Cascaded Deferred Rendering

Marcus Faleij, Alexander Ivannikov
This thesis is submitted to the School of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Bachelor in Digital Game Development and Computer Science. The thesis is equivalent to 10 weeks of full time studies.
Contact Information:
Author(s):
Marcus Faleij
E-mail: faleij@live.se

Alexander Ivannikov
E-mail: Alexivan91@gmail.com

University advisor(s):
Dr. Veronica Sundstedt
School of Computing

School of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona
Sweden

Internet : www.bth.se/com
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57
Abstract

A long-standing difficulty with rendering huge distances is depth-fighting; a visual artefact produced when two or more fragments overlap either due to coplanar geometry or insufficient depth precision. This thesis presents two novel methods, Cascaded Deferred Rendering (CDR) and Logarithmic Cascaded Deferred Rendering (LogCDR), as a solution to solve depth-fighting that is due to insufficient depth precision. This thesis also evaluates an existing method, logarithmic depth buffer, comparing it against the standard depth buffer in OpenGL, CDR and LogCDR. The most prominent solution found was logarithmic depth buffer because of performance, no overhead from frustum division and extensive culling, ease of implementation and conveniences such as easier implementation of transparency.

**Keywords:** Cascaded, Multifrustum, Logarithmic, Deferred Rendering.
References 23
A Appendix 25
Chapter 1

Introduction

Rendering vast scenes with faraway geometry without visual artefacts is proving to be a challenge even for modern game engines as it presents many problems; one of which is depth precision error, also more commonly known as depth-fighting or Z-fighting. Depth-fighting occurs when there is not enough precision in the depth-buffer to differ between two or more fragment positions. When depth-fighting occurs a visual artefact of two or more fragments occurs as a result as seen in Figure 1.1, this artefact tends to flicker as the camera moves.

\[ Figure \ 1.1: \] Depth-fighting between two coplanar planes.

The issue of depth-fighting has not been well researched and thus there are only a few solutions to minimizing the effects [6, 4]. These solutions have also been applied to shadow mapping where similar problems, that stems from the loss of precision in the depth-buffer, occurs [7]. The most basic method for minimizing depth-fighting is moving the near clip plane as far away as possible and likewise moving the far plane as close as possible. However this is not adequate
Chapter 1. Introduction

as games are progressing towards larger environments that requires the user to see as far as the horizon and beyond. The more advanced methods for solving or minimizing depth-fighting will be discussed in-depth in Chapter 2 and 3.

This thesis presents one approach, Cascaded Deferred Rendering (CDR), to solve depth-fighting for virtually infinite rendering distances. CDR is at the core the multifrustum rendering technique presented by Cozzi and Ring but CDR extends the multifrustum technique by using deferred rendering [6]. Cozzi and Ring do not present any implementation details or results from their implementation so this thesis discusses different methods of implementing CDR and presents results from one implementation.

The rest of the thesis is organised as follows: In Chapter 2 and 3, logarithmic depth buffer and respectively CDR is explained in depth. In Chapter 5 the implementation is described. In Chapter 6, the results are presented. In Chapter 8, conclusions of the work presented in this thesis and ideas for future research are presented.

1.1 Background

Figure 1.2: Depth-fighting apparent on a building in EA DICE’s ”Battlefield 3” when viewing through a high-powered scope or binoculars the field of view is lowered to simulate zooming and depth-fighting becomes apparent on distant geometry as shown in Figure 1.2 and 1.3. Another modern game that suffers from depth-fighting issues is Bethesda Softworks LLC’s ”The Elder Scrolls
Chapter 1. Introduction

V: Skyrim”, where the depth-fighting is visible on the mountains because of a layered geometry structure. Other games that suffers from depth-fighting issues are for example Rockstar’s “Grand Theft Auto: IV”, EA Redwood Shores “Dead Space”.

1.1.1 Standard Depth Buffer

On a standard perspective depth-buffer the precision is non-linearly degrading, starting from the near clipping plane and ending at the far clipping plane. The non-linear degradation results in high precision for near objects and low precision for distant objects.

