Progressive Texture Streaming

Masters’s Thesis

Jaap van Muijden
Progressive Texture Streaming

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Jaap van Muijden
born in Leiden, the Netherlands
© 2013 Jaap van Muijden.

Cover picture: Graphical view of used DXT palettes in a test image used during this research.
Progressive Texture Streaming

Author: Jaap van Muijden
Student id: 1250264
Email: jaapvanmujiden@gmail.com

Abstract

Since its conception, the increasing need for graphical detail has been a driving factor for the gaming industry. The 3D scenes rendered in modern games have increased in detail and scale at a rapid rate, and will continue to do so for the foreseeable future. The ever-increasing processing power and the development of faster 3D rendering hardware has enabled game developers to present the player with large, densely populated 3D virtual worlds containing thousands of high resolution textures and models. In order to render these virtual worlds on current gaming hardware, texture streaming is quickly becoming a necessity: the current gaming consoles cannot contain all texture data of a game world within their graphical memory. The proposed texture streaming solution addresses this problem by managing the texture data between the optical media and graphical memory at run-time. This thesis details a progressive, mipmap aware texture streaming scheme, designed around the optical media used by the current generation of gaming consoles. In the proposed solution, bandwidth usage is reduced by progressively streaming textures through a DCT (Discrete Cosine Transform)-based compression scheme. Seek time of the optical media is reduced by employing a spacial packing scheme which aims to minimize the amount of separate read operations. Using these tools, textures can be accessed and streamed in various levels of detail, providing an efficient use of memory while keeping bandwidth usage and seek times at a minimum.

Thesis Committee:

Chair: Prof.dr.ir. F.W. Jansen, Faculty EEMCS, TU Delft
University supervisor: Prof.dr.ir. F.W. Jansen, Faculty EEMCS, TU Delft
Company supervisor: Dhr. Arno van Wingerden, Triumph Studios
This thesis is part of my graduation requirements for the Master Degree in Computer Graphics at the TU Delft. It contains the results from research done from January 2008 to August 2010. The streaming system has been developed at Triumph Studios in Delft, a Dutch game studio that released six titles to date including one major title during my time there. The system is currently completed and part of Triumph’s technical portfolio.

Jaap van Muijden
Delft, the Netherlands
August 12, 2013
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>iii</td>
</tr>
<tr>
<td>Contents</td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vii</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Problem Definition</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Goals</td>
<td>4</td>
</tr>
<tr>
<td><strong>2 Background</strong></td>
<td>5</td>
</tr>
<tr>
<td>2.1 Level of Detail</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Visibility and sensitivity</td>
<td>6</td>
</tr>
<tr>
<td>2.3 Content streaming in a gaming architecture</td>
<td>11</td>
</tr>
<tr>
<td>2.4 Image Compression</td>
<td>12</td>
</tr>
<tr>
<td>2.5 Mipmapping</td>
<td>15</td>
</tr>
<tr>
<td><strong>3 Design</strong></td>
<td>17</td>
</tr>
<tr>
<td>3.1 Overview</td>
<td>18</td>
</tr>
<tr>
<td>3.2 LOD Management</td>
<td>18</td>
</tr>
<tr>
<td>3.3 Texture Mapping</td>
<td>20</td>
</tr>
<tr>
<td>3.4 Texture Streaming Pipeline</td>
<td>22</td>
</tr>
<tr>
<td>3.5 Mipmap construction from DCT data</td>
<td>29</td>
</tr>
<tr>
<td>3.6 DXT Encoding and performance</td>
<td>34</td>
</tr>
<tr>
<td><strong>4 Implementation</strong></td>
<td>35</td>
</tr>
<tr>
<td>4.1 YCC transform</td>
<td>36</td>
</tr>
<tr>
<td>4.2 Chromatic Subsampling</td>
<td>44</td>
</tr>
<tr>
<td>4.3 DCT transform</td>
<td>52</td>
</tr>
<tr>
<td>4.4 Run-Length Encoding</td>
<td>64</td>
</tr>
<tr>
<td>4.5 Huffman Coding</td>
<td>67</td>
</tr>
</tbody>
</table>
# CONTENTS

4.6 Performance ........................................... 75  
4.7 DXT Encoding ........................................... 75  

5 Measurements and Results ........................................... 79  

6 Conclusion ........................................... 81  

A DXT compressed texture format ........................................... 83  
A.1 DXT 1 ........................................... 83  
A.2 DXT 2, 3 ........................................... 84  
A.3 DXT 4, 5 ........................................... 84  
A.4 DXT Compression ........................................... 85  
A.5 Software ........................................... 86  

B Discrete Cosine Transform ........................................... 87  
B.1 Multidimensional DCTs ........................................... 89  

C Mipmapping ........................................... 91  

D Proposed prediction and control system ........................................... 93  
D.1 Cell based prediction ........................................... 93  
D.2 Camera based prediction ........................................... 94  
D.3 Interface ........................................... 96  

E Data Management ........................................... 97  
E.1 Data Packing ........................................... 97  
E.2 Data overlaying ........................................... 100  

Bibliography ........................................... 103
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Xbox 360 and PS3 consoles, both high-end, current-generation gaming platforms.</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>In-engine screenshots of the two recent games developed by Triumph Studios. Overlord (right) and Overlord 2 (left) [25].</td>
<td>2</td>
</tr>
<tr>
<td>1.3</td>
<td>Comparison of a scene running with various texture resolutions. Notice the difference on the wall on the right side of the image. [10].</td>
<td>3</td>
</tr>
<tr>
<td>1.4</td>
<td>On the left the standard rendering system with intermediate loading stages. On the right a streaming model with constant updates and constant user interactivity.</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>CSF function plotting threshold contrast against frequency [21].</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>Threshold Contrast measured over multiple degrees of eccentricity[21].</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>A scene that has been weighted with the PODS algorithm[23], showing higher values for front objects than the occluded models behind them.</td>
<td>10</td>
</tr>
<tr>
<td>2.4</td>
<td>A Basic rendering architecture where the rendering and simulation are and global streaming data flow. The dotted line denotes the additional prediction data that is fed into the streaming system.</td>
<td>12</td>
</tr>
<tr>
<td>2.5</td>
<td>An image shown decomposed into the three channels of the YCbCr space.</td>
<td>14</td>
</tr>
<tr>
<td>2.6</td>
<td>An image shown decomposed into the three channels of the YUV space.</td>
<td>14</td>
</tr>
<tr>
<td>2.7</td>
<td>Example of a mipmap chain, each layer is half the size of the one before it.</td>
<td>15</td>
</tr>
<tr>
<td>2.8</td>
<td>Global data flow of the proposed streaming method.</td>
<td>16</td>
</tr>
<tr>
<td>3.1</td>
<td>Global data flow of the proposed streaming method. It shows the streaming data flow along with the three design units: LOD Management, Data Packing and Texture Reconstruction.</td>
<td>18</td>
</tr>
<tr>
<td>3.2</td>
<td>Workflow of dynamically updating a texture.</td>
<td>21</td>
</tr>
<tr>
<td>3.3</td>
<td>Texture Streaming data flow of the proposed streaming method. It shows the streaming data flow along with the four stages of the pipeline: Precaching, Progressive Reconstruction, DXT Compression and Texture Update.</td>
<td>22</td>
</tr>
<tr>
<td>3.4</td>
<td>Progressively creating a full DXT mipmap hierarchy from stream data.</td>
<td>23</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

3.5 Two test images and their frequency distributions. It shows that sharp edges result in high frequencies; the wood has sharp edges along the vertical axis, resulting in higher energy vertical frequencies. ........................................... 24
3.6 Basic dataflow of the proposed texture compression scheme during data conversion. ......................................................................................... 26
3.7 The eight base cosine functions that are used in the DCT. ....................... 28
3.8 The eight cosine functions and their filtered variants. .............................. 31
3.9 The four top mipmaps and their valid coefficients .................................. 32
3.10 The global layout of a streamable mipmapped texture, showing the four sets of coefficients used to generate the four progressive mipmaps. ................. 32
3.11 The finished data layout. Although it is framed on a scan line basis, other frame sizes can be used. Note that the two scan lines containing luminosity info are twice the size as the chromatic ones, creating a 1:1:4 grouping. .................. 33
4.1 The main stages of the entire streaming and compression pipeline. ........... 35
4.2 Simply shifting out bits results in corruption on some input values. ............ 38
4.3 Various methods of changing input data to reduce image size. Left shows the more complex algorithm, right shows the original solution. The images show the difference between the original and the adjusted image magnified by 32. ...... 39
4.4 A hundred runs of converting a four megapixel image to YCoCg space and back. The horizontal axis denotes milliseconds. ........................................ 40
4.5 The data flow of SIMD-enabled conversion. The input data is unpacked, processed in two parts and repacked again. ............................................. 41
4.6 A hundred runs of converting a four megapixel image to and from YCoCg space. The vertical axis denotes milliseconds. .................................... 44
4.7 Detailed data flow between YCC conversion, Chromatic Subsampling and DCT transform. Note that the chromatic subsampling also separates the data into separate channels. ................................................................. 45
4.8 Initial performance of the subsampling code in milliseconds per run over a four megapixel image; the results are a magnitude worse than the YCoCg pass. .... 46
4.9 The data layout of a block of data before- and after tiling. The numbers show the order in which the data blocks are stored in memory. ...................... 47
4.10 The tiled image data in the YCoCg-Chromatic Subsampling dataflow .......... 47
4.11 Comparing tiled and non-tiled subsampling. ........................................... 49
4.12 The global steps for SIMD-enabled chromatic subsampling. .................... 50
4.13 Comparing the various subsampling implementations. .......................... 51
4.14 Comparing the Co channel of a subsampled image, the PSNR was 44 dB. .... 52
4.15 Comparing a subsampled image with the original; the PSNR is 42 dB. ......... 53
4.16 Subtiling the 16x16 Y channel tiles into four 8x8 tiles using SIMD intrinsics. 54
4.17 The Y channel converted into DCT values. ............................................ 55
4.18 The Y channel recreated as a 2x2 subsampled mipmap using a subset of the DCT coefficients to the left with the original data to the right and the difference between the two in the middle. ................................. 56
4.19 The timings of the first DCT conversion. .............................................. 57
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.1</td>
<td>Example of a mipmapped texture. Notice that the second level takes up only</td>
<td>92</td>
</tr>
<tr>
<td></td>
<td>one fifth of the texture data.</td>
<td></td>
</tr>
<tr>
<td>D.1</td>
<td>Two scenes and their visibility graphs and time tables, using simple cell-based</td>
<td>94</td>
</tr>
<tr>
<td></td>
<td>prediction rules.</td>
<td></td>
</tr>
<tr>
<td>D.2</td>
<td>Prediction volumes for directional prediction and distance prediction</td>
<td>95</td>
</tr>
<tr>
<td>E.1</td>
<td>Convolution with a sphere to create footprints of the various texture levels</td>
<td>98</td>
</tr>
<tr>
<td>E.2</td>
<td>Rasterizing footprints of the various mipmaps of a single textured polygon mesh</td>
<td>99</td>
</tr>
<tr>
<td>E.3</td>
<td>A Flat Merge, followed by two cut operations</td>
<td>100</td>
</tr>
<tr>
<td>E.4</td>
<td>Two layers: after Flat Merges and Cuts, the boxes of different layers are merged</td>
<td>101</td>
</tr>
<tr>
<td>E.5</td>
<td>Scanning the allocated data for a suitable overlay by keeping the last encoun-</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>tered positions of each data pack. In the figure, a minimum span of 8 blocks</td>
<td></td>
</tr>
<tr>
<td></td>
<td>has been found. Instead of allocating another 3 blocks, the same data can be</td>
<td></td>
</tr>
<tr>
<td></td>
<td>acquired by seeking over 8 existing blocks, reading 3 of them.</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In 2007 I started working on a texture streaming system for a Dutch game studio called Triumph Studios, located in Delft, The Netherlands. Triumph Studios is a relatively small game studio that produced several successful franchises such as Overlord and Age of Wonders. The studio incorporates high quality models, textures and post-process effects into their games which are rendered by a state of the art render engine. The ability to deliver a diverse game world that is filled with high quality models and textures is one of their biggest selling points. The latest games made by Triumph Studios use cross-platform engine that runs on PC and current generation high-end consoles: the Playstation® 3 (PS3) from Sony and the Xbox 360 from Microsoft.

![Figure 1.1: Xbox 360 and PS3 consoles, both high-end, current-generation gaming platforms.](image)

The biggest challenges that the technical team of Triumph Studio faces are the hardware constraints imposed by the two gaming consoles in terms of memory, processing power and bandwidth. The Xbox 360 only contains 512MB of shared memory in total, while Sony’s PS3 narrows this constraint even further by using separated System- and Graphical memory of 256MB each [19]. In both cases, this is a fraction of the memory available on current PC hardware. Furthermore, both consoles have optional hard disk drives, meaning that each
game must be able to run directly from the game disk. Even with these constraints, the games released during the lifetime of both consoles keep evolving both in visual quality and scale. In order to stay ahead of the competition, Triumph Studios needs to constantly improve the efficiency of their engine while incorporating the latest innovations in rendering technology.

The main platform for Overlord 1 was Xbox 360, which was released in 2007; however, both PC and PS3 versions of the game were developed in parallel and released within a year of the Xbox 360 release. With Overlord 2 in 2009, the PS3 took a more prominent role, becoming the main platform during development. The technical team incorporated more of its hardware specifications into their development cycle, resulting in a more optimized rendering engine and a vast performance improvement in the PS3 build of the game.

