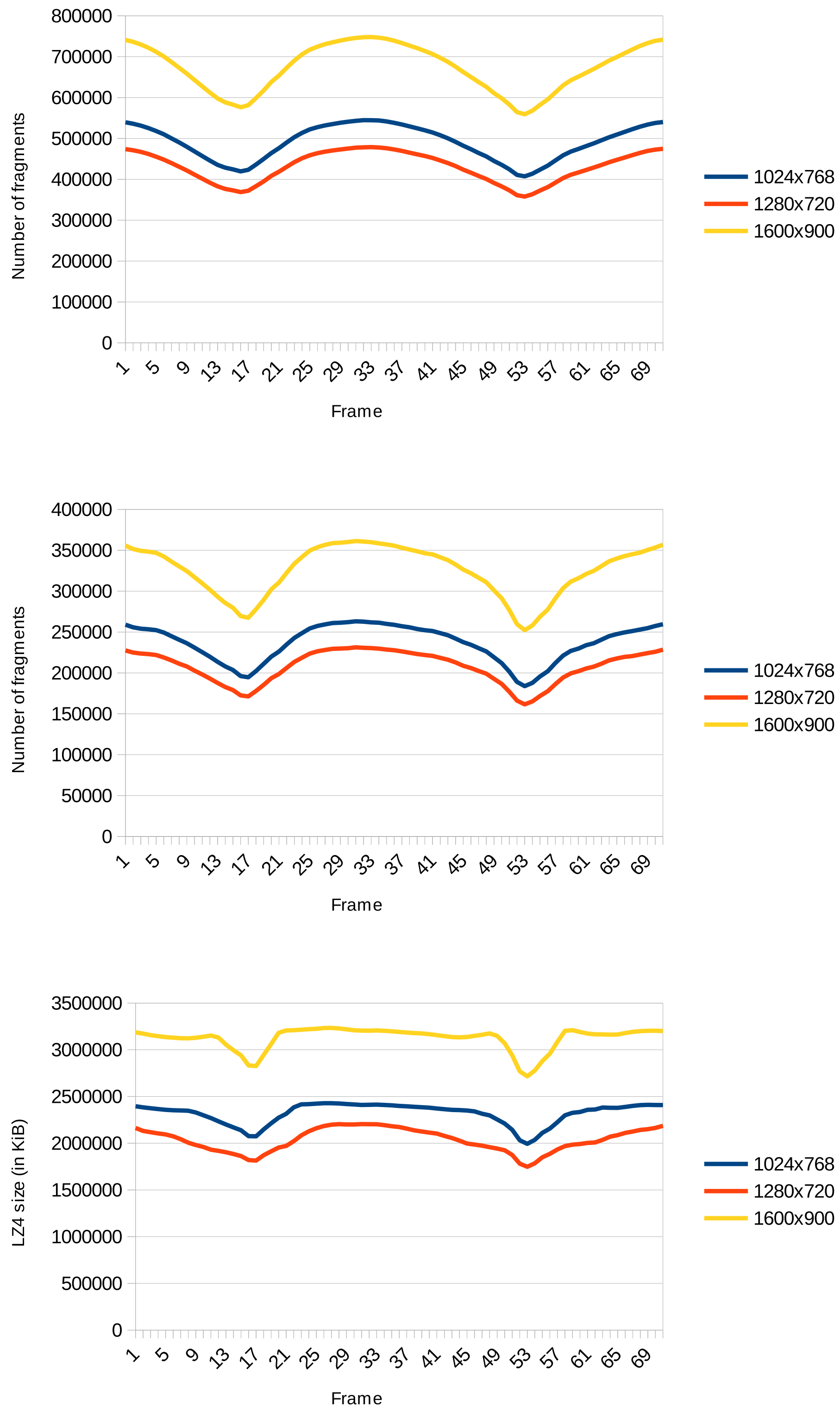

Figure A.6.: Network traffic using visual difference rendering for the dragon model with different resolutions