In OpenGL, the depth-buffer values are calculated as defined in Equation 1.1 [2]. The Z is proportional to the reciprocal of the depth-buffer value resulting in great depth precision closest to the near clip plane and lesser precision towards the far clip plane.

\[
\text{depth buffer value} = (1 << N) \times \left( a + \frac{b}{z} \right) \quad (1.1)
\]

Where:

- \( N \) = number of bits of \( Z \) precision
- \( z_{\text{Far}} \) = Far clip
- \( z_{\text{Near}} \) = Near clip
- \( a = \frac{z_{\text{Far}}}{z_{\text{Far}} - z_{\text{Near}}} \)
- \( b = \frac{(z_{\text{Far}} \times z_{\text{Near}})}{(z_{\text{Near}} - z_{\text{Far}})} \)
- \( z = \text{distance from the eye to the object} \)

The nonlinearity is part of the cause for depth-fighting, which occur when there is not enough precision - usually in the distance. Other factors that contribute to the depth buffers precision is field of view, window coordinate precision, error accumulated by single-precision projection, viewport and rasterisation arithmetics [1].

1.1.2 Frustum Culling

Frustum culling is the only culling technique used in the implementation. It’s function is to prevent objects outside of the camera’s view from being rendered thereby reducing the amount of objects to render. Frustum culling requires each object to have a bounding-volume with which is used to intersection test against the frustum. The negative aspect of this is that the intersection tests requires
CPU resources which potentially could become a bottleneck of the architecture. To improve performance of the implementation each cube in the implementation uses bounding-spheres to test intersection as sphere against frustum tests require minimal CPU power. The implementation of the frustum intersection against a sphere used can be found in appendix Listing A.1.

1.2 Purpose

When rendering distant geometry with OpenGL or DirectX it is by default limited by the depth buffers precision. Using a too high range causes depth-fighting which disturbs the players immersion resulting in an reduction of the visual experience. Enabling virtually infinite precision would solve depth-fighting and remove the immersion disturbance thereby providing a greater visual experience. Solving depth-fighting will allow the designers and artists to have more freedom by removing restrictions when designing games and making content [9]. The purpose of this study is to compare the standard depth buffer of OpenGL with CDR, LogCDR and Logarithmic Depth Buffer and to evaluate if these techniques are suitable for real-time applications.

1.3 Related Work

Many games bypass the problems of depth-fighting by using various techniques. Distance fog is often used to fade distant objects, which are usually the ones suffering from depth-fighting. Another technique used is level of detail to switch models to a more tolerant form [9]. These are the basic techniques but they do not actually solve the problem, instead they work around it.

The most prominent solution available today is a technique called logarithmic depth buffer [4]. Logarithmic depth buffer uses a logarithmic distribution of the Z-values so that the resolution near the camera is proportionally equal to that of one far away from the camera. Storing the depth value in a logarithmic fashion enables great depth precision even at hundreds of kilometers away from the camera. The drawbacks of using logarithmic depth buffer are either disabling of early-z optimisation [8] or having visual artefacts when thin geometry or huge polygons are behind the camera [3].

Cozzi and Ring present a frustum split method to maintain good depth-buffer precision allowing the rendering of virtually infinite distant scenes [6]. Cozzi and Ring also mention that their implementation has performance issues and visual artefacts between the frustums.
1.4 Research Questions

The first research question of this thesis is if CDR is possible, How well it would work compared to logarithmic depth buffers as it is the current prominent method of solving depth fighting.

The second question of this thesis is the combination of logarithmic and CDR which will be referred to logCDR from hence on and if the logCDR would be achieve even better depth quality.

The third and last question would be if the performance implications would be low enough to be used in real-time applications such as games as it would allow games to have greater view distances or better depth buffering at low distances.

1.5 Scope

The techniques presented in this thesis are not meant to solve coplanar depth fighting as in Figure 1.1. This thesis will limit its implementations to OpenGL 4.1. Implementation details for DirectX or any other rasterised rendering APIs will not be mentioned. Multithreaded CDR approaches will be discussed but not implemented.

All the implementations will include frustum culling. Advanced level of detail techniques in combination with CDR will not be implemented, but the possibilities will be briefly discussed.

Cozzi and Ring mention that visual artefacts appear in their implementation between the frustum splits on translucent geometry. However, translucency, transparency and opacity is beyond the scope of this thesis and will not be discussed [6].