![In-engine screenshots of the two recent games developed by Triumph Studios. Overlord (right) and Overlord 2 (left) [25].](image)

### 1.1 Problem Definition

In order to create detailed environments, the studio is looking into ways of optimizing memory usage, and it was decided early on that the main focus of this research would be improving texture memory efficiency. In Overlord and Overlord 2, by far most of the memory was used by textures; optimizing the texture memory usage will enable the designers to add more texture detail, higher quality meshes, and the additional graphical buffers that are needed for the more complex visual rendering and post-processing effects.

During the development of Overlord 2, it was not uncommon for the game to crash during the development cycle due to newly added graphical assets (meshes and textures). The additional data would cause the memory allocation system to exceed the memory budget in mid-game when additional allocations were done. Although the engine was designed to limit run-time allocations for exactly this reason, memory usage was constantly pushed to its limit and dynamic memory allocation could not be completely avoided. To work around these issues, artists had to manually assess what texture assets could be removed or reduced in size, fixing memory issues as they were encountered.

To stay competitive in the current generation of consoles, these memory problems need to be resolved. The limited amount of available memory is proving to be a serious obstacle
in the creation of larger and more detailed game worlds. Triumph Studios needs a solution that manages memory usage at run-time, allowing new titles to make better use of the available memory and present the player with a higher quality game experience.

For an effective way to optimize texture memory usage, I opted for a streaming architecture. Streaming involves the continuous loading of additional (texture) data from the game disk into the video memory: loading in textures in the resolution and at the precise time they are required by the game while simultaneously swapping out textures that are no longer needed. For this concept to work, the streaming system needs to constantly run in the background, constantly communicating with the game and the graphical memory system of the console.

In addition to alleviating the memory problems, such a system would reduce the loading times after the initial start of the game. In this particular case, the streaming system would also serve as a memory debugging platform: the texture streaming system would be explicitly aware of any memory budget problems and can be used as a memory profiler to detect any situations where texture memory would be exceeding its budget. The use of a streaming system would reduce memory management issues since textures would no longer use up a variable amount of memory; texture data would always be managed within the fixed memory budget of the streaming system.

Because the data sizes of the individual textures could be upwards of 2MB, bandwidth constraints have to be taken into account. Reading large amounts of data at run-time from
1. Introduction

the optical media in particular is subject to long and unpredictable seek times and a relatively low data throughput: both the PS3 and Xbox have optical transfer speeds around 10MB/s. In order to mitigate this constraint, additional research was done into progressive streaming and graphical data compression. Progressive texture streaming enables the streaming system to increase the resolution of an existing texture by adding additional data instead of loading in a completely new texture from optical disk. Compression reduces the data size of textures on the disc, further increasing the efficiency of the streaming system. Both techniques would lower peak throughput requirements and reduce the needed bandwidth from optical disk overall. In addition to the texture streaming system, Appendix E describes spacial partitioning and packing algorithms to create an optimal data layout on the optical disk, leveraging data duplication to reduce seek times.

1.2 Goals

Following from this analysis of the problem space, we can establish a number of concrete goals our texture streaming system should achieve:

- The texture streaming solution should leverage progressive streaming to reduce bandwidth between graphical memory and optical disk.
- In order to function in a memory constrained environment, the streaming architecture should have an easily scalable footprint.
- DXT compression is a widely used compression standard for textures. In order to make this a relevant alternative, our compression solution should out-perform regular DXT compression.
- In order to compete with the more standard streaming solutions, our compression algorithm should have a similar or higher pixel throughput while effectively reducing bandwidth usage.

In this document, all the above issues will be discussed: Chapter 2 introduces the contextual issues further, discussing some underlying principles and exploring some of the research done in the field of bandwidth-limited streaming systems. Chapter 3 contains the completed design of the texture streaming system that was created for this research, adding further designs for a supporting data management framework in Appendix E. Chapter 4 describes the implementation of the streaming pipeline, including algorithms and performance timings. The research concludes with Chapter 5, containing test results of the streaming pipeline using a set of representative textures from the game industry. Finally, the conclusion and any thoughts on future research can be found in Chapter 6.
Chapter 2

Background

Since the introduction of 3D-acceleration hardware, graphical memory size has been a big constraint to most rendering solutions. The access time to GPU memory was usually several factors faster than normal system memory, and this fact holds true even for the current generation of high-end consumer electronics. For realistic, real-time rendering applications and games, the graphics memory is primarily used to store the texture and mesh data used for the rendering of the current scene. To get as much detail as possible using limited memory and processing power, most 3D scenes limit the amount of textures used in a single scene and try to reuse the available textures and meshes as much as possible. In most modern 3D environments, textures take up more memory than any other type of asset. Because of this, we will focus on streaming and compression techniques with the goal to achieve maximum texture detail with a given memory and bandwidth budget. In this sections, some of the used techniques and design choices in the periphery of the main research subject are discussed. Section 2.2 will discuss the basics of visibility and human perception. Section 2.3 goes into some detail about the data architecture of a game and how streaming can be integrated. Section 2.4 will discuss various aspects of image compression. More detailed information about the used prediction model, image formats and mathematical constructs can be found in the appendices.

2.1 Level of Detail

One of the functions of a streaming system is managing the amount of detail that is currently rendered in the scene in order to get the best image quality, given the memory constraints and processing power of a system. In most rendering engines there are various (sub)systems that deal with this metric, such as impostor rendering [8] and occlusion systems [4]. In the most basic case, objects are either rendered or culled, based on their visibility to the viewer. In this chapter we extend this to include various levels of detail for each rendered asset. This means that assets can be rendered in different complexities, using (sub)sets of the related asset data. The goal of any render system is to reduce the complexity of the various assets.

1\footnote{An asset denotes any external data that is used in the creation of a 3D scene. This includes the textures and meshes used in the scene.}
parts of a scene in order to maximize the trade-off between the perceived visual quality \( Q_S(S) \) and rendering costs \( C_S(S) \) of a scene \( S \). This quality/complexity metric is denoted by Detail Efficiency \( E \) in this thesis. This metric can be expressed by a simple ratio between the complexity and quality of a scene.

\[
E(S) = \frac{Q_S(S)}{C_S(S)}
\]  

(2.1)

The method used to determine the visibility of an object and its impact on the scene is called the visibility heuristic.

2.1.1 Bandwidth-critical systems

With the introduction of more complex on-line applications, lowering the bandwidth needed to stream a 3D scene over a communications channel has become an interesting research subject. Bandwidth critical systems employ more advanced visibility heuristic in order to minimize bandwidth usage at the cost of other resources and make extensive use of progressive model generation. Texture detail becomes an even bigger issue since textures contribute heavily to the data volume of a model. Bandwidth is not a problem on most PC applications since scenes can be usually loaded completely into memory. However, with hardware like Microsoft’s Xbox 360, and Sony’s PlayStation 3, the scenes can not be loaded directly into memory because of the relatively small amount of memory available. Big scenes must be streamed directly from optical media without any way of caching the data beforehand. Together with the high display resolution of HD-gaming, bandwidth becomes one of the main bottlenecks of the streaming system. From the standpoint of texture streaming, this means that the loaded resolution of each texture should be taken into account and weighed against its visibility.

2.2 Visibility and sensitivity

In the world of real-time graphics, loss-less texture compression is a luxury; every byte has to be weighed against the increase in visual fidelity. In order to create an efficient rendering system, trade-offs between looks and real-time performance must be considered at every design step. It is very tempting to use “common sense” to determine the heuristics and guidelines when considering the optimum detail levels of the various objects in a rendered scene. However it may prove useful to investigate some of the theory behind detail perception by looking at the way users (humans) perceive images. This can help to better understand what impact the various levels of detail have on the perceived image quality, and how to optimize Detail Efficiency.

Luebke et al.[7] describes the perceptual metric called the contrast sensitivity function (CSF) a measure of perception of visual stimuli by the human eye as a function of their spatial frequency. This CSF can be used as a way of determining the perceptual impact caused by the various simplifications that are presented in this chapter. An example of a typical CSF be seen in Figure 2.1, where threshold contrast is plotted against frequency.
The threshold contrast is the lowest luminance contrast still distinguishable at a certain frequency.

![Graph showing CSF function plotting threshold contrast against frequency](image)

**Figure 2.1:** CSF function plotting threshold contrast against frequency [21].

### 2.2.1 Visual frequencies

A large amount of research has been done into the perceptibility of visual stimuli. The simplest relation between visual stimuli and its visibility is *Webers Law*. This states that the visibility of a spot on a uniform field increases linearly with the background luminance. However, this bears little resemblance to the images created by 3D applications and games. Various studies have established that perceptibility of a sinusoidal grating has a strong relation with its frequency in cycles per degree of visual arc. These sinusoidal gradings are still a great distance from the typical scenes of interest. However, it turns out that this CSF still holds for composed sinusoids. Using Fourier Transforms, any image can be decomposed into base sinusoids, and these decompositions can be used to get relatively accurate visibility information. Figure 2.2 shows the results of a visibility test, where the contrast sensitivity is measured function of frequency, tested using multiple degrees of eccentricity (angular distance from the stimuli to the visual center). Contrast sensitivity is defined as the reciprocal of the Threshold Contrast. It can be seen that high-frequency stimuli are more...
2. **BACKGROUND**

readily visible to the user while low-frequency stimuli are less visible, especially in the peripheral view.

![Spatial Frequency vs. Threshold Contrast](image)

Figure 2.2: Threshold Contrast measured over multiple degrees of eccentricity[21].

### 2.2.2 Center of View

The visual acuity\(^2\) changes as a stimuli gets closer to the center of vision. The *fovia*, the central one degree of vision, has the highest acuity, and the visual periphery has a significantly lower acuity. This can be used to estimate the visibility of details around the edge of the screen since the viewer will usually focus on the center of the screen. Leubke et al.[21] describe a way to measure the worst-case visual impact of the edge-collapse simplification used in Progressive Meshes and uses the perceptual attributes of spatial frequencies to estimate the visibility of such simplifications.

### 2.2.3 Visibility heuristics

One of the key elements of an effective streaming system is the heuristic that determines in what detail an object should be rendered for an optimal Detail Efficiency of the entire scene. Most systems use the distance of the model to the camera as only factor; this has the advantage of being easy to work with but it does not take (partial) occlusion into account. Sander et al.[22] propose a simple distance based heuristic for their detail level determination where the distance to the camera determines what level of detail is used. The spacing between the various detail levels increases exponentially, and every detail level increases the amount of faces in their models by four. This way, every doubling in rendering distance increases the detail four-fold. This keeps the rendered faces at a somewhat constant size in screen space, and thus create a somewhat constant visual detail. The heuristic is as follows:

\[
i = \text{floor}\left(\log_2\left(\frac{d - s}{k}\right) + 1\right)
\]

\(^2\)Acuity is measured as the highest perceptible frequency.

8
Visibility and sensitivity

Where \( d \) is the distance to the camera, and \( k \) and \( s \) are variables that can be used to tune the detail based on the rendering power available. When for instance the frame rate drops, the overall level of detail can be decreased by increasing \( s \) or \( k \). By altering \( s \) and \( k \), the switching points can be positioned on different points in the distance curve. Although this algorithm will be sufficient for most real-time rendering of complex scenes. For some systems it pays to spend more time computing the visibility of a model in order to get a more exact lower bound for the acceptable level of detail.

2.2.4 Model weighting

Tian et al. [23] propose a more precise method in their PODS scheme. In this system, objects are first received at the lowest level of detail through a (bandwidth constrained) channel. The resulting scene is then inspected to determine how visible the rendered models are. The amount of occlusion is then used as a weight to determine the impact of a model on the image quality. In order to determine the visibility of the various models, first the bounding volumes of the models are checked against the viewing frustum of the camera. This culling step makes sure no further processing is done on the objects that are not visible. The remaining models are sorted back to front, using the distance from the middle of the model to the viewpoint.

In the second pass, the projection plane is divided into an \( m \times n \) grid of patches \( \{ p_{ij} | i = 0..m, j = 0..n \} \), and every patch is issued a weight based on the distance of its center \( v_{ij} \) to the viewpoint \( v_0 \):

\[
\omega_{ij} \propto \frac{1}{||v_{ij} - v_0||} \tag{2.3}
\]

Where \( v_{ij} \) is the distance of the middle point of patch \( p_{ij} \), and \( v_0 \) is the viewpoint. The next step is to paint the patches that are inside the bounding volume of the models, starting with the front model. When a patch is painted, it cannot be painted by other models. After this step, the amount of patches a model painted is used as a visibility metric. The occlusion values are normalized so that the sum of the visibility metric over all models in the scene is 1.

When bandwidth becomes available, detail information is requested and sent to increase the detail of the models that are most visible in the scene. This model also incorporates a progressive streaming system: it is not efficient to send new detailed versions of the same mesh, so instead only additional detail information is sent, which is used to increase the detail on the model. This results in an effective usage of bandwidth, where the available bandwidth is not wasted on improving objects that cannot be seen by the user.

Determining Scene Quality

Before a choice can be made as to which model should be improved, the relative quality between detail levels should be determined. This can be done during the creation of the static meshes in the case of a static LOD system. Tian et al. [23] describe the use of a Peak Signal To Noise Ratio (PSNR) metric to compare the quality between different detail levels of a model, and how this can be used to determine the overall distortion of a scene. With
2. Background

Figure 2.3: A scene that has been weighted with the PODS algorithm[23], showing higher values for front objects than the occluded models behind them.

the effective increase in PSRN along with the bandwidth costs of every detail increase per model, the most efficient use of bandwidth can be determined.