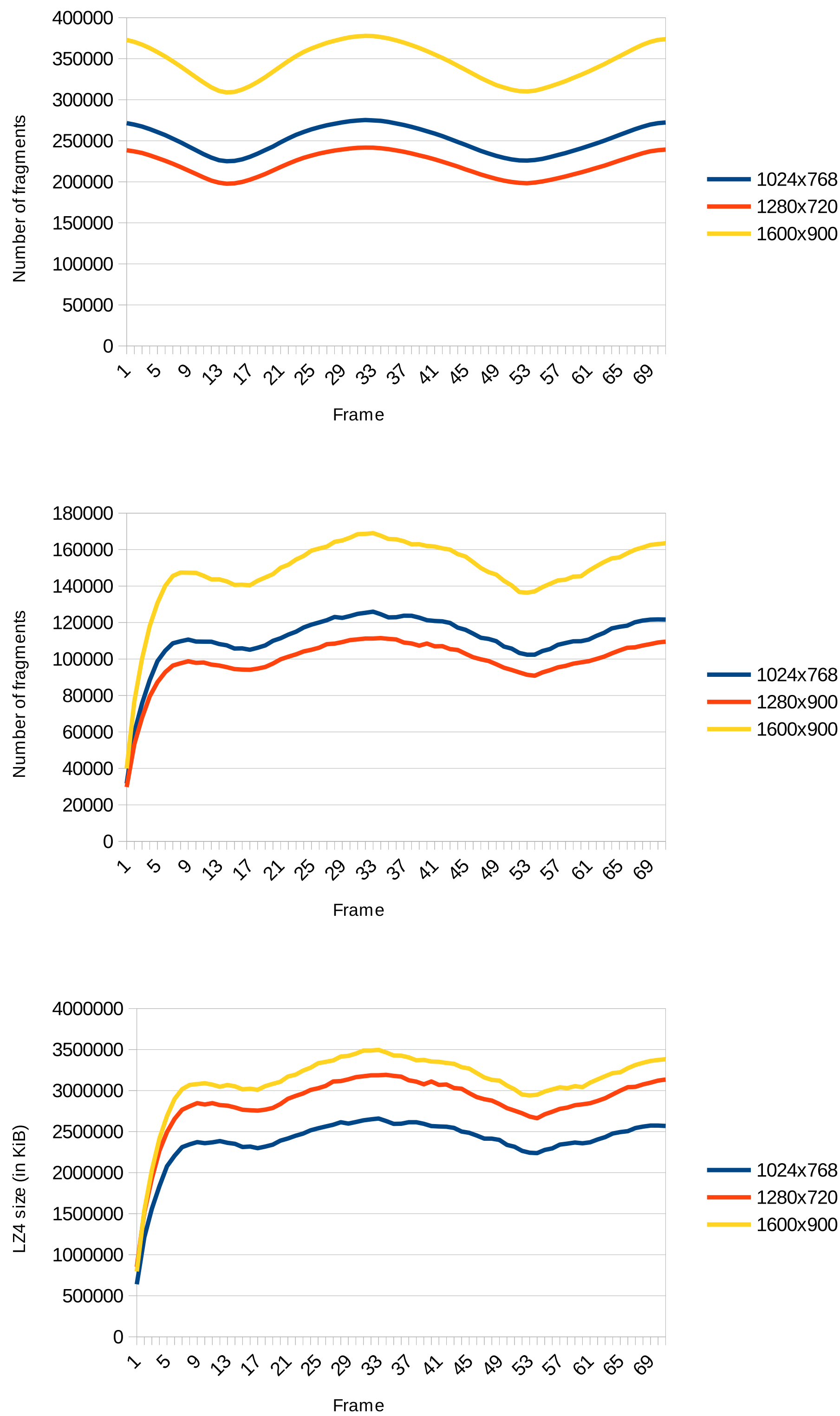

Figure A.7.: Network traffic using bilateral rendering for the dragon model with different resolutions

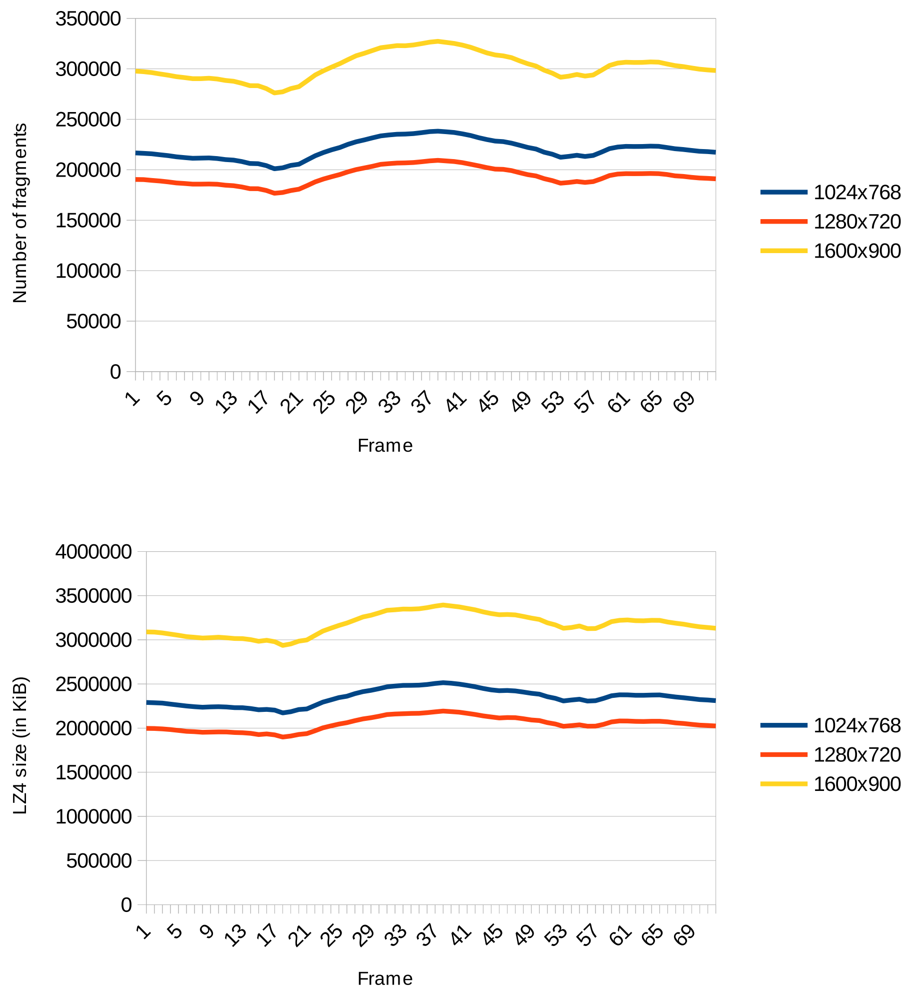

Figure A.8.: Network traffic using depth-limited rendering for the buddha model with different resolutions