1.6 Methods

To test the performance of CDR against a standard depth buffer and logarithmic depth buffer all three methods will be implemented in C++ using OpenGL 4.1. To simulate the worst case scenario; a low field of view will be used to simulate zooming in on distant geometry. The visual quality will be demonstrated with pictures and evaluated with a user case study, where human subjects performs a two-alternative forced-choise (2AFC) experiment [10], and by calculating the
smallest depth separation for each technique. The performance will be evaluated using time measurements.
Chapter 2

Logarithmic Depth Buffer

As opposed to the standard depth buffers’ non-linear way to store depth values, a logarithmic depth buffer stores the depth values in a linear fashion using a logarithmic function; Ulrich describes a way to have constant precision for the entire near to far range as seen in Equation 2.1 [11]. Storing depth values linearly results in equal precision along the whole view distance and ensures that objects far away have the same quality as objects closeby.

\[
\text{fragment depth} = \left( \frac{(2^K - 1) \times \log_{\frac{z_{\text{Near}}}{z_{\text{Far}}}} \text{fragment depth}}{2 \cdot \log_{\frac{z_{\text{Far}}}{z_{\text{Near}}}} \frac{z_{\text{Far}}}{z_{\text{Near}}}} \right)^2 \tag{2.1}
\]

Where:

- \( K \) = number of bits of Z precision
- \( z_{\text{Far}} \) = Far clip
- \( z_{\text{Near}} \) = Near clip

Using logarithmic depth buffer requires the modification of the depth buffer value in all shader programs. This can be done on a vertex or fragment level. The problem with doing it on the vertex shader is that the values get interpolated before reaching the fragment shader and may differ from the expected value and thus may produce visual artefacts when at least one vertex of a visible triangle lies behind the camera. This happens due to the non-linearity of the logarithmic function because of the implicit linear interpolation of the depth values in the rasteriser. The solution is to write to \( \text{gl\_FragDepth} \) in the fragment shader. However, writing to \( \text{gl\_FragDepth} \) in the fragment shader disables the hardware acceleration of early z-testing and may result in loss of performance.
Chapter 3

Cascaded Deferred Rendering

Figure 3.1: How the frustum split might occur. Frustum divisions not to scale.

Cascaded rendering is a method that takes advantage of parallel split frustums by rendering each frustum on its own, rendering can be done in parallel or one by one. CDR, much like Cascaded Shadow Mapping, uses this technique to bypass the inherent float precision limit on depth buffers by using a separate depth buffer for each frustum.

CDR uses the standard depth buffer available in OpenGL but can also be used with a logarithmic depth buffer to decrease the number of required frustum divisions and further increase the precision in the depth buffer.

When splitting the frustum using a 24 bit depth buffer the appropriate far/near ratio to use for the near-clip and far-clip planes is 1000 [6]. This ensures the same quality across the whole frustum. For example if near-clip is 0.1 the far-clip should be 100.

When rendering, the near-clip and far-clip planes do not clip the fragments precisely, thus it is necessary to have the frustums overlap slightly to avoid artefacts appearing at the frustum splits. This overlap might cause artefacts when rendering translucent or transparent geometry [6].
Rendering using multiple frustums requires determining in which frustums an object belongs to. There are many frustum culling techniques available, one approach to efficiently handle this is to perform an frustum cull with the camera frustum against all objects. Then determine which subfrustum each object intersects by its distance from the near clip plane of the camera and the radius of its bounding sphere.

There are four methods for implementing CDR. The first one requires only a single render target. This one works like basic rendering, accumulating the scene by rendering the subfrustums back to front. The second method would be using two render targets, using one as accumulative and another for each frustum iteration. This allows screen space effects when applying each layer to the accumulative one. The third method would be an extension of the second method. This would require three or more render targets and allows for rendering each frustum in different threads applying them at completion. This would allow for multithread rendering. Fourth would be using one render target for each frustum division. This would ultimately allow to multithread all rendering and merge all frustums as preferred. However it will require huge amounts of video memory and should be only be used when having data from every frustum is required.

In this thesis, the first method, being the easiest to implement and cheapest in resources has been implemented and will be used for testing the visual quality and will serve as a proof of concept. However the other implementations should work just as well, if not even better.

When rendering using one of the above mentioned methods, all the necessary information for calculating the shading is saved to the render target or targets to produce the final image in a later stage. When all the frustums have been rendered the final image is produced using the information stored in the render target or targets in a deferred rendering manner.

The performance implications and solutions of using multiple frustums is discussed in depth by Cozzi and Ring [6] however, using fast culling techniques with a good data structure will minimise the time spent on frustum checks. As costly as the frustum splits may be they also allows for a level of detail culling depending on which subfrustum they are rendered upon allowing one to render the same object with multiple levels of detail at once.