First we define the difference between two surfaces \( S \) and \( \hat{S} \) by using the root mean squared (rms) distance between them:

\[
\varepsilon_{\text{rms}}(\hat{S}, S) = \left( \frac{1}{\hat{S}} \int_{\hat{S}} e^2(v, S) ds \right)^{1/2}
\]

And the maximum distance:

\[
\varepsilon_{\text{max}}(S, \hat{S}) = \max_{v \in \hat{S}}(e(v, S))
\]

The point to surface distance function \( e(v, S) \) is given as:

\[
e(v, S) = \min_{\hat{v} \in S} \|v - \hat{v}\|
\]

With these defined, the normalized distortion of a model \( M \) in resolution \( n \) can be expressed as:

\[
D(M^n) = \frac{\varepsilon_{\text{rms}}(M^n, M^L)}{\varepsilon_{\text{max}}(M^O, M^L)}
\]

Which gives a measure how distorted model \( M \) is, compared to its lowest and highest possible detail. Given these, the distortion \( D_s \) of a scene \( S \) with \( k \) models who in turn have detail level \( i_k \) can be expressed as:

\[
D_s = \sum_{M_k^i \in S} \omega_k \cdot D(M_k^i), 0 \leq \omega_k \leq 1
\]
Where $\omega_k$ is the visibility factor of model $k$, and $D_s = \sum_{M^k_i \in S} \omega_k = 1$ With this information, the application should choose to upgrade the detail $i$ of the model $k$ with cost $C_{ki}$ where

$$\omega_k \cdot \frac{D(M^k_i) - D(M^k_{i+1})}{C_{k,i+1}}$$

is optimal.

A more precise way of measuring visibility would be to count the amount of pixels every model draws on the screen, similar to the PODS system. With the recent availability of hardware capable of doing visibility-checks during rasterizing, the measurement can probably be implemented fully in hardware, using a low-resolution render to a stencil buffer or using the various z-test techniques available. The bounding volumes can be made as complex or simple as the application allows. Breaking the volumes into smaller parts can also yield more occluding information, which can be used with the more complex progressive model structures, such as Selective Refinement as proposed by Hoppe et al. [14].

Although it is outside of the scope of this document to go deeper into this material, it is shown that the human eye is more sensitive to high-frequency visuals than to low-frequency visuals, falling off at the high end when the details become too small to be perceived. What we can take from this, is that high frequency artifacts will be more noticeable to the human eye. Any hard numbers on peak visibility frequencies cannot be used efficiently in the proposed system, since the eventual resulting frequencies of each detail will be highly dependent on viewing distance and physical dimensions of the rendering viewport (TV or PC monitor).

### 2.3 Content streaming in a gaming architecture

Streaming content at run-time can solve a lot of memory and performance issues, but content streaming itself is also not without intrinsic drawbacks. One of the biggest issues with content streaming in general is determining what content should be streamed in and out at any given moment, and what content should remain permanently in memory. To further discuss this, it pays to look at the global architecture of an application as a whole; although there are a lot of different architectures that can be employed when creating a game engine, we limit our discussion to a simple rendering architecture shown in Figure 2.4 construction.

The simulation of the virtual scene is running completely independently of its rendering. This means that any graphical representation that does not affect the simulation does not need any synchronization back into the simulation model.

In the most simplistic streaming model, the only synchronization that takes place is between the rendering system (where the streaming system resides) and the rendering API$^3$. The synchronization centers around memory usage: the rendering software must take into account that texture data cannot be altered or moved when it is used by the rendering hardware. However, rendering (by the GPU) is by definition done in parallel to the application itself (which runs on the CPU). Because of this, API-specific callbacks and interrupts must be used to synchronize the internal rendering process and graphical memory access of the rendering system.

$^3$Through the course of this document, we will only consider DirectX and OpenGL, since most high-end real-time graphical applications use a derivative of these two APIs.
2. Background

In more complex streaming models, prediction logic is used that reads user input (from the control layer) and camera behavioral logic (from simulation model), creating an additional data flow from the model simulation into the streaming system.

![Figure 2.4: A Basic rendering architecture where the rendering and simulation are and global streaming data flow. The dotted line denotes the additional prediction data that is fed into the streaming system.](image)

Rendering systems clearly define what data is needed in the graphical memory for each individual frame: any data that is used in the rendering of the scene has to be present in graphical memory, because reading from system memory is too slow for most real-time applications\(^4\). Since there is delay between the initial request for an asset from disc and the asset being completely streamed in, the streaming system ideally needs to look ahead to make sure all data is requested even before it is called for by the renderer. Prediction logic plays a critical part in this: prediction further into the future creates a longer time-frame in which to load in the needed asset data but will result in more false positives, reducing efficiency. Prediction with more accuracy means there will be less data thrown away without being used, using the available time and memory space more efficiently. This is also the area where visibility heuristics become important since the various streaming commands should be prioritized according to their impact on the quality of the visible scene. In many ways, the prediction system is part of the streaming system, because the prediction centers around the various asset detail levels and their memory footprints. A detailed description of a prediction framework we are using during the development of the streaming system can be found in Appendix D.

2.4 Image Compression

Besides managing the texture data, the proposed streaming system also compresses the image data in order to reduce bandwidth even further.

\(^4\)In architectures like the Xbox 360, there is a shared memory between CPU and GPU; however, there is also a limited amount of high performance embedded DRAM (eDRAM).
2.4.1 DXT Compression

In all modern rendering hardware, the texture data is compressed using DXT compression. This compression has a fixed compression of 6:1 on RGB data\(^5\), encoding blocks of 4 \(\times\) 4 pixels in 64 bits. This compression scheme is block based and can be decoded internally by the rendering hardware. Even though the decoding requires additional decoding hardware, the overall performance increases on DXT compressed textures due to the bandwidth bottleneck in modern GPU architectures. However DXT compression does not support any kind of progressive scheme and cannot compress more efficiently than the said 1:6 compression ratio. Therefore we will base our streaming on a different compression scheme. However, in order for the streamed textures to be used by the rasterizing hardware, we will need to recode all textures to DXT before they are loaded into video memory. The DXT compression scheme is further discussed in Appendix A.

2.4.2 Discrete cosine transform

Because DXT does not fulfill all the requirements we set for our compression scheme, we will need to look at other, lossy compression schemes that do support progressive compression and deliver better compression rates. The schemes discussed in this paper convert the image data into the frequency domain in order to leverage the frequencies characteristics of the CSF function discussed in Section 2.2. The most common frequency transform is the Discrete Cosine Transform (DCT), which we will be using as part of our proposed compression scheme. This type of transform is also used in image compression schemes such as JPEG and audio compression schemes such as MP3. The DCT we will be using is a 8 \(\times\) 8 2D transformation, which will be applied on separate blocks of 8 \(\times\) 8 pixels, resulting in 64 frequency coefficients belonging to 64 2D base functions. Details of the basic process are described in Appendix B and the encoding and decoding is further detailed in Chapter 3.

2.4.3 Color Spaces

One of the fundamental choices in image compression is the choice of color space. In computer graphics hardware (and most software), sRGB is the standard color space. This color space was defined by HP and Microsoft for use in most computer graphics software, monitors and printers [27][15]. This three-dimensional color space is made up by three additive primaries: red, green and blue.

The sRGB color space, like all RGB color spaces, has the unfortunate property of combining chromaticity and luminance in its three components. This makes it unsuited for any lossless compression of color data; human eyes are much more sensitive to brightness than chromaticity, in other words the CSF is usually much higher for brightness than chromaticity. By processing the chromaticity and brightness separately, textures can be compressed with a much higher efficiency and detail levels can be defined in accordance with human perception. In order to process these two properties separately, the chromaticity and luminance information have to be split; this can be done by changing to color spaces that work

---

\(^5\)On RGBA data, DXT has a compression of 1:4, in this thesis we will only consider the case of non-alpha textures.
with split chromaticity and luminance, such as the YCbCr [6], YUV [26] or YCoCg [16] space.

### 2.4.4 YCbCr color space

The YCbCr color space was created because the standard RGB color space has a lot of redundancies and is therefore not suited for transmission over media. The YCbCr is a practical approximation of human perception: Better compression can be achieved by separately compressing the chromaticity (Cb, Cr) and luminance (Y), because of the difference between luminance and chroma sensitivity in the human eye.

![Figure 2.5: An image shown decomposed into the three channels of the YCbCr space.](image)

### 2.4.5 YUV color space

The YUV space is used frequently inside the rendering pipeline of HDR lighting systems; because the chromaticity is split from the luminance, the luminance can be stored in a higher (floating point) precision, while the chromaticity can be stored as low resolution integers. For instance, instead of saving a RGBA value to a 32-bit render surface, a short float (16 bits) for the luminance and two bytes for the two chromaticity channels can be stored in the same surface, sacrificing a channel for increased luminance range [20].

![Figure 2.6: An image shown decomposed into the three channels of the YUV space.](image)

### 2.4.6 YCoCg color space

Like the YCbCr color space, the YCoCg space separates the RGB space into two chromaticity channels and one luminance channel; the YCoCg space is designed to have a (near) loss-less conversion to and from RGB space, working completely with integer arithmetic. A second definition, sometimes called YCoCg-R can transform loss-less to and from 24-bits RGB space, but requires two extra bits. In our design, we choose to use the first definition, since we will be dealing with loss-less compression further down the pipeline and we
Mipmapping greatly value the fact that we can transform data in-place using only integer arithmetic. In Chapter 3, the YCoCg color space is examined in more detail.

2.5 Mipmapping

Mipmapping is a rendering technique used by all modern rendering hardware in order to render textures more efficiently. A mipmapped texture is stored as a set of images in multiple resolutions; each image in the set is half the resolution of the image before it, containing downsampled versions of the original texture. The individual images are called mipmaps, or mipmap levels, while the entire set of images is called a Mipmap Chain. When a mipmapped texture is sampled by the rasterizer, the mipmap that best matches the sampling frequency is used. This increases cache efficiency and removes the need for additional filtering. In practice, the lower mipmaps are used when the texture is viewed from a distance while the higher mipmaps are used when the texture is viewed up closely.

![Example of a mipmap chain, each layer is half the size of the one before it.](image)

2.5.1 Mipmap Streaming

The texture streaming system can use mipmaps as a way to effectively increase texture detail of existing textures. In Figure 2.8, a simple streaming case is shown that uses mipmap based texture streaming: it shows a corridor with 3 textured objects and a representation of the camera. In the left image, the rendered screen shows object A and B, while object C is not seen, and its textures are not loaded into the graphics memory. In the middle image, the camera moves and the lower (smaller) mipmaps of object C are loaded (streamed) into texture memory. In the last image, the camera looks upon the freshly updated object C while object A is out of sight, and thus can be partially or completely removed (streamed out) from
memory to make room for the higher (larger) mipmap levels of C. If the streaming is fast enough, the higher mipmaps are loaded in before they are used by the rasterizer hardware.

2.5.2 Graphics Hardware API

In order to manage memory efficiently, the streaming system does not only change the contents of the texture data, but also other properties of the textures such as the resolution and compression methods. All of these changes are done at run-time and influence the internal rendering logic of the hardware. All modern gaming hardware such as the Xbox 360 and PS3 support such run-time adjustments without completely resetting internal texture systems or any other additional overhead, although platform-specific timing and synchronization systems are necessary to avoid graphical glitches. These details will be further discussed in Chapter 3.
Chapter 3

Design

The main design focus of this texture streaming solution is to minimize the bandwidth usage of the optical media while still presenting the user with the best possible graphical detail. The main effort towards this goal is the reduction of the amount of data that needs to be read from media using progressive compression.

The most straightforward way to reduce bandwidth usage is to compress the texture data itself. Current rendering hardware works with DXT-compressed\(^1\) data in order to increase bandwidth efficiency. Unfortunately this type of compression does not meet all our requirements for use in an efficient progressive streaming solution. DXT is a fixed-ratio compression that does not support any progressive streaming scheme and cannot easily be compressed further beyond its maximum ratio of 1:6. Because of these shortcomings, our streaming pipeline needs to be built around a different compression standard which does support progressive construction. Section 3.4 details the texture compression method we are using in more detail, and how a progressive compression scheme can be designed to meet run-time streaming requirements. In addition to streaming mipmap data from optical disk, the mipmap data has to be recoded to a DXT format so they can be rendered. Because there is a strong data correlation between the various detail levels of the same texture image, this thesis focuses on a progressive decompression approach, where streamed in data can be combined with the data already available to reconstruct a new level of detail. These higher levels of detail are implemented as additional mipmap layers as suggested in Section 2.5. In the two leading gaming consoles, the data is stored on optical disk; Microsoft’s XBOX 360 and Sony’s PS3 use DVD and Blu-Rays disc respectively. Although there is a difference in access times and storage capacity between the two consoles, the main focus of the proposed system is to minimize bandwidth and seek time overhead to the optical media by reducing the amount of separate read instructions. Additionally, Appendix E shows a packing architecture that can improve seek times at the cost of storage space.

\(^1\)The DXT compression standard is known under its original name: S3 Texture Compression (S3TC), and is now part of a set of hardware supported block compression schemes (BCn).
3. DESIGN

Figure 3.1: Global data flow of the proposed streaming method. It shows the streaming data flow along with the three design units: LOD Management, Data Packing and Texture Reconstruction.

3.1 Overview

Figure 3.1 shows the general layout the streaming system, it is a more detailed layout of the streaming system that was presented in Figure 1.4.

At run-time, the streaming system is controlled by the LOD Management system. This system determines what level of detail is required for each texture using visibility determination along with application specific information such as camera behavior, hinting heuristics and additional meta-data. During rendering, the system determines what the optimum resolution is for each texture given the bandwidth and memory budget available. It then prioritizes the streaming operations and issues commands to disk I/O and the Texture Reconstruction system. After loading the required data into memory, the Texture Reconstruction system decompresses the textures from a packed format and recodes it to a format compatible with the GPU. During the final step in the streaming pipeline, the texture is updated on the GPU side by the Texture Update functionality. Next to these two run-time systems, there are two stages of the Data Conversion system during off-line data conversion: compression that compresses the textures into a streamable form and data packing that compiles the compressed data into packs to reduce seeking overhead.