Figure A.9.: Network traffic using visual difference rendering for the buddha model with different resolutions

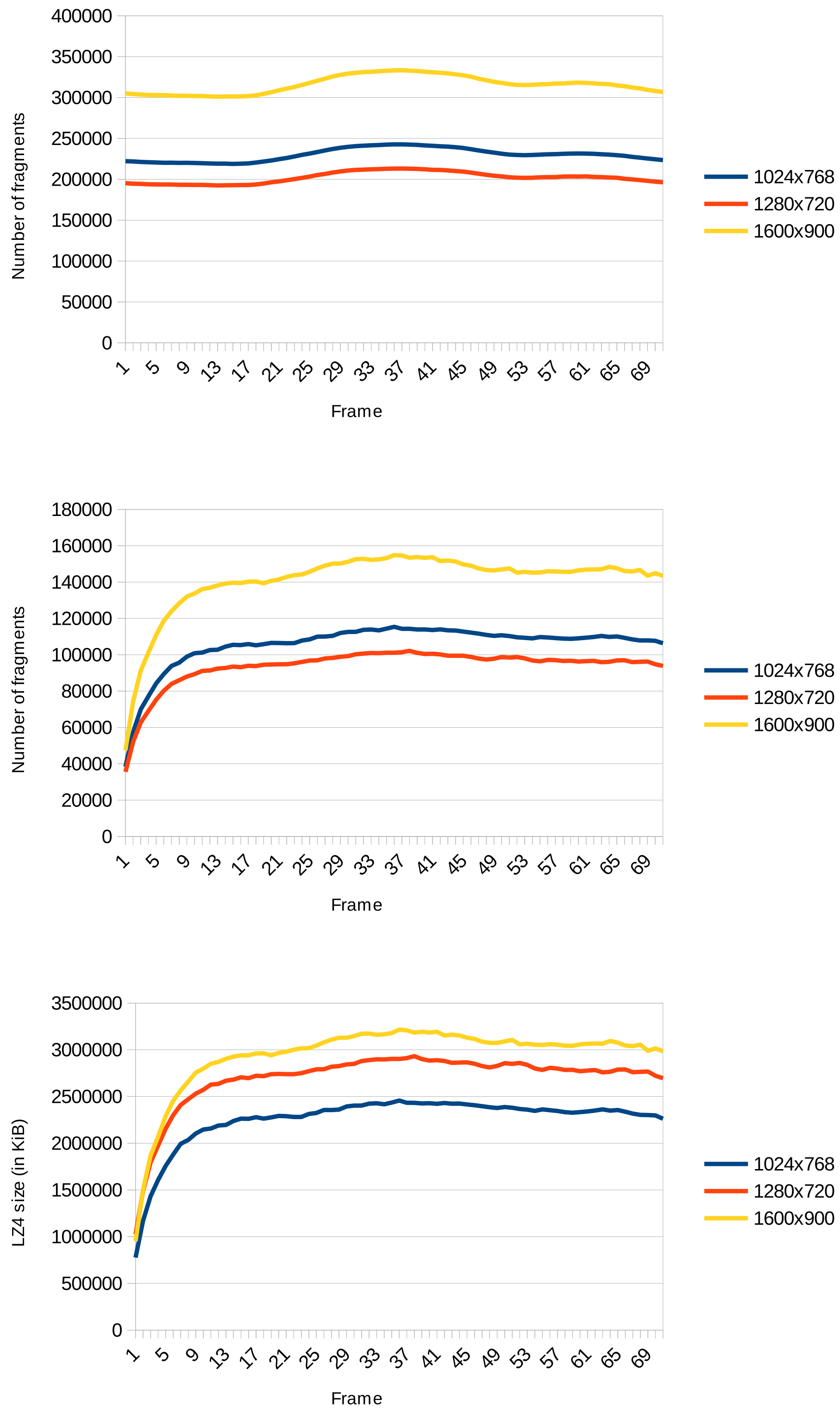

Figure A.10.: Network traffic using bilateral rendering for the buddha model with different resolutions