Producing a screen space effect like depth of field without any special shaders may be produced by rendering each subfrustum in different resolutions depending on which frustum that needs to be in focus, however this is untested and may produce artefacts at the frustum splits due to high resolution differences, but may produce an increase in render speeds because of fewer fragments to shade.
Chapter 4

Smallest Resolvable Depth Separation

The smallest resolvable depth separation gives a clue to when depth fighting will start to occur as it denotes how small the view space depth difference between two polygons can be before depth fighting appears and this thesis uses it to measure the quality between the techniques. To calculate the smallest resolvable depth separation for the standard depth buffer the code from the Z Calculator by Baker [2] is used and for the logarithmic depth buffer the equation given by Ulrich [11] is used. CDR used the same equation as Baker but with frustums divisions as described in Section 5.2. The reliability of the equations by Baker and Ulrich is questionable but they seem to be correct at first sight, it is noteable that they do not take into account other factors such as field of view, which has an impact on the smallest resolvable separation noted by Akeley and Su [1]. The full implementation of the script used to compile the data below can be found in appendix Listing A.2.

All the results below are extracted with the following settings: near clip at 0.001, far clip at 100 million, Depth Buffer at 24bit.

<table>
<thead>
<tr>
<th>Smallest Resolvable Depth</th>
<th>Standard</th>
<th>CDR</th>
<th>Logarithmic</th>
<th>LogCDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.001</td>
<td>4.096</td>
<td>129.592</td>
<td>662.386</td>
<td>2428.750</td>
</tr>
<tr>
<td>0.01</td>
<td>12.958</td>
<td>409.810</td>
<td>6623.860</td>
<td>24287.501</td>
</tr>
<tr>
<td>0.1</td>
<td>41.010</td>
<td>40980.545</td>
<td>66238.604</td>
<td>242875.013</td>
</tr>
<tr>
<td>1</td>
<td>130.028</td>
<td>129592.205</td>
<td>662386.037</td>
<td>3643125.454</td>
</tr>
</tbody>
</table>

Table 4.1: At what view space depth the smallest resolvable depth equals to 0.001, 0.01, 0.1 and one unit for each technique.

Table 4.1 show data from Figure 4.1 at with smallest resolvable depth separation at 0.001, 0.01, 0.1 and one unit. Assuming that one meter is qual to one unit the table shows at what depth, for each method, that we have meter, decimeter, centimeter and millimeter precision. With the standard buffer only able of re-
Figure 4.1: The horizontal axis denotes the view space fragment depth and the vertical axis denotes the resolvable depth separation.

solving one unit of depth until the view space z-depth is 130.027. CDR manages about 99,565 percent further than the standard, Logarithmic at about 509,321 percent increase and LogCDR with a 2,801,722 percent increase from standard. However these numbers do not scale with the smallest resolvable depth number with the exception of logarithmic as it has constant relative precision.
Chapter 5

Implementation

The standard depth buffer in OpenGL can be either 8, 16, 24, 32 bits. The depth buffer used in this thesis’ implementation is of 32 bits resolution for standard, logarithmic and CDR. The standard implementation is the same as the implementation for CDR but with only one frustum thereby skipping the dividing of subfrustums and sorting which subfrustum each object belongs to.

5.1 Implementation of Logarithmic Depth Buffer

This thesis uses Ulrich’s logarithmic depth buffer as shown in Equation 2.1 and as seen implemented in Listing 5.1. Ulrich’s implementation suffers from the issue with long polygons close to the camera but this thesis chose Ulrich’s implementation as it is one of the most recent publications that solves other issues that previous works suffers from. The implementation in Listing 5.1 takes a vertex point and transforms it according to Ulrich’s Equation. The transformed location is easier to store in the depth buffers allowing for much higher precision.

\[
gl\_Position\_z = 2.0 * \log (gl\_Position\_w/\text{near})/\log (\text{far/near}) - 1;
\]
\[
gl\_Position\_z = gl\_Position\_w;
\]

Listing 5.1: This thesis implementation of a Ulrich’s logarithmic depth buffer equation.

5.2 Implementation of Cascaded Deferred Rendering

Of the methods mentioned for implementation in Chapter 3, this thesis uses the first method mentioned; where a single render target is used and the frustums are rendered back to front.