3.2 LOD Management

The LOD Management system manages the level of detail of all streamable assets in the scene. In the current context of texture streaming, this comes down to the amount of mipmaps of each texture asset. It needs to keep track of available memory and analyze the scene to determine the visibility of streamable assets. From that data, it outputs and prioritizes streaming requests to the disk I/O and Texture Reconstruction system. Because there
is a latency when requesting textures from disk, a certain predictive functionality needs to be added to ensure texture data is available when needed by requesting it beforehand. The next section describes the method that is used to determine the amount of mipmaps that should be streamed in for each texture, and how the various mipmaps are prioritized across all textures.

### 3.2.1 Visibility determination

In most cases, the textures used by an asset are presented to the user in a subsampled form due to the limited output resolution: when textures are sampled below half of their original size, the second mipmap is used, this means that 75% of the texture data is not accessed by the GPU during rendering. In these cases, a texture can be loaded in a lower detail without having a negative effect on the visual quality of the rendered scene. In a case of maximum efficiency, all textures should be streamed in up to this optimal resolution at all times.

Determining the visibility and sampling frequency of a textured surface is very complex; size, opacity, occlusion and orientation are factors in determining the visible area of a texture in both screen and texel space. In current modern graphics hardware, there are ways of obtaining the amount of pixels rendered from a draw call [9]. However, even when the amount of affected pixels can be obtained, the mapping and sampling determine what part of the texture is actually sampled. A better approach to get sampling information is to check the geometric data and UV-maps from the object; a first suggestion is to use the position and projected size of each polygon surface, combined with the UV coordinates as a simple detail visibility metric, however this would mean every surface should be considered separately. Moreover, most artists make use of varying amounts of intentional oversampling (stretching) on textures without high frequency components and/or unintentional undersampling on poorly visible areas. This approach becomes very complex, invasive and is not usable for a high-performance real-time system.

Luckily a more high-level, user-driven metric can be used instead. Most 3D models are created with an ideal viewing distance in mind. Given the viewing angle of the camera and the display resolution, this translates into an ideal rendering resolution. When we assume all textures are created to be sampled at full resolution when the asset is rendered in this ideal resolution \( R_{\text{ideal}} \), we can take the full texture resolutions as a high bound for texture sampling frequencies. This gives us a starting point in our visibility space: at \( R_{\text{ideal}} \), we assume all textures are fully visible, and sampled at their full resolution. The rendering resolution and camera view angle are known parameters during asset creation, so this resolution can be converted into an ideal in-game viewing distance \( d_{\text{ideal}} \).

Since the texture sampling frequency of a surface is linearly proportional to the rendering resolution of that surface, and that the rendering resolution of an object is inversely proportional to rendering distance, we can conclude that the relative sampling frequency of a texture is inversely proportional to rendering distance. This means that if an object has a maximum texture sampling frequency of \( f_{\text{max}} \) at distance \( d_{\text{ideal}} \), it will have a maximum texture sampling frequency of \( \frac{f_{\text{max}} d_{\text{ideal}}}{d_a} \) at distance \( d_a \). Its sampling frequency will halve for each doubling of the rendering distance.
3. Design

Similarly, we can take the camera viewing angle into account. When the camera zooms in, the rendering angle changes and the effective rendering resolution $R_{\text{eff}}$ of the objects becomes significantly larger. This can be combined with the rendering distance as follows, where $\alpha_i$ is the base viewing angle, and $\alpha_a$ is the zoomed-in camera setting.

$$R_{\text{eff}} = \frac{R_{\text{ideal}} \times d_i}{\tan(\alpha_i) \times \tan(\alpha_a)} \times d_a$$

(3.1)

Note that the exact value or definition of $R_{\text{eff}}$ and $R_{\text{ideal}}$ is not directly of importance; the linear relation between the two is important since this will be used to determine what mipmaps of the texture are used and should be streamed in/out.

What we take from this, is that we can assume that each object will need all its highest, full-resolution mipmaps when rendered from $d_{\text{ideal}}$ or closer, while the used mipmap level for each texture will drop by one (halving resolution) for each doubling of the viewing distance. From a streaming standpoint, we will call this adjusted mipmap level the Requested Level of Detail. A big advantage of this model is that $d_{\text{ideal}}$ is a very intuitive value that can be later tweaked or adjust by asset creators and designers. We will consider $d_{\text{ideal}}$ to be available as additional meta-data for each object in our 3D scene. For most objects this value can be set automatically to reasonable defaults, making it relatively easy to integrate into an existing asset library.

3.2.2 Prioritizing and prediction

After collating the visibility of all streamable assets, the LOD Management system has to send requests to the disc I/O and texture reconstruction system to start streaming in the texture with the highest priority. The system goes through all textures, and will select the texture with the largest disparity between requested and actual level of detail (mipmap level). In case of a tie, the streaming command with the smallest amount of data is chosen first, this effectively being the lower hanging fruit in this context.

3.3 Texture Mapping

After the disc I/O has downloaded the texture data from disk, the Texture Reconstruction system transcodes (decoded and then recoded into another format) the texture data from a progressive compression scheme into a hardware-compatible (DXT) mipmap. However, this new mipmap is not being referenced by the rendering hardware yet: it needs to be appended to the mipmap chain of the existing texture in order for it to be accessed by the GPU.

In all current hardware, a texture is divided in at least two structures: a header and a block of texel data containing the mipmap chain. The texture header contains all information needed by the rasterizer to use the texture during rendering such as height, width, format and a pointer to the texel data. When a texture is selected into the hardware, the header is read and used to setup the sampler hardware. During rendering, the sampler uploads data from the data block to its caches as needed. While the header and mipmap chain can reside in different parts of the graphical (accessible) memory; the mipmap chain itself needs to be structured as a contiguously block of memory.
Because the mipmap chain needs to be contiguous, the mipmap chain has to be reconstructed as a whole, with the new mipmap attached to it. Only then can the texture header be updated to reference the newly created chain. After the header has been updated, the old mipmap chain can be discarded and its memory returned to the memory pool. This update operation is shown in Figure 3.2. Note that this will cause memory fragmentation to occur over time(!) and additional systems are needed to periodically defragment texture memory.

Figure 3.2: Workflow of dynamically updating a texture

### 3.3.1 Mipmapping hardware

As mentioned in Chapter 2, mipmap chains consist of several mipmap levels $l_0...n$. Each texture contains a re-sampled form of the full resolution image, each layer being half the resolution of the one before it. The top level $l_0$ of the set being the texture in full resolution, the bottom one $l_n$ being a $1 \times 1$ texture. The resolution $R_m$ of each level $m$ can be calculated according to Equation 3.2$^2$, where $R_{\text{full}}$ denotes the full base resolution of the texture. In this document we will stick with the numeration in most graphics APIs, which uses 0 to denote the full resolution texture. Note that not all textures have mipmap chains, but we will only consider mipmapped textures suitable for streaming. More technical information can be found in Appendix C.

$$R_m = \frac{R_{\text{full}}}{2^m} \quad (3.2)$$

During the rasterization of a surface, the texture sampling hardware determines the texture coordinate differential $(du, dv)$ between adjacent pixels and calculates the required mipmap level to use for optimal performance. This automatic selection is done completely by hardware and causes no additional computational overhead. Additionally, all current gaming hardware supports trilinear filtering, which adds the mipmap chain as an additional dimension to the standard bilinear sampling, enabling the hardware to smoothly blend between two mipmap levels. This removes any visible edges on surfaces where the hardware switches between mipmaps.

By default, the sampler has access to the entire mipmap chain; this means that adding or removing a mipmap level while the texture is visible on screen could cause visible glitches

---

$^2$This document uses OpenGL mipmap numbering, other systems inverse the numbering, naming the $1 \times 1$ layer as layer 0.
when the sampler suddenly switches to a higher mipmap level. Fortunately, all modern hardware support manual overrides for mipmap selection through pixel shader instructions, making it possible to smooth out any visual pops or glitches by smoothly increasing or decreasing a maximum mipmap level, leveraging the trilinear filtering to smooth the transition.

3.4 Texture Streaming Pipeline

![Figure 3.3: Texture Streaming data flow of the proposed streaming method. It shows the streaming data flow along with the four stages of the pipeline: Precaching, Progressive Reconstruction, DXT Compression and Texture Update.](image)

As discussed in Section 3.3.1, the streaming pipeline should generate texture updates in the form of additional mipmap layers for existing textures. Furthermore, in order to be compatible with the rendering pipeline, all textures must be stored in DXT-format; this compression scheme is discussed in detail in Appendix A. The DXT compression scheme has a static compression ratio, so the size of any mipmap layer in GPU memory is fixed by its format and size. In the basic case, the streaming subsystem can upload pre-compressed mipmap layers into graphical memory which are then simply combined with the existing mipmap chains in memory to stream in additional detail, as described in Section 3.3.1. The drawback of this approach is that due to the DXT’s fixed compression ratio, the amount of (progressive) compression that can be leveraged is minimal. Since the DXT format does not lend itself well for further compression or optimizing (see Appendix A), any compression must be applied to the original texture data, and later transcoded to DXT at run-time before uploaded into a mipmap chain. At the front-end of the pipeline, a precaching stage will preload the streaming texture data into memory when needed. As with the Texture Reconstruction, this precaching functionality is controlled by LOD Management.
3.4.1 Progressive Compression

It should be clear that there is a strong correlation between each layer of a mipmap chain: each level contains basically the same image sampled at a different rates. Progressively constructing a higher mipmap using the already loaded texture data is therefore a viable option. Figure 3.4 shows the proposed texture construction data flow of the streaming pipeline. Note that it shows certain pre-loaded base data; instead of progressing all the way from a 1 × 1 texture, the first n – 4 levels can be provided as a non-progressive baseline, this will remove unnecessary overhead since the data size of the smaller mipmap layers decreases exponentially into chains small enough to keep permanently in memory. It will be shown later in Section 3.4.3 that the proposed progressive compression scheme supports up to four progressive levels. The rest of the mipmap levels is considered to be base data that is permanently stored in memory.

![Figure 3.4: Progressively creating a full DXT mipmap hierarchy from stream data.](image)

3.4.2 Image Characteristics and Quantization

Before the compression scheme was decided, some research was done to determine what the characteristics were of the data that would be streamed. For this purpose, a series of test images was studied. The graphical style of a game or 3D environment strongly influences the kind of texture used but in the context of this thesis the focus will be on more realistic, noisy and organic textures used in modern games.

At first glance, progressively streaming the texture would have only a limited impact on bandwidth usage: each mipmap is four times as big as its lower one and using progressive streaming "only" saves the cost of resending the lower mipmap, which comes down to a bandwidth reduction of 25% (Appendix C goes into more detail). However, using quantization and entropy encoding, the effective data size of the higher mipmap can be reduced significantly, resulting in a more even data distribution between mipmap. Quantization is the mapping of a large set of input values to a less continuous, smaller set of discrete val-
ues. Given a fixed data range, quantization reduces entropy and increases the effectiveness of entropy encoding methods such as Huffman Encoding.

However before we can assess the impact of quantization of the image data we need to take a closer look at some test images. The first property that can be leveraged is the fact that the CSF of the human eye is much lower for chromaticity than it is in luminance. To take advantage of this fact, the luminance data has to be separated from the luminance data; this is done by transforming the RGB data into a color space that has a separate luminance channel, a few of these spaces are described in Section 2.4.3. After separation, we can subsample the chromaticity data in a process called Chromatic Subsampling. This reduces the image size across all mipmaps.

![Two test images and their frequency distributions. It shows that sharp edges result in high frequencies; the wood has sharp edges along the vertical axis, resulting in higher energy vertical frequencies.](image)

Another fact we can take advantage of is the fact that the CSF is monotonically decreasing after a certain frequency and that the size of a rendered texel on a monitor or TV lies well beyond this point. When we convert a typical image to the frequency domain, a large energy differential in the frequency spectrum can be seen: there is a lot more contrast in the lower frequencies than in the higher frequencies, resulting in larger coefficient values. This is also where the type of image plays a large role: graphics containing clear, sharp edges seem to have a different energy distribution among its frequencies than the more organic, natural images such as photographs. Figure 3.5 shows the energy distribution of the pic-
Textures in frequency space\textsuperscript{3} It clearly shows the high energy in the lower frequencies, while the higher frequencies contain close to zero energy.

Lower energy values means a lower variance which (with a constant quantization) results in lower entropy. Because the CSF for these higher frequencies is lower, we can apply additional quantization, further reducing the entropy, increasing the compression ratio of these high frequencies significantly while maintaining image quality when it is transformed back to the spatial domain. During subsampling, low frequencies are preserved while high frequencies are discarded. Since a mipmaps is effectively a subsampled versions of the one above it, it follows that the higher mipmaps may be larger, but they contain a lot more low energy/low entropy frequencies, resulting in smaller compressed data sizes while the lower mipmaps are smaller but have less impressive compression ratios.

The conclusion is that even though progressive streaming only saves the lower mipmaps, these mipmaps contain more information per texel and when entropy compression is used in the frequency domain, the lower mipmaps make up a sizable portion of the compressed data and the savings are quite significant. The details of Chromatic Subsampling, the frequency domain transform and the various compression methods will be described in more detail in this chapter.

3.4.3 Compression method

A main design goal of the compression method is low bandwidth usage, something that it has in common with most image compression methods used in another very active field of research: the Internet. Below is a short list of the criteria the compression method should meet.

Size The used format should have a good compression ratio, ideally with a quality/size ratio that can be specified per texture. The type of compression must be suitable for textures. This excludes compression schemes that use global palettes such as the compression used in Graphics Interchange Format images.

Image Except for some very specific graphical styles, most textures are bitmap-based and of somewhat organic nature, ruling out any vector-based compression schemes and encouraging the use of frequency-domain compression methods.