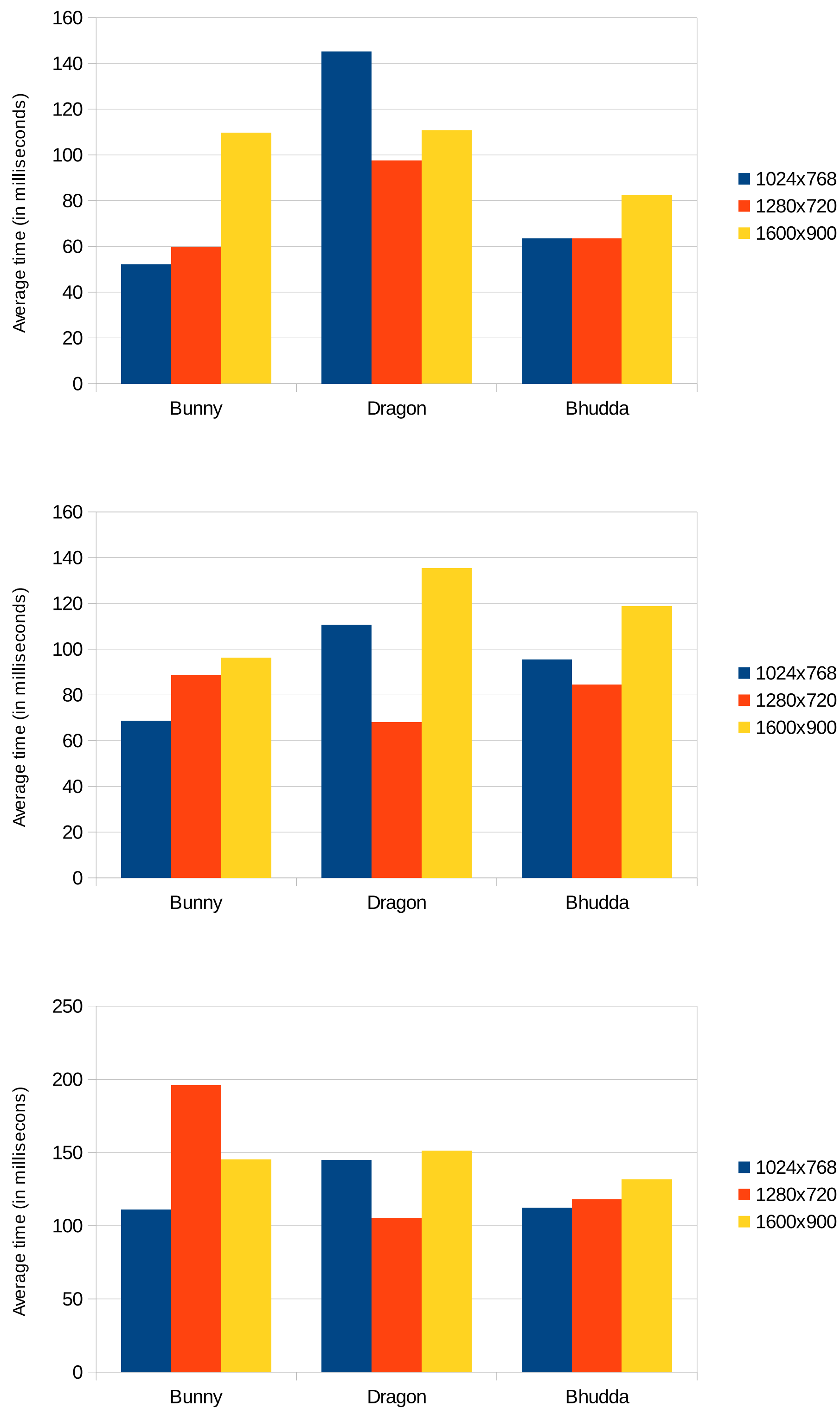

Figure A.11.: Average network times in milliseconds for the three methods top: depth-limited, middle: difference, bottom: bilateral