Each frame is rendered by the following process. The scene is culled using the frustum and all objects are stored in a list. Then it loops through all found objects
and places them in the subfrustum lists corresponding to which subfrustums the object intersects. An optimisation here is done by measuring the distance from the nearclip and then comparing if it is inside the range of each subfrustum instead of doing a full frustum intersection test. After this step all the data required to render the scene is collected. The rendering is iterative with each subfrustum. Each iteration clears the depth buffer and sets the new perspective matrix and then renders the objects intersecting the subfrustum.

```java
import java.lang.Math;

float target_near_clip = 0.1f; // user defined near clip
float target_far_clip = 12000.0f; // user defined far clip

float near_clip = target_near_clip;
float far_clip = 0.0f;
float overlap = 1.0f;

List<Float> frustums = new ArrayList<>();

// Create frustums
while (far_clip != target_far_clip) {
    far_clip = Math.min(near_clip * 1000.0, target_far_clip); // calculate the far clip
    frustums.push(new float[]{near_clip, far_clip + overlap}); // store frustum near and far clip
    near_clip = far_clip; // store near clip for next frustum
}

// Render Frustums from back to front
while (frustums.size() > 0) {
    RENDER_FRUSTUM(frustums.pop()); // Render with last frustum
}
```

**Listing 5.2:** Frustum Splitting with Near/Far ratio at 1000.

The far clip for a frustum split can be calculated easily by multiplying the current near clip by one thousand and the whole procedure can be implemented as described in Listing 5.2. The reason for doing the frustum calculations from near to far is that the first frustum is guaranteed to have the full ratio thus having a smaller chance at producing visible artefacts between the first and second frustum.
Chapter 6

Results

6.1 Picture Quality

Figure 6.1: Standard Depth Buffer.

Figure 6.2: Logarithmic Depth Buffer.

Figure 6.3: CDR.

Figure 6.1, 6.2 and 6.3 show a field of cubes at a distance of 12,000 units, lit by one point light. The camera’s near clip is at 0.01 units and far clip at 100 000 units.

Figure 6.1 shows how the depth testing failing for about every fragment as cubes are overlapping each other in the wrong order. Both Figure 6.2 and 6.3 both shows correct depth sorting of the fragments with a just few pixels difference.

6.1.1 2AFC Experiments

Although smallest resolvable depth separation proves the quality of each technique it is important to validate the resulting images using responses from human observers. In total, 12 participants (12 men) in an age range of 21-27 took part in the experiment. All stimuli were presented remotely on the subjects own equipment.
6.1.2 Experiment Scene

The experiment scene was constructed as follows. In the center there was a field of cubes containing 1000 cubes and a white point light, each cube 1 unit in size, the field was 32 by 32 by 32 units, each cube was placed at random within the field. The camera was set at a distance of 1 million units away from the cubes and circled the field of cubes 360 degrees, field of view was set at 0.002, near clip at 0.001 and far clip at 2 million units. 360 frames were exported in tga format for each technique (Standard, CDR, Log, LogCDR) and converted to MP4 videos using ffmpeg.

6.1.3 Procedure

The subjects were asked to view six 19 second videos containing two separate video sequences, side by side, containing two different techniques. After each trial, the participants were asked to judge which side had the least amount of flickering. Participants should choose between "left" or "right". The videos were presented as in Table 6.1, with each technique appearing three times.

<table>
<thead>
<tr>
<th>Video Nr.</th>
<th>Left</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Standard</td>
<td>CDR</td>
</tr>
<tr>
<td>2</td>
<td>CDR</td>
<td>Logarithmic</td>
</tr>
<tr>
<td>3</td>
<td>Standard</td>
<td>LogCDR</td>
</tr>
<tr>
<td>4</td>
<td>Standard</td>
<td>Logarithmic</td>
</tr>
<tr>
<td>5</td>
<td>CDR</td>
<td>LogCDR</td>
</tr>
<tr>
<td>6</td>
<td>Logarithmic</td>
<td>LogCDR</td>
</tr>
</tbody>
</table>

*Table 6.1: The six video sequences.*

6.1.4 Results

<table>
<thead>
<tr>
<th></th>
<th>Standard</th>
<th>CDR</th>
<th>Logarithmic</th>
<th>LogCDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>10</td>
<td>25</td>
<td>35</td>
</tr>
</tbody>
</table>

*Table 6.2: Experiment results showing how many times a technique were chosen producing the least amount of flickering.*

<table>
<thead>
<tr>
<th></th>
<th>Standard</th>
<th>CDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>10</td>
</tr>
</tbody>
</table>

*Table 6.3: Comparing Standard with CDR from the results, showing how many times Standard, respectively CDR, were chosen producing the least amount of flickering.*
Chapter 6. Results

Table 6.4: Comparing CDR with Logarithmic from the results, showing how many times CDR, respectively Logarithmic, were chosen producing the least amount of flickering.

<table>
<thead>
<tr>
<th>CDR</th>
<th>Logarithmic</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 6.5: Comparing Logarithmic with LogCDR from the results, showing how many times Logarithmic, respectively LogCDR, were chosen producing the least amount of flickering.