Progressive Since all mipmaps can be created from a single source that is progressively streamed in, the compression scheme must support progressive loading of images. Most importantly it must support it in such a way that instead of simply increasing quality within an image, it should be able to increase (double) resolution at each progressive step.

Speed Decompression will be done at run-time, so speed is certainly a large factor.

Simplicity The compression algorithm has to be ported and maintained on several architectures with limited resources, so (code) simplicity is an issue. Using a minimum

\textsuperscript{3}The shown spectra are not straight Fourier transforms, but rather a visualization of the discrete cosine coefficients.
amount of passes and a minimum amount of additional external tuning data keeps the compression system transparent to the users (which in this context are the artists working on creating texture assets for a game).

Two known compression formats fit these criteria: JPEG and JPEG2000 [1]; the first being based on a DCT (Discrete Cosine Transform) compression method, while the second is based on a wavelet compression method. Favoring simplicity and speed above pure compression power, the JPEG compression scheme is used as a starting point for our own. The streaming overview with the JPEG compression stack integrated can be seen in Figure 3.6.

As a basic layout of our compression scheme, we use a compression stack based on the JPEG standard, with additional layers to enable progressive mipmap creation. Figure 3.6 shows this compression stack, consisting of the following layers:

**YCC transform** RGB space is not very well suited for compression, since it mixes chromatic and luminosity information (see Section 2.4). Therefore the compressed texture data is split into Luminosity ($Y$) and two color (chromatic) channels ($C_o$ and $C_g$).

**Chromatic Subsampling** Since chromatic information is much less noticeable to the human eye, the two chromatic channels $C_o$ and $C_g$ can be subsampled. In our proposal, we downscale them to half size, although different scales can be chosen (or no subsampling at all) if the result needs to be of higher quality.

**DCT conversion** The main step in the proposed compression is the transition to the frequency domain through the Discrete Cosine Transform. It will be explained in more detail in Section 3.4.5.

**Quantization** Each converted DCT coefficient is quantized according to its importance for overall image quality. Because the human eye is less sensitive for high frequencies,
these values are quantized more heavily, while the low frequency components are kept at a high precision. These quantifying values are read from a table containing the quantization size for each frequency coefficient. This table is called the Quantization table, or Q-table.

**LOD separation** Since every mipmap needs to be decompressed and read separately from the rest, the data needed for each Level Of Detail (mipmap) needs to be separated before compression layers are applied.

**RLE encoding** Because of quantization, most high-frequency components are reduced to zero, making Run-Length Encoding (RLE) an effective way of reducing the data size without loss of information.

**Huffman encoding** Lastly, an entropy-based compression method is applied as a last way of compressing the data. The effectiveness of the decompression relies on the construction of a Huffman tree[17], which needs to be created and read for each decompression operation. Using separate trees for each LOD increases compression performance on the coefficients, but adds the tree data to the storage overhead.

### 3.4.4 YCC conversion

For the proposed scheme, the YCoCg transformation was chosen. It separates the RGB data into a separate Y channel, containing the luminosity of the pixel, and two chromatic channels, green (Cg) and orange (Co). All calculations are done with integer arithmetic, and the three output channels are of the same word size as the input R, B and G channels. This would make it possible to perform in-place transforms. Although the current implementation does not use this feature, this makes it possible to reduce the memory footprint of the streaming system even further. The run-time memory footprint becomes an important issue when the system is implemented in an environment with very limited memory, such as the small coprocessing elements called Synergistic Processing Elements (SPE) on the Cell chip used by the PS3 gaming console.

### 3.4.5 DCT conversion

The Y, Co and Cg channels are subdivided into blocks of \(8 \times 8\) pixels and transformed into an \(8 \times 8\) DCT coefficient matrix. For the 1D DCT conversion and its inverse, the DCT II and DCT III transform are commonly used, and can be seen in Equation 3.3 and Equation 3.4 respectively. In order to transform image data, the transformation is easily expanded to two dimensions. More general information about DCT transform and its various versions can be found in Appendix B.

\[
X_k = \sum_{n=1}^{N-1} x_n \cos \left[ \frac{\pi}{N} n \left( k + \frac{1}{2} \right) \right] \quad k = 0, \ldots, N - 1. \tag{3.3}
\]

\[
X_k = \sum_{n=0}^{N-1} x_n \cos \left[ \frac{\pi}{N} (n + \frac{1}{2}) \left( k + \frac{1}{2} \right) \right] \quad k = 0, \ldots, N - 1. \tag{3.4}
\]
DCT II and DCT III can be seen as series inner products of the constant source $x$ of length $N$, and $N$ cosine functions of increasing frequency. We can rewrite both equations into more general matrix multiplications where the DCT II equation is expressed as an $8 \times 8$ matrix $V$ while the DCT III conversion is rewritten as its inverse $\hat{V}$. The resulting matrix values for the transforms are described in Equation 3.5 and 3.6.

$$V_{n,k} = \cos\left[\frac{\pi}{N} n \left( k + \frac{1}{2} \right)\right] \quad k = 0, \ldots, N - 1 \mid n = 0, \ldots, N - 1.$$ (3.5)

$$\hat{V}_{n,k} = \cos\left[\frac{\pi}{N} \left( n + \frac{1}{2} \right) \left( k + \frac{1}{2} \right)\right] \quad k = 0, \ldots, N - 1 \mid n = 1, \ldots, N - 1.$$ (3.6)

$$\hat{V}_{0,k} = \frac{1}{2} \quad k = 0, \ldots, N - 1.$$ (3.7)

The similarity between both matrices is already clear, but by adding scale factors and offsets to the $x_0$ term, the two matrices of DCT II and DCT III can be made orthogonal, making them each others transpose. This is covered in more detail in Appendix B. The eight cosine functions stored $V$ and $\hat{V}$ are shown in Figure 3.7. It clearly shows eight cosine functions of increasing frequency but constant amplitude.

![Figure 3.7: The eight base cosine functions that are used in the DCT.](image.png)

As with any matrix, the DCT transform can be seen as a base transform: the input data is rewritten as a summation of base (cosine) functions. Each DCT coefficient corresponds to an orthogonal cosine function, and the input can in turn be decomposed into a weighed sum of these (cosine) functions. In the 2D version of DCT, the base functions effectively become two dimensional, and consist of all 64 combinations of the original 8 cosine functions. Figure 3.7 shows these 64 base functions, ordered similar to their coefficients in the result coefficient matrix.
3.4.6 2D DCT

Doing the 64-coefficient transform as a single 3D matrix transformation is a possibility, using the 64 orthogonal bases seen in Figure 3.7. Luckily, the two dimensional version of DCT is a separable product, as seen in Equation 3.8.

\[
X_{k_1,k_2} = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x_{n_1,n_2} \cos \left( \frac{\pi}{N_2} \left( n_2 + \frac{1}{2} \right) k_2 \right) \cos \left( \frac{\pi}{N_1} \left( n_1 + \frac{1}{2} \right) k_1 \right)
\]

(3.8)

The created orthogonal matrix \( M \) can be applied in a 2D DCT transform on a \( 8 \times 8 \) input array by first transforming the eight rows of input data, and then applying the matrix again over the resulting eight columns, similarly to the row-column algorithm used in Fast Fourier algorithms. The end result is a \( 8 \times 8 \) coefficient matrix. We will see in Section 3.5 that we can modify the base functions in the inverse matrix to create mipmaps of varying sizes using subsets of the DCT input matrix, enabling us to set up a progressive dataflow.

3.5 Mipmap construction from DCT data

In Section 3.4.5 it has been shown that the matrix \( V \) can be used to perform discrete cosine transforms while its transpose can be used to perform an inverse DCT, by using the row-column algorithm, this can be extended into a 2D DCT transform. This establishes a workable conversion between 2D image- and DCT spaces.

This far, the DCT conversion is set up to convert \( n \times n \) blocks of input data into \( n \times n \) DCT coefficients and back again, where each coefficient influences all inputs and vice versa. At first glance, this schema does not support progressive construction: all coefficients are needed to recreate the image, and the image will always be generated at its original resolution.

In order to enable some sort of progressive mipmap generation, we need to enable the creation of mipmaps using a subset of the full DCT coefficient data. Since the DCT conversion works on separate 2D blocks of data, we can focus on creating scaled versions of individual blocks. In the case of \( 8 \times 8 \) blocks, we can aim to create \( 8 \times 8, 4 \times 4, 2 \times 2 \) and \( 1 \times 1 \) blocks. Applying this to an entire image will result in the effective creation of the four highest mipmap layers.

Ignoring the progressive aspect for now, in order to create a smaller version of the original full resolution texture, the easiest way is to simply create the texture and subsample it using a simple mean filter. This filter function splits the function into sets \( n \) values and outputs the mean of each set of values. Using \( n = 2 \), this halves the resolution. It should be noted that this filter function preserves multiplication and addition, thus making it a linear \( \mathbb{R}^n \rightarrow \mathbb{R}^\frac{n}{2} \) function. Subsampling using a 2D mean filter can be used iteratively to create the entire mipmap chain from the highest layer, however this method still requires the full texture to create any mipmap, making a progressive approach impossible.
Looking again at the described 2D DCT transform, the $8 \times 8$ block can be reconstructed by simply using the inverse $\hat{V}$ matrix, but it is not directly clear from the definition how smaller images can be created. Instead, looking at the transform as a base decomposition provides a more useful insight.

Similarly to 1D DCT, the original 2D image is reconstructed by using the (weighed) sum of the input 2D base functions. Since the linear filter mentioned above is a linear function, it follows that the filter function can be (pre) applied to these base functions. Furthermore, as all the 64 2D base functions are products of the DCT base functions, instead of using a $n \times n$ subsampling filter on the 2D result, a one dimensional filter can be used on the original 1D base functions. Applying a mean filter to the 8-length the base functions halves them to 4; applying the same filter multiple times creates base functions of length of 4, 2 and 1.

Now we have a more efficient way of generating mipmap; instead of filtering the image itself, we can filter the DCT base functions. Modifying the base transform to use these filtered bases results in smaller output matrices which in term can be used to create mipmap.

Going back to the matrix-notation of the DCT transform, this effectively results the inverse matrices $\hat{V}_4$, $\hat{V}_2$, $\hat{V}_1$ with dimensions $8 \times 4$, $8 \times 2$ and $8 \times 1$ respectively. Although this is more efficient than the original box filter, it still requires all DCT data to create a mipmap since there are still eight base functions to process. In order to make this a functionally progressive method, we need to alter our inverse transform further to use only a subset of the full coefficient data.

### 3.5.1 Progressive construction

The only straight-forward way of making this approach progressive from a data flow standpoint, is to enable the creation of mipmap from a subset of DCT coefficients. This way, the DCT coefficients can be gathered and resorted with the coefficients needed for the smallest mipmap layer first, and the coefficients needed for the full resolution layer last.

Looking at the unfiltered base functions, all of the coefficients have a similar impact on image quality; meaning all coefficients are needed to create the original image.

$$X_7 = \sum_{n=0}^{N-1} x_n \cos \left( \frac{\pi}{N} \left( n + \frac{1}{2} \right) \left( 7 \cdot \frac{1}{2} \right) \right)$$  \hspace{1cm} (3.10)

However, filtering the base functions to create the mipmap effectively removes information. The amount of samples of the original function are halved, meaning that the Nyquist frequency is effectively halved. Looking at the most high-frequency base function as shown in Equation 3.10, it shows that the function has a highest frequency of 3.5, while the sampling rate is at most 4.0, which gives a Nyquist Frequency of 2.0. Figure 3.8 shows the original cosine samples, followed by the 2x-mean filtered samples. Of the filtered samples, the first four still contain all frequency information (green), but higher frequencies are aliased (red). Even though each coefficient is needed to create the original image, the higher frequencies get effectively removed when the base functions get filtered. From this follows that coefficients representing these higher frequencies become irrelevant and can be set to zero with only a minimal influence on the output.
Removing these coefficients from the calculations means that $\hat{V}_4$ becomes a square $4 \times 4$ matrix, and that only the first four coefficients are used. In the 2D DCT, the $4 \times 4$ input matrix is created from the first four columns of the first four rows of the original DCT coefficient matrix. We can now effectively create a mipmap using only a subset of the original DCT value, laying the basis for a progressive compression scheme.

The same principle can be expanded to the lower mipmap levels: $V_2$ and $V_1$ are reduced to square matrices, where $V_1$ only consists of a scale to the DC component of each DCT coefficient block. This all results in a usable progressive reconstruction, where the four top mipmap layers of a texture can be streamed. The fourth mipmap is created by using only first coefficient of the data, representing the constant component of the original $8 \times 8$ block. To create the third, another 3 DCT samples are needed, adding the first three gradients. The second mipmap layer requires another 12 and the top mipmap requires all 64 samples. The "valid" coefficients for each mipmap layer are shown in Figure 3.9. The lower mipmap layers that cannot be streamed in this manner will not be streamed. Instead they will be stored as DXT data and loaded from disc using traditional methods. These static mipmap layers make up less than one percent of the original texture size.

### 3.5.2 Coefficient layout

Before the compression layers can be applied on top of the DCT coefficient data, the layout must be considered. The JPEG compression scheme specifies the so-called zigzag order of coefficients, which stores the 64 coefficients in diagonals to order to optimize numeric correlation between coefficients. However, this does not support the progressive properties needed for mipmap generation: all coefficients that are needed to construct the various
3. Design

matrices mipmaps must be stored in-order: the first part of the data must contain the first coefficient of all DCT blocks to generate mipmap 3, which is one-eighth of the full resolution, while the second part must contain the first three additional coefficients to reconstruct the next mipmap and so on for each of the four sets of coefficients. A new ordering is required that incorporates this ordering, referenced in this thesis with the name jzag order. The specific order is shown in figure 3.10.