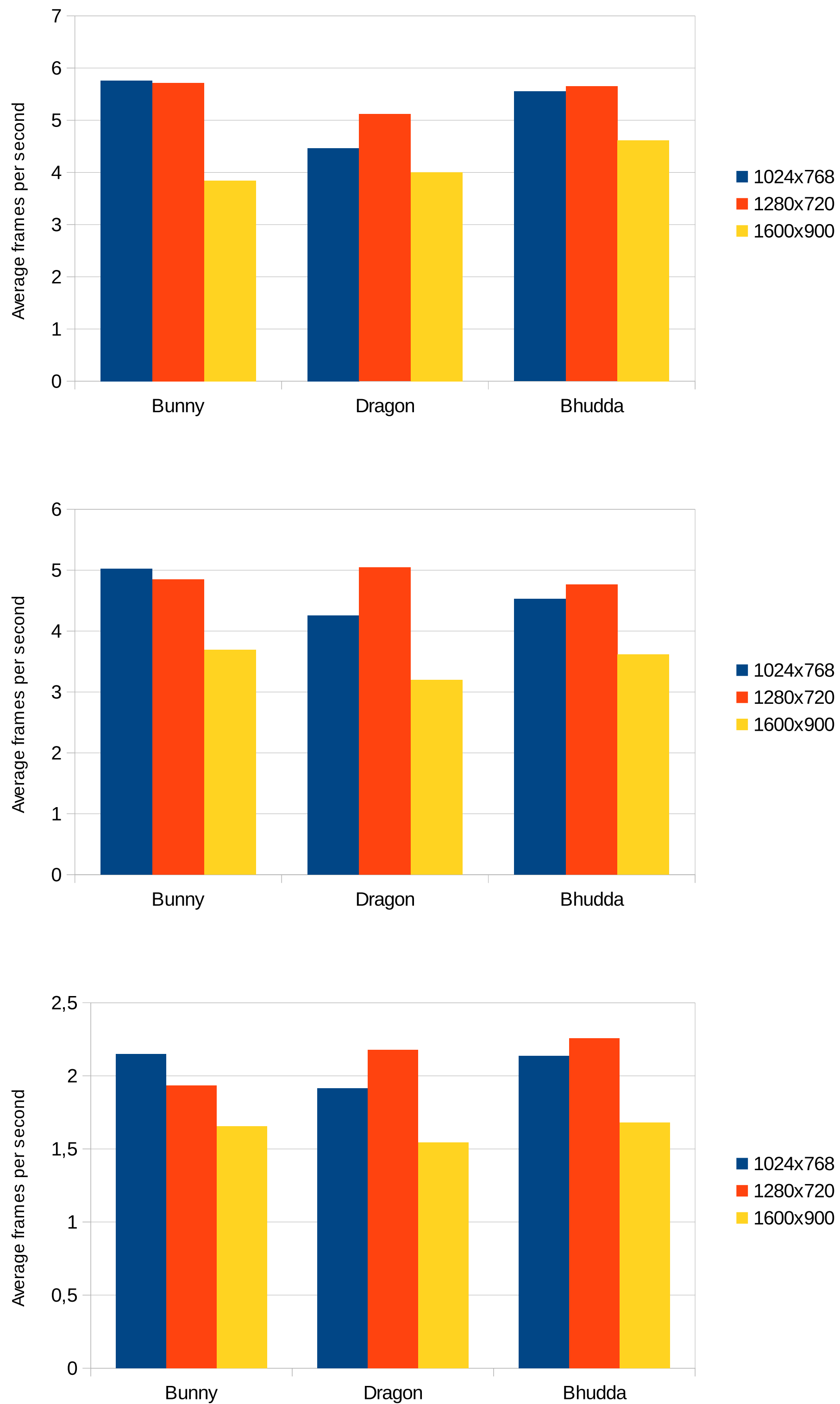

Figure A.12.: Average frames per second for the three methods
top: depth-limited, middle: difference, bottom: bilateral