<table>
<thead>
<tr>
<th>Logarithmic</th>
<th>LogCDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 6.6: Comparing Standard with Logarithmic from the results, showing how many times Standard, respectively Logarithmic, were chosen producing the least amount of flickering.

<table>
<thead>
<tr>
<th>Standard</th>
<th>Logarithmic</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 6.7: Comparing CDR with LogCDR from the results, showing how many times CDR, respectively LogCDR, were chosen producing the least amount of flickering.

<table>
<thead>
<tr>
<th>CDR</th>
<th>LogCDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 6.8: Comparing Standard with LogCDR from the results, showing how many times Standard, respectively LogCDR, were chosen producing the least amount of flickering.

<table>
<thead>
<tr>
<th>Standard</th>
<th>LogCDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 6.2 shows the overall results of the experiment. Table 6.3 to Table 6.8 shows the results of the experiment when comparing two individual techniques from the results.
6.1.5 Statistical Analysis and Discussion

The results are believed to be realistic due to the conformity of the observed data although the user study was not optimally outlined; the viewing conditions and equipment may have vastly differentiated between the test subjects and may have incurred skewed results as the experiments was performed remotely. Also, the structural outline of the user study may have produced a learning effect due to the non-randomized sequence of the video playlist where the right side always had videos with a better depth buffer quality.

The results from Table 6.2 to Table 6.5 were analyzed to determine any statistical significance. A technique called Chi-squared is used to find out whether there is a pattern of preference in the number of participants who chose the techniques that were predicted to produce less depth fighting [5]. As the results is binomially distributed a nonparametric test is used. In this experiment a one-sample Chi-squared includes only one dimension. The observed frequencies were compared to an, for each result, equally distributed expected value to ascertain whether the difference would be significant. The Chi-squared values were computed and then tested for significance, as shown in Table 6.9. The calculated Chi-squared values show that the subjects preferred CDR over Standard, Logarithmic over CDR and LogCDR over Logarithmic, as shown in Table 6.9, also notable is that the subjects chose Logarithmic over Standard, LogCDR over CDR and LogCDR over Standard. This indicates that the subjects preferred the techniques that were expected to produce better results and that there is a significant visual difference between the techniques.

<table>
<thead>
<tr>
<th></th>
<th>( \chi^2 = )</th>
<th>df =</th>
<th>p &lt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard vs CDR</td>
<td>5.333</td>
<td>1</td>
<td>.05</td>
</tr>
<tr>
<td>CDR vs Logarithmic</td>
<td>12</td>
<td>1</td>
<td>.05</td>
</tr>
<tr>
<td>Logarithmic vs LogCDR</td>
<td>8.333</td>
<td>1</td>
<td>.05</td>
</tr>
<tr>
<td>Standard vs Logarithmic</td>
<td>12</td>
<td>1</td>
<td>.05</td>
</tr>
<tr>
<td>CDR vs LogCDR</td>
<td>12</td>
<td>1</td>
<td>.05</td>
</tr>
<tr>
<td>Standard vs LogCDR</td>
<td>12</td>
<td>1</td>
<td>.05</td>
</tr>
</tbody>
</table>

*Table 6.9*: Output for the Chi-Squared Analysis.

6.2 Performance

The performance results were gathered using a Intel Xenon E5-1650 3.2Ghz CPU with one Nvidia Quadro 4000 GPU on Windows 7.

The performance measurements were taken using a scene with 64,000 cubes and one point light. The camera rotates one revolution around the y-axis for
10,000 frames before switching to the next mode. The rotation is to simulate a dynamic camera moving around in a virtual environment as found in games. The times presented include the whole process of rendering the scene. Time measurements are taken by the implementation using Windows specific “QueryPerformanceCounter” function which allows for time measurements with millisecond precision. Rendering is also asynchronous and stalls the whole pipeline until the frame is fully completed to ensure valid time measurements.

The graph in figure 6.4 shows the average times it takes to render a single frame. Using the standard method as the baseline one can deduct that with logarithmic depth buffers disabled, CDR with two divisions is 3.4 percent slower, CDR with three divisions is 4.6 percent slower, CDR with four divisions 5.7 percent slower and CDR with five divisions is 7.1 percent slower. Logarithmic without CDR enabled amounts to 4.4 percent slower. With CDR enabled the performance is 8.0 percent slower for two divisions. CDR with three divisions is 8.2 percent slower. CDR with four divisions is 8.4 percent slower. CDR with five divisions is 10.4 percent slower.
Chapter 7

Discussion

The performance measurements of logarithmic combined with CDR having two, three, and four divisions seem a bit odd but the tests have been done three times on multiple computers showing the same oddity. A reason for this strange result could be optimizations across hardware implementations which we do not know about. Another reason could be that either the GPU or the CPU is bottlenecking the results since the counter waits until the whole frame is completed. However one should still be able to use these numbers to calculate an approximate of the expected performance loss when implementing this.