In addition to coefficient ordering, all three color channels must be interleaved in such a way that all coefficients to recreate a complete mipmap in RGB space are stored together. Because the chromaticity channels are subsampled, the Y channel needs four DCT blocks for each one block of Co and Cg data. The basic layout is shown in Figure 3.10.

Figure 3.9: The four top mipmaps and their valid coefficients

Figure 3.10: The global layout of a streamable mipmapped texture, showing the four sets of coefficients used to generate the four progressive mipmaps.
Loosely following the JPEG standard, RLE and Huffman are chosen as compression methods. RLE has no further impact on the data layout and can be added transparently to the compression stack, but Huffman compression requires additional data in the form of a Huffman tree that is created from the encoded data. The tree can be generated for each layer, each channel, each level of detail, or even each scan line, making compression more or less effective in exchange for additional overhead. More information about the details of this choice can be found in Section 4.5. In the current design, a separate tree is generated for every level of detail. This way each level of detail can be decompressed and streamed independently, while making use of the correlation within the coefficient data.

### 3.5.3 Decompression and framing

Although the layout now contains all necessary information to create and progressively stream textures, the internal sorting of the data within each level of detail needs to be addressed and to be able to do this, the frame size must be determined. A frame is the minimal data set that can be processed by the pipeline. Bigger frames have less overhead but could require too much memory to be processed at once. Smaller frames are easier to process but can result in bad compression characteristics and computational overhead.

![Figure 3.11: The finished data layout. Although it is framed on a scan line basis, other frame sizes can be used. Note that the two scan lines containing luminosity info are twice the size as the chromatic ones, creating a 1:1:4 grouping.](image)

Because the chromatic channels $C_o$ and $C_g$ are subsampled, four $Y$ coefficient matrices are grouped with each single $C_o$ and $C_g$ matrix to make sure all data can be processed into raw RGB output. This results in a $16 \times 16$ RGB output at full resolution, and a $2 \times 2$ pixel output for the fourth mipmap level. This forms the first lower bound of the frame size: six DCT blocks. Because the whole decompression system scales linearly with output size (see Section 4.3 for further details) The frame size must be known at conversion time in order to make sure each data frame can be decompressed independently. For instance, RLE-
encoding must be "flushed" so no token spans a data block boundary. This will be further
discussed in Chapter 4. For the remainder of the design, we will take a full scan line as data
block size. This means the coefficients of each channel need to be interleaved on a scan line
basis within each of the four mipmap coefficient blocks. The full data layout can be seen in
Figure 3.11.

3.6 DXT Encoding and performance

After the texture data has been completely decoded and recomposed into RGB mipmaps,
it needs to be rendered by the hardware. However, rendering raw RGB data is extremely
inefficient in both rendering speed and memory usage. In order to have a viable design, the
raw RGB data has to be recoded into DXT data in order to be rendered. This recoding is
done as the final step in the streaming pipeline. There are many heuristics to create a DXT
image, but most of them are iterative and time consuming and any implementation will be
a trade-off between quality and speed.

This completes the pipeline design, Chapter 4 will detail the various stages, their im-
plementation and the various performance challenges that were encountered during imple-
mentation.
Chapter 4

Implementation

In this chapter, the implementation of the various steps of the compression pipeline will be discussed. This chapter will go through the pipeline in the order of conversion, starting with the original raw 24-bits RGB data and ending with the compressed streamable data. We will then discuss the DXT transcoding and texture update steps. Finally we look at the full pipeline and overall performance. Figure 4.1 shows an overview of the various stages of the streaming pipeline.

![Diagram of the streaming pipeline](image)

Figure 4.1: The main stages of the entire streaming and compression pipeline.
4. IMPLEMENTATION

4.1 YCC transform

The first step of our compression pipeline is the conversion of the original RGB values into a color space that is better suited for compression, while keeping the conversion complexity at a minimum. For these reasons we are using the YCoCg color space, which has a single luminance channel and two separate color channels Co and Cg, roughly corresponding to the green and orange colors respectively. The conversion matrices between the two spaces is originally [11] defined as:

\[
\begin{bmatrix}
Y \\
Co \\
Cg
\end{bmatrix} =
\begin{bmatrix}
1/4 & 1/2 & 1/4 \\
1/2 & 0 & -1/2 \\
-1/4 & 1/2 & -1/4
\end{bmatrix}
\begin{bmatrix}
R \\
G \\
B
\end{bmatrix}
\] (4.1)

\[
\begin{bmatrix}
R \\
G \\
B
\end{bmatrix} =
\begin{bmatrix}
1 & 1 & -1 \\
1 & 0 & 1 \\
1 & -1 & -1
\end{bmatrix}
\begin{bmatrix}
Y \\
Co \\
Cg
\end{bmatrix}
\] (4.2)

The conversion can be done completely in integer math and has improved compression characteristics over similar spaces such as YCbCr [12]. However, the compression is lacking one characteristic that could simplify our streaming pipeline a great deal: the resulting values do not fit the original 24-bit of the original RGB space. An altered compression method was designed, which has better RGB reversibility: YCoCg-R. This has been designed to create a lossless compression that stores the original RGB values into 26 bits. The conversion between RGB and YCoCg-R is defined as follows:

\[
\begin{align*}
Co &= R - B \\
t &= Y - (Cg >> 1) \\
G &= Cg + t \\
Cg &= G - t \\
Y &= t + (Cg >> 1) \\
R &= Co + B
\end{align*}
\] (4.3)

Using this the starting point of our implementation, we can start to look at the specifics of this transform. It can be clearly seen that the values Co and Cg are outside the input data range: we define our RGB input as three 8-bit unsigned integer values, so Co can range between -255 and 36.

\[
Y = \frac{B + ((R - B) >> 1)) + ((G - (B + ((R - B) >> 1))) >> 1)}{2} = Y = B + R >> 1 - B >> 1 + G >> 1 - B >> 1 - R >> 2 + B >> 2
\] (4.4)

If we then treat the shifts as divisions, this translates to:

\[
Y = B + R/2 - B/2 + G/2 - B/2 - R/4 + B/4 \quad Y = B/4 + G/2 + R/4
\] (4.5)

Which puts Y in the same 8-bit range as the RGB parameters. However, we cannot simply treat shifts as divisions since there is a loss of precision after each shift and the operation is not distributive.

Alternatively we can do a range analysis on each of the variables by evaluating the maximum and minimum of each expression. Given that Co has a range between -255 and
255, the range of $t$ is that of 0 for $(R, B) = (0, 0)$ to 255 for $(R, B) = (255, 255)$. This in term gives $C_g$ the range of -255 for $(G, t) = (0, 255)$ and -255 for $(G, t) = (255, 0)$. For $Y$, the range becomes the result of two dependent variables $C_g$ and $t$; calculating all input ranges gives a minimum of 0 for $(G, t, C_g) = (0, 0, 0)$ and a maximum of 255 for $(G, t, C_g) = (255, 255, 0)$.

We can see that with the YCoCg-R transform, the $Y$ component can be stored as an unsigned 8-bit value, while $C_o$ and $C_g$ should be stored as signed 9-bit values. We will see later on that we can sacrifice some color information to store $C_o$ and $C_g$ in 8-bit values with a minimum of color loss.

### 4.1.1 Initial implementation

The initial implementation of the transform is shown below, along with the two representations of a YCoCg RGB color structure. Note that all YCoCg data is stored in 16-bit short values because of the 9-bit signed chromaticity channels.

#### Listing 4.1: RGBA to YCoCg.

```c
struct YCoCg
{
    YCoCg(short inY, short inCo, short inCg) :
        Y(inY), Co(inCo), Cg(inCg) {}
    short Y;
    short Co;
    short Cg;
};

struct RGBColor
{
    RGBColor(short inR, short inG, short InB) :
        mRed(inR), mGreen(inG), mBlue(inB) {}
    byte mRed;
    byte mGreen;
    byte mBlue;
};

YCoCg RGBToYCoCg(byte inRed, byte inGreen, byte inBlue)
{
    short co = inRed - inBlue;
    short t = inBlue + (co >> 1);
    short cg = inGreen - t;
    short y = t + (cg >> 1);
    return YCoCg(y, co, cg);
};

RGBColor YCoCgToRGB(uint inY, uint inCo, uint inCg)
{
    short t = inY - (inCg >> 1);
    short g = inCg + t;
    short b = t - (inCo >> 1);
    short r = inCo + b;
    return RGBColor(r, g, b);
}
```

Since our compression pipeline is not lossless and the output color depth of DXT is only 16-bits, we can sacrifice some color depth in order to lose the two additional chromacity bits. However, this is not as straight-forward as would be expected: simply shifting out the least significant bits of the chromaticity channel will result in incorrect reconstruction of the original colors as seen in Figure 4.2. In practice, it results in negative color values where round off errors cause the output RGB values to fall just outside the 8-bit range. This
4. IMPLEMENTATION

Figure 4.2: Simply shifting out bits results in corruption on some input values.

can be fixed by adding additional saturation steps into the deconstruction, but this results in additional performance penalties. Instead we can look again more closely to the transform itself and what information the least significant bits (LSB) contain. Since we are interested in keeping the inverse function as fast as possible, we can try to solve the problem in the off-line conversion stage. If we can assure that the LSB of the chromaticity channels always holds zero, we can safely remove it from our output. Looking at the transform equation 4.3 again, we can see that the LSB of Co holds the result of an addition and therefore the bit-wise Exclusive Or (XOR) of the LSB of both operands, represented with the caret(ˆ) operator in C++. If we can match the LSB of Red and Blue, the LSB will always be zero. We can do the same to Cg, which holds the result of shifted additions and subtractions. The resulting function is given below. gSetLowBit sets the LSB of its first operand to true or false depending on its second boolean operand.

Listing 4.2: Prepare RGB data for YCoCg compression.

```cpp
void PrepareForPack(RGBColor& inColor)
{
    // manipulate input values so the
    // converted values can be packed in 8.8.8
    // cg0 = blue0 ^ red1 ^ blue1 ^ green1 ^ green0
    // co0 = red0 ^ blue0
    gSetLowBit(inColor.mBlue, inColor.mRed&1);
    bool t_bit_value = ((inColor.mBlue((inColor.mRed^inColor.mBlue)>>1))&1) == 1;
    gSetLowBit(inColor.mGreen, t_bit_value);
}
```

Applying this to the input data before it is being converted results in a proper image this time, although with LSB changes to the green and blue channel. The above function is very simplistic and modifies the red and green channels. However, since this is done during the off-line phase, the algorithm can be made more complex to minimize image distortion by choosing which channel to adjust according to gamma ramp or dithering patterns. Below is a more complex version that minimizes the amounts of bits changed while keeping the green channel intact since this is visually the most prominent channel.

Listing 4.3: More complex version of the prepare algorithm.

```cpp
void PrepareForPack(RGBColor& inC)
```
YCC transform

```cpp
bool bit_value_cg = ((inC.mGreen^inC.mBlue^((inC.mRed^inC.mBlue)>>1))&1) == 1;
bool bit_value_co = ((inC.mBlue^inC.mRed)&1) == 1;

if (bit_value_cg && bit_value_co)
{
    // we can fix both channels by only changing blue
    gFlipLowBit(inC.mBlue);
}
else if (bit_value_cg)
{
    // fix cg by changing blue, the patch co by changing red
    gFlipLowBit(inC.mBlue);
gFlipLowBit(inC.mRed);
}
else if (bit_value_co)
{
    // co fix by changing red
    gFlipLowBit(inC.mRed);
}

// reevaluate after changes
bit_value_cg = ((inC.mGreen^inC.mBlue^((inC.mRed^inC.mBlue)>>1))&1) == 1;
bit_value_co = ((inC.mBlue^inC.mRed)&1) == 1;
// this should always turn all msb to zero
assert(bit_value_co == false && bit_value_cg == false);
```

Figure 4.3: Various methods of changing input data to reduce image size. Left shows the more complex algorithm, right shows the original solution. The images show the difference between the original and the adjusted image magnified by 32.

4.1.2 Optimization and performance

The next step is to see what the performance is of the conversion code, and how we might be able to increase it. Because this step has little dependencies on outside data, the first performance test simply runs is to run the function over several four megapixel images several times, and measures the performance of encoding and decoding the data to-and-from YCoCg space. The conversion is run using both 16-bit and 32-bit versions of the conversion logic.
4. IMPLEMENTATION

Figure 4.4: A hundred runs of converting a four megapixel image to YCoCg space and back. The horizontal axis denotes milliseconds.

It seems that the integer version of the conversion logic is much faster, which is as expected since all arithmetic is done natively on 32-bits, incurring additional overhead on non int32 operations. There are a few optimizations that can be done such as tweaking compiler settings or hand-optimizing the generated assembly code, another solution is the usage of compiler SIMD (Single Instruction Multiple Data) intrinsics, which are special functions known to the compiler which make use of specialized CPU instructions to perform vectorized calculations. Even though these intrinsics are somewhat platform specific, they do create a more portable way of creating CPU-side optimizations. Seeing how the conversion is done on separate, small units of data, the algorithm can benefit from using SIMD instructions which available on most x86-compatible and “Power-Architecture”-based CPUs.

In order to leverage the SIMD instructions, the input data needs to be supplied in larger sets so multiple data sets can be converted simultaneously. The SIMD instructions run on 128-bit registers meaning we can store 16 channels into one register. Starting with the YCoCg conversion, we redefine our input as a set of three 128-bit values. Together these three registers contain 16 pixels of RGB data that will be converted in parallel. Since our intermediate values will be larger than 8 bits, and because most SIMD instructions work only on 16-bit values, we need to unpack the 16 pixels into two sets of 8, so we can run 16-bit SIMD operations, 8 data sets at a time. Figure 4.5 shows the dataflow of the SIMD-enabled YCoCg conversion. In order to use SIMD instructions, all data needs to be packed into __m128i registers, these are the integer registers that are used by the SIMD instruction set. Listing 4.4 shows the in- and output formats of the SIMD-enabled YCoCg conversion. Listing 4.4 shows the layout of SIMD-enabled C++ structures.