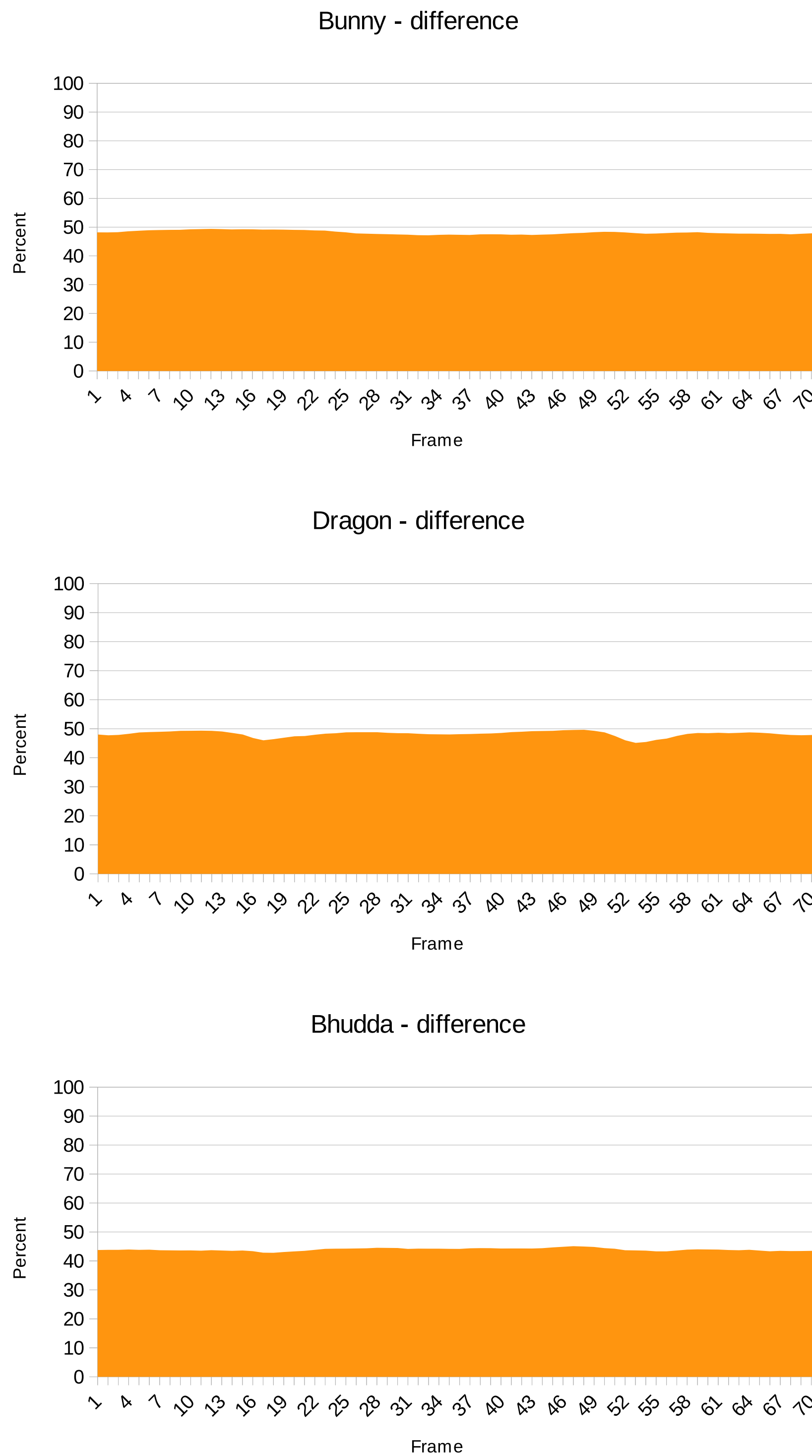

Figure A.13.: Percentaged number of difference fragments compared to the original number of fragments for the visual difference method
top: bunny, middle: dragon, bottom: buddha

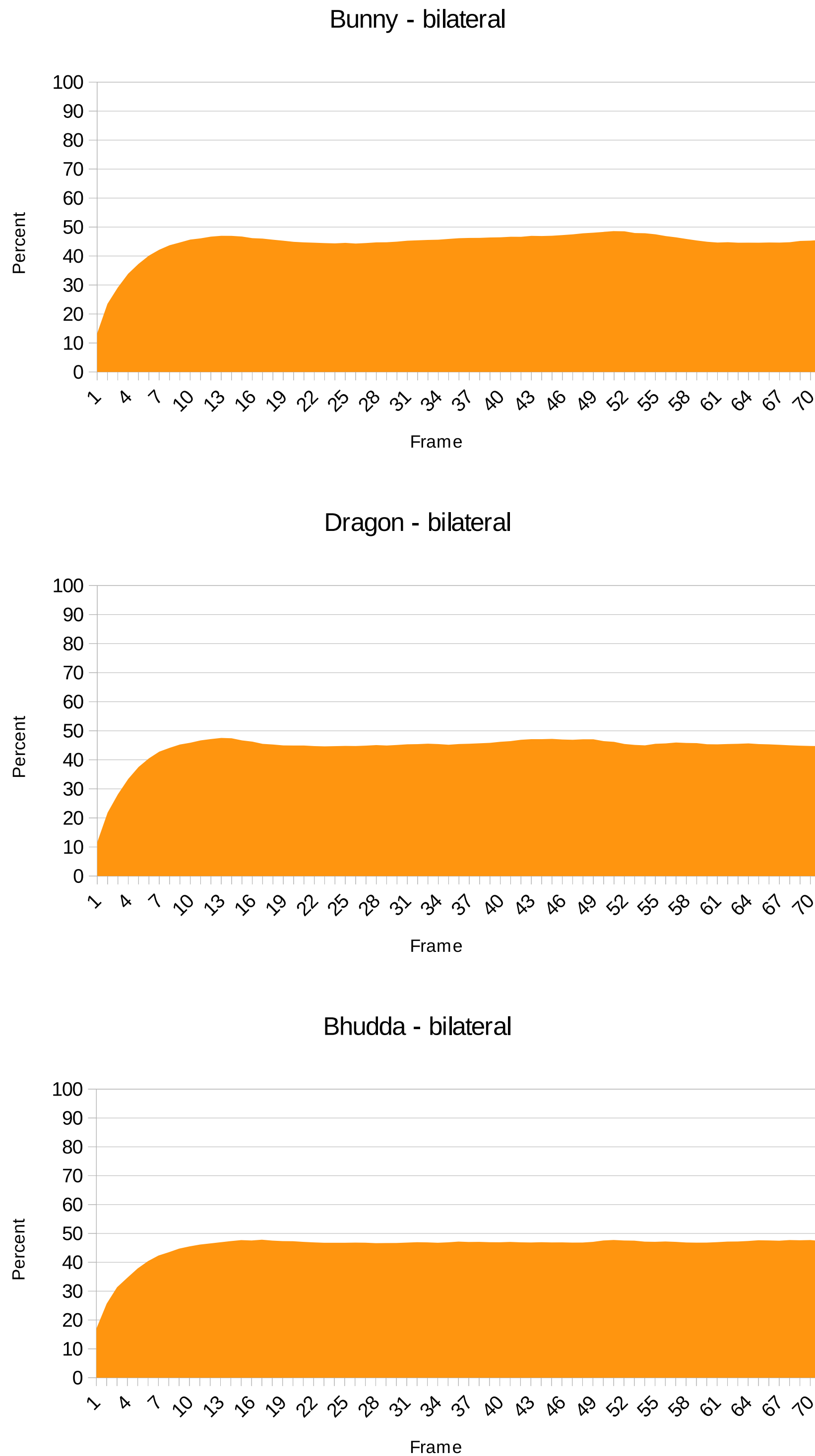

Figure A.14.: Percentaged number of difference fragments compared to the original number of fragments for the bilateral method
top: bunny, middle: dragon, bottom: buddha