One worry when using logarithmic depth buffer is the possible artefacts mentioned by Cozzi and Ring [6]. These artefacts are described to appear when using logarithmic depth buffer with thin objects behind the camera however no sufficient specification of reproduction exists and during the testing of the implementation they never appeared. Therefore the artefacts should not be a problem in applications and if they appear there are usable solutions as mentioned in Chapter 2.

The 1 to 1000 near-clip to far-clip ratio mentioned by Cozzi and Ring was used to define where our subdivisions happened in the camera frustum [6]. The 1 to 1000 ratio was usable with 3 divisions with a near-clip of 0.001 but as seen in Figure 4.1 after the distance of 100,000 depth-fighting appears.

The smallest depth resolvable formulas in Chapter 4 are not reliable in practice. While the math behind it appears correct, assuming our references are correct, and tests commited shown that some depth fighting still occurs before the values calculated. This is likely due to various important parameters being left out, such as FOV, or due to graphics card supplier’s implementation as different results appeared on different GPUs. Most likely it is a combination of multiple factors and therefore it is recommended to have a practical test instead of relying on the formulas in applications where a certain depth buffer quality is required.
Chapter 7. Discussion

This thesis focus is on expanding the view frustum to the maximum size while maintaining good depth buffer resolution. However when designing an application which does not require a near to far ratio greater than 1000 one should consider abstaining from using the solutions mentioned in this thesis as they add complexity and require time to implement. If the depth resolution is still insufficient when using 1000 as near to far ratio the recommendation would be to try logarithmic depth buffer as it is simpler to implement and see if that solves it.

The current implementation of the standard non-linear depth buffer exists because it was beneficial when the depth buffer was first invented. This allowed for high precision near the camera and low precision far away. It was most likely not a big problem at all since old games probably didn’t cross the 1 to 1000 ratio and the low screen resolution made the flickering much less obvious. The standard depth buffer most likely met all requirements while also being very fast for old games but a new method would be required for newer games with far distances.

The usage of logarithmic depth buffers is unlikely to replace the standard solution for OpenGL or DirectX as they do cause a performance loss and they are not required if you make applications that do work with the standard buffer. CDR is also unlikely to become the standard as it requires scene management which lies on a higher level. If these appear somewhere it would be inside game engines where they are explicitly used to solve the issues of depth fighting.
This thesis has explored three techniques, logarithmic, CDR and LogCDR, to solve depth-fighting and then compared them against the standard depth buffer and each other. Logarithmic being a GPU based solution, CDR being a CPU based solution and LogCDR being both CPU and GPU based.

This thesis predicted that it would be possible to divide the frustum into sub-frustums and render each separately to reuse the depth buffer, this proved possible until the depth buffer was no longer usable at extreme distances. Using CDR, one would be able to render objects virtually infinitely far away with maintained quality, with the definition of quality being no visibile depth fighting, proved to be false, the standard depth buffer used in CDR degraded quite quicky even with the 1000 near-clip to far-clip ratio and logarithmic depth buffer had less degration than the one of CDR. And as noted by the user study, CDR generated more flicker than logarithmic and LogCDR.

A minor performance loss was predicted but it would be worth the quality gain and the technique should be usable in real time applications such as games. It is difficult to assess the performance loss against quality gain as the scale of the performance loss is relative to the complexity of the scene as well as near- and far-clip and other factors. Using the smallest resolvable depth separation one could calculate that CDR could render 98,000 percent further than the standard depth buffer having at most one unit of depth precision error as seen in Table 4.1. In our scenario, this generated a performance loss of about 4.6 percent compared to the standard depth buffer. And because of such small performance losses were measured it is higly suitable for real time applications. Combining CDR with logarithmic depth buffer proved to be not only possible but also proved to be the method that could render the furthest with the best precision as calculated and shown in Table 4.1, as well as proven by the user study. LogCDR could render about 450 percent further than logarithmic with at most one unit of depth precision error 4.1.