Listing 4.4: SIMD-enabled YCoCg data structures.

```c
struct DIBColor16
{
  union
```
Figure 4.5: The data flow of SIMD-enabled conversion. The input data is unpacked, processed in two parts and repacked again.

```c
struct {
  __m128i mRed16;
  __m128i mBlue16;
  __m128i mGreen16;
};
struct {
  byte mRed[16];
  byte mBlue[16];
  byte mGreen[16];
};

struct YCoCg16 {
  union {
    struct {
      __m128i mY16;
      __m128i mCo16;
      __m128i mCg16;
    };
    struct {
      byte mY[16];
      byte mCo[16];
      byte mCg[16];
    };
  };
}
```

The code of packing and unpacking the data is shown in Listing 4.5. The _mm_unpacklo_epi8 and _mm_unpackhi_epi8 intrinsics pad and unpack the 16x8-bit data into 16-bit words as
shown in Figure 4.5. Since the 16-bit values are padded with zeroes, the _mm_set1_epi8(0) is used to generate a zero filled register for the second parameter. The _mm_pack intrinsics repack the 16-bit words back into 8-bits. Because the packing intrinsics also saturate the values, it is vital that we use the signed _mm_pack_epi16 version for the chromaticity channels Co Cg and the unsigned _mm_packus_epi16 version for luminance channel Y. The RGBToYCoCg16Sub subroutine contains the actual conversion code, which is shown in Listing 4.6.

Listing 4.5: Packing code for SIMD-enabled YCoCg encode.

```
inline void RGBToYCoCg16(YCoCg16& outYCoCg, const RGBColor16& inColor)
{
    m128i blue_low = _mm_unpacklo_epi8(inColor.mBlue16, _mm_set1_epi8(0));
    m128i blue_hi = _mm_unpackhi_epi8(inColor.mBlue16, _mm_set1_epi8(0));
    m128i red_low = _mm_unpacklo_epi8(inColor.mRed16, _mm_set1_epi8(0));
    m128i red_hi = _mm_unpackhi_epi8(inColor.mRed16, _mm_set1_epi8(0));
    m128i green_low = _mm_unpacklo_epi8(inColor.mGreen16, _mm_set1_epi8(0));
    m128i green_hi = _mm_unpackhi_epi8(inColor.mGreen16, _mm_set1_epi8(0));
    m128i Yj, Coj, Cgj;
    m128i Yhl, Cohl, Cghl;
    // do the actual conversion in a subfunction
    RGBToYCoCg16Sub(red_hi, green_hi, blue_hi, Yhl, Cohl, Cghl);
    outYCoCg.mY16 = _mm_packus_epi16(Yj, Yh);
    outYCoCg.mCo16 = _mm_pack_epi16(Coj, Coj);
    outYCoCg.mCg16 = _mm_pack_epi16(Cgj, Cgj);
}
```

Now that all values are unpacked into 16 bits, we can use the same logic as used in the original version of the code. _mm_srai_epi16 is a signed shift right on each 16-bit word while _mm_sub_epi16 and _mm_add_epi16 perform adds and subtracts on each 16 bit word respectively.

Listing 4.6: SIMD-enabled YCoCg encode.

```
inline void RGBToYCoCg16Sub(const _m128i& inRed, const _m128i& inGreen, const _m128i& inBlue, _m128i& outY, _m128i& outCo, _m128i& outCg)
{
    // co = inRed - inBlue;
    _m128i co = _mm_sub_epi16(inRed, inBlue);
    // t = inBlue + (co >> 1);
    _m128i co_div2 = _mm_srai_epi16(co, 1);
    _m128i t = _mm_add_epi16(inBlue, co_div2);
    // cg = inGreen - t;
    _m128i cg = _mm_sub_epi16(inGreen, t);
    // y = t + (cg >> 1);
    _m128i cg_div2 = _mm_srai_epi16(cg, 1);
    outY = _mm_add_epi16(t, cg_div2);
    // shift out zero bits (always zero due to pre-pack stage)
    outCg = _mm_srai_epi16(cg, 1);
    outCo = _mm_srai_epi16(co, 1);
}
```
The next step is to rewrite the decoder to SIMD instructions. This is done in a similar fashion to the encoder. However, there are a few small changes to account for some missing functionality when dealing with signed values. The unpacking of 8-bit values into 16 bit words is done using the the `mm_unpacklo_epi8` and `mm_unpackhi_epi8` intrinsics, but these do not support signed values; the higher eight bits can only be padded with constant values. This was correct for the encoder, where all inputs were unsigned, however the Co and Cg values in the encoder are signed and need to be sign-extended into the 16-bit words. In order to fix this, we use the signed shift functionality to fill the higher byte of each word with the 7th bit of the packed value, effectively doing a signed extension of the 8-bit value. This is done by first shifting eight bits to the left, putting the 7th bit at the MSB position. We then do a signed shift right, extending the MSB across the entire high byte. We can combine this with the shift left that was necessary to extend the chromaticity values to 9-bit by shifting back one position less. The code of the packing function and the decoding subroutine can be found in Listing 4.7.

Listing 4.7: SIMD-enabled YCoCg decode.
Running the same test with SIMD instructions shows a significant speed increase, as can be seen in Figure 4.6.

Figure 4.6: A hundred runs of converting a four megapixel image to and from YCoCg space. The vertical axis denotes milliseconds.

Note that the packing logic is a substantial part of the functional logic in the decoding phase. If the YCoCg data is streamed using 16-bit values, the decoding speed can be increased substantially. Since the YCoCg code is streamed in from the higher compression layers (RLE in our current scheme), we can choose to adjust the RLE decoder to output its tokens in 16-bit aligned values. This will double the memory needed to buffer between the two layers, but increase the speed of decoding. As a final test result, we add the decoding speed without packing and unpacking overhead. Running the decoder again without packing enabled shows no real improvement over the original SIMD version. This is probably due to memory bandwidth taking over as the primary bottleneck during the decoding process.

4.2 Chromatic Subsampling

The YCoCg decoding step dictates a minimum frame size of 16 pixels to neatly fill in a full 128-bit register. The next step in the pipeline is chromatic subsampling, where we
Chromatic Subsampling

subsample the Co and Cg channels before compressing it on a per-channel basis. The subsampling itself is straightforward: we average the values of a 2x2 pixel block of the Co and Cg channels. Combining this with the YCoCg conversion step creates a dataflow with an input that consists of a block of 16x16 pixels in YCoCg space, and an output of three separate channels of 8x8 and 16x16 blocks. Figure 4.7 shows the dataflow for the combined YCoCg conversion and chromatic subsampling.

Figure 4.7: Detailed data flow between YCC conversion, Chromatic Subsampling and DCT transform. Note that the chromatic subsampling also separates the data into separate channels.

4.2.1 Initial Implementation

The subsampling code is listed in Listing 4.8. In the code, we assume that the image width is a multiple of 16, so block and index calculations can be simplified. Figure 4.8 shows the performance of this initial implementation.

Listing 4.8: Chromatic Subsampling.

```c
inline void ChromaSub_NonTiled(
    const YCoCg16* in_channel,
    uint8* Y_channel,
    int8* Co_channel,
    int* Cg_channel)
{
    uint w = GetImage().GetWidth();
    uint h = GetImage().GetHeight();
    for (uint x = 0; x < w/2; x++)
        for (uint y = 0; y < h/2; y++)
            {
                // calculate input 128-bit block and index from pixel offset
                int in_offset = (x+2)*(y+2)*w;
                int out_offset = x+y+(w/2);
```
int blockA = in_offset/16; // Block containing even scanline
int idx = in_offset - (blockA*16); // Index into block
int blockB = blockA + w/16; // Uneven scanline, scanline is a multiple of 16

// Copy Y
Y_channel[in_offset] = in_channel[blockA].mY[idx];
Y_channel[in_offset+1] = in_channel[blockA].mY[idx+1];
Y_channel[in_offset+w] = in_channel[blockB].mY[idx];
Y_channel[in_offset+w+1] = in_channel[blockB].mY[idx+1];

// Subsample Co
int co_accum = in_channel[blockA].mCo[idx];
co_accum += in_channel[blockA].mCo[idx+1];
co_accum += in_channel[blockB].mCo[idx];
co_accum += in_channel[blockB].mCo[idx+1];
Co_channel[out_offset] = co_accum / 4;

// Subsample Cg
int cg_accum = in_channel[blockA].mCg[idx];
cg_accum += in_channel[blockA].mCg[idx+1];
cg_accum += in_channel[blockB].mCg[idx];
cg_accum += in_channel[blockB].mCg[idx+1];
Cg_channel[out_offset] = cg_accum / 4;


Figure 4.8: Initial performance of the subsampling code in milliseconds per run over a four megapixel image; the results are a magnitude worse than the YCoCg pass.

4.2.2 Tiling

Although this code is straightforward, the performance seems to be lacking, as seen in Figure 4.8. Even though the calculations are simpler than the YCoCg calculation, the performance is worse. The most obvious cause is the cache unfriendliness of the code. The subsampling code is reading and writing memory in a non-linear fashion, causing cache misses. To remedy this, the image data should be stored in a tiled layout: instead of storing the image scan-lines sequentially in memory, the pixel data should be stored into blocks (tiles), grouping neighboring pixels together in memory. One of the big challenges of efficient tiling is finding the optimal tile size; because we will be using DCT compression in the next stage, a tile size of 8x8 seems logical. However, since we are subsampling the chromaticity channels, the tile size should be doubled, resulting in an output tile size of 8x8.
Chromatic Subsampling

for chromaticity channels, and 16x16 of the luminance channel Y. The adjusted data flow can be seen in Figure 4.10. It also shows the layout of a single SIMD-enabled tile before YCoCg conversion. Figure 4.9 shows an example of a tiled layout: a 16x16 image is stored linearly on the left and tiled on the right; the tilesize shown is 8x8.

Figure 4.9: The data layout of a block of data before- and after tiling. The numbers show the order in which the data blocks are stored in memory.

Figure 4.10: The tiled image data in the YCoCg-Chromatic Subsampling dataflow
4. IMPLEMENTATION

Using tiled textures does complicate the code somewhat. First we must define new structures to hold our tiled data: Co8x8, Cg8x8 and Y16x16. These new structures, along with the function that performs the subsampling is shown in Listing 4.9. We input the pre-tiled input data as a flat array of YCoCg16 data as it is coming from the YCoCg16 conversion, which is not affected by the tiling as it treats the data as a one dimensional stream.

Listing 4.9: Tiled Chromatic Subsampling.

```cpp
template<typename T>
struct Tile8 {
    union {
        m128i mData128[4]; // 16x16
        T mData[8][8];
    }
};

template<typename T>
struct Tile16 {
    Tile8<T> mSubTile[2][2];
};

// tiled, per-channel YCoCg
typedef Tile8<uint8> Y8x8;
typedef Tile8<int8> Co8x8;
typedef Tile8<int8> Cg8x8;
typedef Tile16<uint8> Y16x16;
typedef Tile16<int8> Co16x8;
typedef Tile16<int8> Cg16x8;

inline void ChromaSubTiled(const YCoCg16* in_channel, Y16x16* Y_channel16x16, Co8x8* Co_channel8x8, Cg8x8* Cg_channel8x8) {
    uint tile_count = GetImage().GetWidth() * GetHeight() / 256;
    for (uint c = 0; c < tile_count; c++) {
        for (uint s = 0; s < 4; s++) {
            Y_channel16x16[c].mData128[s*4+0] = in_channel[c*16+s*4+0].mY16;
            Y_channel16x16[c].mData128[s*4+1] = in_channel[c*16+s*4+1].mY16;
            Y_channel16x16[c].mData128[s*4+2] = in_channel[c*16+s*4+2].mY16;
            Y_channel16x16[c].mData128[s*4+3] = in_channel[c*16+s*4+3].mY16;
        }
        for (int y = 0; y < 8; y++)
            for (int x = 0; x < 8; x++) {
                int offset = (x*2)+(y*2)*16;
                int block = offset/16;
                int idx = offset - (block*16);

                int co_accum = in_channel[c+16+block].mCo[idx];
                co_accum += in_channel[c+16+block+1].mCo[idx+1];
                co_accum += in_channel[c+16+block+1].mCo[idx+1];
                co_accum += in_channel[c+16+block+1].mCo[idx+1];
                Co_channel8x8[c].mData[y][x] = co_accum / 4;

                int go_accum = in_channel[c+16+block].mGc[idx];
                go_accum += in_channel[c+16+block+1].mGc[idx+1];
                go_accum += in_channel[c+16+block+1].mGc[idx+1];
                go_accum += in_channel[c+16+block+1].mGc[idx+1];
                Cg_channel8x8[c].mData[y][x] = go_accum / 4;
            }
    }
}
```
Chromatic Subsampling

Looking now at the performance, we see a big improvement, speeding up the process almost 10x(!), as shown in Figure 4.11.

Figure 4.11: Comparing tiled and non-tiled subsampling.