A.3. Algorithms

A.3.1. Fragment Sorting

Algorithm A.1 Sorting of fragments

Require:
- Header
- Linked List
- Maximum number of fragments stored in a list
- Screen position of fragment
- Window size

```
function SortFragments(headerBuffer, linkedListBuffer, maxFragments)
    id ← screenPos.x + screenPos.y · windowWidth
    currentIndex ← headerBuffer[id]
    fragmentBuffer[maxFragments]
    numFragments ← 0
    while currentIndex ≠ 0 do                                    // Cache fragments
        fragment ← linkedListBuffer[currentIndex]
        fragmentBuffer[numFragments] ← fragment
        currentIndex ← fragment.next
        ++numFragments
    end while
    for i ← 0 to numFragments − 1 do                             // Sort fragments
        for j ← i + 1 to numFragments do
            if fragmentBuffer[i].depth < fragmentBuffer[j].depth then
                tempFragment ← fragmentBuffer[i]
                fragmentBuffer[i] ← fragmentBuffer[j]
                fragmentBuffer[j] ← tempFragment
            end if
        end for
    end for
end function
```

A.3.2. Depth-limited Rendering

Algorithm A.2 Depth-limited rendering remote node

Require:

- Model to render
- Reference to display node
- Flag wether fragments should be merged into single fragment

```
function RenderNodeDepthLimited(model, displayNode, preBlend)
    camera ← displayNode.getCamera()
    createFragments(model, camera)
    if preBlend then
        blendFragmentsIntoSingleFragment()
    end if
    sendHeader(displayNode)
    sendLinkedList(displayNode)
end function
```

Algorithm A.3 Depth-limited rendering display node

Require:

- References to render nodes

```
function DisplayNodeDepthLimited(renderNodes)
    for all node ← renderNodes do
        sendCamera(camera, node)
    end for
    clearHeader(displayHeader)
    for all node ← renderNodes do
        tempHeader ← readHeader(node)
        tempLinkedList ← readLinkedList(node)
        blendFragments(displayHeader, displayLinkedList, tempHeader, tempLinkedList)
    end for
    render(displayHeader, displayLinkedList)
end function
```

A.3.3. Visual Difference Rendering

Algorithm A.4 Visual difference rendering remote node

Require:
- Model to render
- Reference to display node
- Flag which indicates the current texture buffer (either 0 or 1)

function RenderNodeDifference(*model*, *displayNode*, *textureIndex*)
  $camera \leftarrow displayNode.getCamera()$
  $createFragments(model, camera)$
  $blendIntoTextureBuffer(textureIndex)$
  if $firstFrame$ **then**
    $sendHeader(displayNode)$
    $sendLinkedList(displayNode)$
  else
    $createDifference(textureIndex)$
    $sendDifferenceHeader(displayNode)$
    $sendDifferenceLinkedList(displayNode)$
  end if
end function

Algorithm A.5 Visual difference rendering display node

Require:
- References to render nodes

function DisplayNodeDifference(*renderNodes*)
  for all $node \leftarrow renderNodes$ **do**
    $sendCamera(camera, node)$
  end for
  for all $node \leftarrow renderNodes$ **do**
    $differenceHeader \leftarrow readDifferenceHeader(node)$
    $differenceLinkedList \leftarrow readDifferenceLinkedList(node)$
    $mergeDifference(displayHeader, displayLinkedList,$
      $differenceHeader, differenceLinkedList)$
  end for
  $render(displayHeader, displayLinkedList)$
end function

A.3.4. Bilateral Rendering

Algorithm A.6 Bilateral rendering remote node

Require:

- Model to render
- Reference to display node

function RENDERNODEBILATERAL(*model*, *displayNode*)
  $camera \leftarrow displayNode.getCamera()$
  $createFragments(model, camera)$
  if $firstFrame$ **then**
    $sendHeader(displayNode)$
    $sendLinkedList(displayNode)$
    $copyHeaderToTransformationHeader()$
    $copyLinkedListToTransformationList()$
  else
    $transformFragments(transformationMatrices,$
      $previousTransformationMatrices,$
      $transformedHeader, transformedLinkedList)$
    $constructDifferenceListFromTransformedFragments()$
    $sendDifferenceHeader(displayNode)$
    $sendDifferenceLinkedList(displayNode)$
    $copyDifferenceListToTransformationList()$
  end if
  $previousTransformationMatrices \leftarrow inverse(transformationMatrices)$
end function

Algorithm A.7 Bilateral rendering display node

```
Require:
  • References to render nodes
function DisplayNodeBilateral(renderNodes)
    for all node ← renderNodes do
        sendCamera(camera, node)
    end for
    for all node ← renderNodes do
        if firstFrame then
            displayHeader ← readHeader(node)
            displayLinkedList ← readLinkedList(node)
            copyListToTransformationList(node.getID())
            render(displayHeader, displayLinkedList)
        else
            transformFragments(transformationMatrices,
                               previousTransformationMatrices,
                               transformedHeader, transformedLinkedList)
            differenceHeader ← readDifferenceHeader(node)
            differenceLinkedList ← readDifferenceLinkedList(node)
            mergeDifference(transformedHeader, transformedLinkedList,
                            differenceHeader, differenceLinkedList)
            render(transformedHeader, transformedLinkedList)
        end if
    end for
    previousTransformationMatrices ← inverse(transformationMatrices)
end function
```