With the data used in Figure 4.1 the 1000 ratio presented and used for both
Chapter 8. Conclusions

CDR and LogCDR was found to be non-optimal; the depth imprecision is increasing with the depth and thus the ratio should decrease accordingly to maintain a maximum resolvable depth separation for each subfrustum.

The best technique found for rendering long distances is LogCDR with about 2,975,000 percent distance coverage increase over the standard depth buffer as shown in Table 4.1 and produced the lowest amount of flicking according to the user study, Table 6.9. However LogCDR requires both GPU and CPU performance and therefore the most expensive technique as shown in Figure 6.4. Logarithmic being is the fastest at rendering both fairly long distances, according to the data presented in Table 4.1 and Table 6.4, while only requiring changes to the shader program with no CPU overhead. CDR can be an appropriate solution as it can render great distances but not nearly as far as logarithmic. CDR also requires great caution when implemented and a solution to transparent geometry still needs to be found. CDR in itself requires CPU time for frustum checks and thus brings a performance implication as noted by the results, that can scale greatly with the complexity of the scene.

For future work a GPU multithreaded implementation of CDR would be interesting to assess and compare against a logarithmic depth buffer. It would also be interesting to see what performance difference between logarithmic and CDR would be in a real-life example. Solving the issue of transparent and translucent geometry with CDR is also a great topic for future work. As the 1000 ratio for CDR is not optimal another equation could be constructed to further increase the range of CDR but this would increase the frustums required.
References


Appendix A

Listing A.1: Frustum Intersection (C++)

```cpp
// Standard Depth Buffer
// Uses the equation from The "Z Calculator" by Steve Baker
// at http://www.sjbaker.org/steve/omniv/love_your_z_buffer.html
function zprec(nbits, zFar, zNear, z) {
    var b = zFar * zNear / (zNear - zFar);
    var res = (b / ((b / z) - 1.0 / (1 << nbits))) - z;
}
```
if (res < 0.0001) return -res;
else return -Math.floor(res * 100000.0) / 100000.0;
}

// CDR
function cdrzprec(nbits, zFar, zNear, z) {
  var near_clip = zNear
  var far_clip = 0.0
  var overlap = 1.0
  var frustums = [];
  // Create frustums
  while (far_clip !== zFar) {
    far_clip = Math.min(near_clip * 1000.0, zFar); // calculate the far clip
    if (z <= far_clip && z >= near_clip) {
      return zprec(nbits, far_clip, near_clip, z);
    }
    near_clip = far_clip; // store near clip for next frustum
  }
}

// Logarithmic Depth Buffer
// Uses the separation equations given by Tatcher Ulrich
// at http://tulrich.com/geekstuff/log_depth_buffer.txt
function logzprec(nbits, zFar, zNear, z) {
  var K = Math.pow(2, nbits) - 1;
  // tatcher log depth z
  function f(z) {
    return K * Math.log(z / zNear) / Math.log(zFar / zNear);
  }
  // the view z value that maps to depth--buffer value i
  function g(i) {
    return zNear * Math.exp((i / K) * Math.log(zFar / zNear));
  }
  // how far (in view space) to the next discrete depth value
  function delta(z) {
    return g(f(z) + 1) - z;
  }
  return delta(z); // relative precision
}

// LogCDR
function logcdrzprec(nbits, zFar, zNear, z) {
  var near_clip = zNear
  var far_clip = 0.0
  var overlap = 1.0
  var frustums = [];
  // Create frustums
  while (far_clip !== zFar) {
    far_clip = Math.min(near_clip * 1000.0, zFar); // calculate the far clip
    if (z <= far_clip && z >= near_clip) {
      return logzprec(nbits, far_clip, near_clip, z);
    }
    near_clip = far_clip; // store near clip for next frustum
  }
Appendix A. Appendix

```javascript
var res = "Z-Depth, Standard, CDR, Logarithmic, LogCDR\n";
var bit = 24;
var zfar = 100000000;
var znear = 0.001;
for (var i = znear + 0.0011; i <= zfar; i = Math.min(i * 1.10, zfar)) {
    res += i.toFixed(4) + ",";
    res += zprec(bit, zfar, znear, i).toFixed(20) + ",";
    res += cdrzprec(bit, zfar, znear, i).toFixed(20) + ",";
    res += logzprec(bit, zfar, znear, i).toFixed(20) + ",";
    res += logcdrzprec(bit, zfar, znear, i).toFixed(20) + 
    if (i === zfar) break;
}
console.log(res); // Prints CSV Formatted Data

Listing A.2: Smallest Depth Separation Data Extraction Tool (Javascript)
```