A.4. Notation

DWORD		Double Word, usually equivalent to 4 byte
VRAM		Video Random Access Memory, usually residing on the graphics card or as dedicated memory
KiB		Kibibyte. Equivalent to 1024 Byte
MiB		Mibibyte. Equivalent to 1024 KiB
GiB		Gibibyte. Equivalent to 1024 MiB
KB		Kilobyte. Equivalent to 1000 Byte
MB		Megabyte. Equivalent to 1000 KB
GB		Gigabyte. Equivalent to 1000 MB
$\vec{\mathbf{p}}_o$	$\in \mathbb{R}^4$	Position in object space
$\vec{\mathbf{p}}_w$	$\in \mathbb{R}^4$	Position in world space
$\vec{\mathbf{p}}_e$	$\in \mathbb{R}^4$	Position in eye space
$\vec{\mathbf{p}}_c$	$\in \mathbb{R}^4$	Position in clipping space
$\vec{\mathbf{p}}_{ndc}$	$\in [-1;1]^3 \times \{1\}$	Position in normalized device space
$\vec{\mathbf{p}}_s$	$\in \mathbb{R}^4$	Position in screen space
$\mathbf{M}$	$\in \mathbb{R}^4 \times \mathbb{R}^4$	Model matrix
$\mathbf{V}$	$\in \mathbb{R}^4 \times \mathbb{R}^4$	View matrix
$\mathbf{P}$	$\in \mathbb{R}^4 \times \mathbb{R}^4$	Projection matrix

Bibliography

[BKKB] F. Bauer, M. Knuth, A. Kujiper, J. Bender. (Cited on page 14)

[Car84] L. Carpenter. The A -buffer, an Antialiased Hidden Surface Method. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, pp. 103–108. ACM, New York, NY, USA, 1984. doi:10.1145/800031.808585. URL http://doi.acm.org/10.1145/800031.808585. (Cited on page 13)

[CBPZ04] L. Cheng, A. Bhushan, R. Pajarola, M. E. Zarki. Real-time 3D Graphics Streaming using MPEG-4. In *In Proc. IEEE/ACM Wksp. on Broadband Wireless Services and Appl*. 2004. (Cited on page 14)

[CSN+12] G. Chen, P. V. Sander, D. Nehab, L. Yang, L. Hu. Depth-presorted Triangle Lists. *ACM Trans. Graph.*, 31(6):160:1–160:9, 2012. doi:10.1145/2366145.2366179. URL http://doi.acm.org/10.1145/2366145.2366179. (Cited on page 11)

[Eve01] C. Everitt. Interactive Order-Independent Transparency, 2001. (Cited on page 13)

[FDFH90] J. D. Foley, A. van Dam, S. K. Feiner, J. F. Hughes. *Computer graphics: principles and practice (2nd ed.)*. Addison-Wesley Longman, 1990. (Cited on page 11)

[GP07] M. Gross, H. Pfister. *Point-Based Graphics*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. (Cited on page 17)

[KKP+13] D. Kauker, M. Krone, A. Panagiotidis, G. Reina, T. Ertl. Evaluation of per-pixel linked lists for distributed rendering and comparative analysis. *Computing and Visualization in Science*, 15(3):111–121, 2013. doi:10.1007/s00791-013-0203-6. URL http://doi.acm.org/10.1007/s00791-013-0203-6. (Cited on page 14)

[KWBG13] E. Kerzner, C. Wyman, L. Butler, C. Gribble. Toward Efficient and Accurate Order-independent Transparency. In *ACM SIGGRAPH 2013 Posters*, SIGGRAPH '13, pp. 109:1–109:1. ACM, New York, NY, USA, 2013. doi:10.1145/2503385.2503504. URL http://doi.acm.org/10.1145/2503385.2503504. (Cited on page 14)

[LWX06] B. Liu, L. yi Wei, Y. qing Xu. Multi-layer depth peeling via fragment sort. Technical report, 2006. (Cited on page 13)

[PHE+11] D. Pajak, R. Herzog, E. Eisemann, K. Myszkowski, H. peter Seidel. Scalable Remote Rendering with Depth and Motion-flow Augmented Streaming, 2011. (Cited on page 14)

[SML11] M. Salvi, J. Montgomery, A. Lefohn. Adaptive Transparency. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pp. 119–126. ACM, New York, NY, USA, 2011. doi:10.1145/2018323.2018342. URL http://doi.acm.org/10.1145/2018323.2018342. (Cited on page 14)

[Wik14] Wikipedia.de@ONLINE, 2014. URL http://en.wikipedia.org/wiki/Spherical_coordinates/. (Cited on page 18)

[YHGT10] J. C. Yang, J. Hensley, H. Grün, N. Thibieroz. Real-time Concurrent Linked List Construction on the GPU. In *Proceedings of the 21st Eurographics Conference on Rendering*, EGSR'10, pp. 1297–1304. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2010. doi:10.1111/j.1467-8659.2010.01725.x. URL http://dx.doi.org/10.1111/j.1467-8659.2010.01725.x. (Cited on pages 12, 13, 15 and 16)

All links were last followed on March 10, 2014.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Ort, Datum, Unterschift