4.2.3 Performance and Optimization

Finally, we look into optimizing the subsampling using SIMD-instructions, similar to the approach used in YCoCg conversion. One of the issues encountered when using SIMD, is the fact that the available averaging instructions _mm_avg_epu8, _mm_avg_epu16 do not support signed values. I have used two approaches to remedy this. The first approach is to manually do all signed averaging where possible, the other approach is to offset the Co and Cg values by 128 into the [0, 255] range, making them effectively unsigned values. Figure 4.12 shows the basic steps followed by both SIMD implementations. The functions themselves can be seen in Listing 4.10

Listing 4.10: SIMD Enabled Chromatic Subsampling.

```c
inline void gChromaSubSampleA(
    _m128i & outScanline,
    const _m128i & inScanlineA,
    const _m128i & inScanlineB,
    const _m128i & inScanlineC,
    const _m128i & inScanlineD)
{
    // divide by two, then add; introduces one MSB error
    _m128i scanline_a0 = _mm_srai_epi8 (inScanlineA, 1);
    _m128i scanline_a1 = _mm_srai_epi8 (inScanlineB, 1);
    _m128i scanline_b0 = _mm_srai_epi8 (inScanlineC, 1);
    _m128i scanline_b1 = _mm_srai_epi8 (inScanlineD, 1);
    
    _m128i scanline_avg_a = _mm_add_epi8 (scanline_a0, scanline_a1);
    _m128i scanline_avg_b = _mm_add_epi8 (scanline_b0, scanline_b1);
    
    // mask even bytes
    a_even = _mm_slli_epi16(a_even, 8); // shift left 8 bits
    a_even = _mm_setl_epi16(0x00FF); // mask even bytes
    a_even = _mm_srai_epi16(a_even, 8); // shift right 8 bits, giving floating point 16 bits
```

49
4. IMPLEMENTATION

Figure 4.12: The global steps for SIMD-enabled chromatic subsampling.

```c
m128i a_even = _mm_and_si128(scanline_avg_a, _mm_set1_epi16(uint16(0xFFF0))); // mask even bytes
m128i b_even = _mm_and_si128(scanline_avg_b, _mm_set1_epi16(0x0FF)); // mask even bytes
m128i b_uneven = _mm_and_si128(scanline_avg_b, _mm_set1_epi16(0x10FF)); // mask even bytes
m128i a_avg = _mm_srli_epi16(a_even, 8); // shift left 8 bits
b_avg = _mm_srli_epi16(b_even, 8); // shift left 8 bits,
outScanline = _mm_packs_epi16(a_avg, b_avg);
}
inline void gChromaSubSampleB(
    _m128i& outScanline,
    const _m128i& inScanlineA,
    const _m128i& inScanlineB,
    const _m128i& inScanlineC,
    const _m128i& inScanlineD)
{
    m128i scanline_a0 = _mm_add_epi8(inScanlineA, _mm_set1_epi8(uint8(128))); // offset to unsigned range
    m128i scanline_a1 = _mm_add_epi8(inScanlineA, _mm_set1_epi8(uint8(128))); // offset to unsigned range
    m128i scanline_b0 = _mm_add_epi8(inScanlineB, _mm_set1_epi8(uint8(128))); // offset to unsigned range
    m128i scanline_b1 = _mm_add_epi8(inScanlineB, _mm_set1_epi8(uint8(128))); // offset to unsigned range
    m128i a_even = _mm_and_si128(scanline_avg_a, _mm_set1_epi16(0x0FFF)); // mask even bytes
```

50
Chromatic Subsampling

Finally, the performance results can be seen in Figure 4.13. The second implementation of the CoCg components and uses the unsigned SIMD instructions is marginally faster than the first one, which uses the manual averaging. Additionally, the unsigned version has better precision because it does not need the intermediate right shift. The second one will be used in our final implementation.

![Figure 4.13: Comparing the various subsampling implementations.](image)

### 4.2.4 Quality impact

Chromaticity subsampling is an often used way of compressing image data, but since we are using the YCoCg color space and relatively simple upscaling logic, it is something that should be tested to make sure we do not add too much error into the pipeline at this (early) stage. In order to do this, we measure the mean squared error (MSE) and peak signal to noise ratio PSNR. In order to get a lower bound, we use no upscaling logic and use raw averages when reconstructing the image. Two images from the test application can be seen in Figure 4.14 and Figure 4.15.
Higher quality can be achieved by applying more complex upscaling filters on the subsampled chromaticity channels; the issue with this is that it would require additional computations and add dependencies between tiles if the filter kernel gets complex. Because of these factors, we use a simplistic upsampling filter that stretches the image to twice its size without additional filtering; this can be implemented by simply sampling each pixel four times during reconstruction, and does not generate any additional memory or computational overhead. If the image quality is found lacking at a later stage, we can decide to revisit the upsampling and implement a higher quality filter.

### 4.3 DCT transform

The next step in the pipeline is the DCT transform; this transform is applied to each individual 8x8 pixel block, and differs from the previous stages in that the initial implementation requires floating point arithmetic. The first issue that arises is that the DCT transform is applied to 8x8 blocks of single-channel data while the Y channel is tiled in 16x16 blocks. Pushing this through DCT conversion would require seeks across the 16x16 data blocks in order to prepare it into 8x8 subtiles. Instead of adding another tiling stage after the subsampling, we (p)re-tile the Y channel during the subsampling stage since it will overlap with...
the non-linear reading from memory at that stage, reducing any additional impact on performance. The final subsampling function with the additional tiling is shown in Listing 4.11.

Listing 4.11: SIMD Enabled Chromatic Subsampling with Y tiling.

```c
inline void gChromaSubSampleFinal(
   Y16x16* outY,
   Co8x8* outCo,
   Cg8x8* outCg,
   const YCoCg16* inYCoCg,
   uint inTileCount)
{
    for (uint c = 0; c < inTileCount; c++)
    {
        for (uint s = 0; s < 4; s++) // each run of s is four 16-wide scanlines
        {
            // tile the Y channel into 4(2x2) 8x8 blocks, this means each 16-wide scanline is split over two blocks
            int block_y = s/2;    // 0 top two blocks, 1 is lower to blocks
            int block_scanline = (s&1)*2; // starting line within block

            outY[c].mSubTile[block_y][0].mData28[0+block_scanline] = _mm_unpacklo_epi64(inYCoCg[c*16+s*4+0].mY16, inYCoCg[c*16+s*4+1].mY16);
            outY[c].mSubTile[block_y][1].mData28[0+block_scanline] = _mm_unpackhi_epi64(inYCoCg[c*16+s*4+0].mY16, inYCoCg[c*16+s*4+1].mY16);
            outY[c].mSubTile[block_y][0].mData28[1+block_scanline] = _mm_unpacklo_epi64(inYCoCg[c*16+s*4+2].mY16, inYCoCg[c*16+s*4+3].mY16);
            outY[c].mSubTile[block_y][1].mData28[1+block_scanline] = _mm_unpackhi_epi64(inYCoCg[c*16+s*4+2].mY16, inYCoCg[c*16+s*4+3].mY16);
        }
    }
}
```

Figure 4.15: Comparing a subsampled image with the original; the PSNR is 42 dB.
4. IMPLEMENTATION

Figure 4.16: Subtiling the 16x16 Y channel tiles into four 8x8 tiles using SIMD intrinsics.

We can now apply DCT conversion to the 8x8 blocks across all channels. Our initial implementation uses a straightforward row-column transformation as shown in Listing 4.12.

Listing 4.12: Initial DCT implementation.

```
struct DCTBlock {
    float mCoeff[8][8];
};

template<typename T>
void DCTForward(Tile8<T>& outBlock, const DCTBlock& inData) {
    DCTBlock temp;
    for (uint dy = 0; dy < 8; dy++)
        for (uint dx = 0; dx < 8; dx++)
        {
            float accum = 0.0f;
            for (uint k = 0; k < 8; k++)
            {
                accum += inData.mCoeff[k][dy] * dct_bases[k][dx];
            }
            temp.mCoeff[dx][dy] = accum;
        }
    for (uint dy = 0; dy < 8; dy++)
        for (uint dx = 0; dx < 8; dx++)
        {
            float accum = 0.0f;
            for (uint k = 0; k < 8; k++)
            {
                accum += temp.mCoeff[dx][k] * dct_bases[k][dy];
            }
            outBlock.mData128[s].mData[dy][dx] = accum;
        }
    // subsample the C-channels; four 16-wide scan lines become 2 8-wide scan lines (in one 128-bit register)
    gChromaSubSampleB(outCo[c].mData128[s],
        inYCoCg[c*16+s+0].mCo16,
        inYCoCg[c*16+s+1].mCo16,
        inYCoCg[c*16+s+2].mCo16,
        inYCoCg[c*16+s+3].mCo16);
    gChromaSubSampleB(outCg[c].mData128[s],
        inYCoCg[c*16+s+0].mCg16,
        inYCoCg[c*16+s+1].mCg16,
        inYCoCg[c*16+s+2].mCg16,
        inYCoCg[c*16+s+3].mCg16);
}
```
The function uses the DCT basis functions through a lookup table that is filled at start-up. The DCT transformation we are using is the adjusted DCT-II function so it is an orthogonal matrix. To invert, the same matrix is read in transposed form, so only one matrix is needed. Listing 3.4.5 shows the implementation of our transformation matrix. Figure 4.17 shows the luminosity channel of an image after DCT conversion has been applied. The regular dotted grid is created due to the high constant values of each 8x8 block.

![Figure 4.17: The Y channel converted into DCT values.](image)


```c
void GenerateDCTTable()
{
    // k = coefficient number
    // n = driving var of base
    for (int n = 0; n < 8; n++)
        for (int k = 0; k < 8; k++)
            {dct_bases[n][k] = cosf(pi/8 + (float)n*0.5f + float(k));
                if (k == 0)
                    dct_bases[n][k] = (1.0f / sqrtf(2.0f));
                dct_bases[n][k] = sqrtf(2.0f/8.0f);
            // transpose = inverse
            dct_bases_inv[k][n] = dct_bases[n][k];
        }
}
```

4.3.1 Creating mipmaps

With a working DCT conversion, we try to create mipmaps from the DCT values by sub-sampling the DCT base-functions and using only the 16 highest frequency components. We do this by simply box-filtering the base functions as shown in Listing 4.14 and modifying the inverse function to output 4x4 values. Figure 4.18 shows a mipmap generated this way on the left side of the image, the highest mipmap on the right side of the image and the difference between the two in the middle image.
4. IMPLEMENTATION


```c
void GenerateDCT_Table4()
{
    if (inMaxFreq == 4) // subdiv
    {
        for (int k = 8; k < 8; k++)
            for (int n = 0; n < 4; n++)
            {
                dct_bases_filter4[k][n] = dct_bases_inv[k][n*2];
                dct_bases_filter4[k][n] += dct_bases_inv[k][n*2+1];
                dct_bases_filter4[k][n] *= 0.5f;
            }
    }
}
```

Figure 4.18: The Y channel recreated as a 2x2 subsampled mipmap using a subset of the DCT coefficients to the left with the original data to the right and the difference between the two in the middle.

4.3.2 Performance

Although the results are as expected, the performance of the DCT conversion is poor; the DCT transform (forward and backward) has three nested loops, resulting in $8 \times 8 \times 8 = 512$ operations per block. This results in the timings seen in Figure 4.19. Luckily there are many more optimized DCT implementations. One of these implementations is the intDCT implementation [24]. This implementation uses an adjusted DCT matrix that can be expressed as a series of shifts and additions instead of a full 8x8 matrix multiplication. One of the big advantages of this implementation is the fact that it uses only integer math. This removes the float-to-integer conversion and enables 16-bit SIMD-optimization. There are many variants of the intDCT transform; in this implementation, the intDCT-C variant is used. The forward- and inverse matrices are shown in Figure 4.21, while the fast implementation is shown in Figure 4.22. Note that in addition to the shown scheme, there are additional 1/2 scalars applied to the result of both the forward and inverse functions. The code implementation is found in Listing 4.15. The performance measurement of the intDCT-C transform compared to the initial DCT implementation can be seen in Figure 4.23.

Although the intDCT conversion is a lot faster than the more direct implementation, it cannot be used unless its frequency distribution is similar to the original DCT transform.
since this is the basis for the efficient mipmap generation. Figure 4.20 shows the effective
2D basis functions of both the DCT and intDCT transform; these functions are generated
by inverse-transforming a set of 8x8 DCT coefficient blocks $i_{nm}$ where $0 \leq n, m \leq 7$ and
$i_{nm}[n][m] = 1$ and all other coefficients equal to zero. To the right it shows the effective base
functions used by mipmaps; the red blocks show basis functions not used during mipmap
reconstruction.

Listing 4.15: IntDCT-C implementation.

```c
// outPos = inPos + inNeg
// inNeg = −inNeg + inPos
// inPos = outPos
inline void gButterfly(int& inPos, int& inNeg)
{
    inPos += inNeg;
    inNeg = (inPos - inNeg + 2);
}

// based on intDCT-C from:
// The BinDCT: Fast Multiplierless Approximation of the DCT
```
4. IMPLEMENTATION

<table>
<thead>
<tr>
<th>Forward Transform Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/2   1/2   1/2   1/2   1/2   1/2   1/2</td>
</tr>
<tr>
<td>1/2   1/2   3/16  0    0    -3/16  -1/2  -1/2</td>
</tr>
<tr>
<td>9/32  -1/8  -19/64  -1/4  1/4  -19/64  1/8  -9/32</td>
</tr>
<tr>
<td>1/4  -1/4  -1/4   1/4  1/4  -1/4  -1/4  1/4</td>
</tr>
<tr>
<td>7/16  -3/4  -7/32  1/2  -1/2  -7/32  3/4  -7/16</td>
</tr>
<tr>
<td>-3/16  -1/2  -1/2  3/16  3/16  -1/2  1/2  -3/16</td>
</tr>
<tr>
<td>-1/16  -1/4  -13/32  1/2  -1/2  13/32  -1/4  1/16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inverse Transform Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/4   1/4   1/4   1/4   1/4   1/4   1/4</td>
</tr>
<tr>
<td>1/2   13/32  1/4   1/16  -1/16  -1/4  -13/32  -1/2</td>
</tr>
<tr>
<td>1/2   3/16  -3/16  -1/2  -1/2  -3/16  3/16  1/2</td>
</tr>
<tr>
<td>1/2  -7/32  -3/4  -7/16  7/16  3/4  7/32  1/2</td>
</tr>
<tr>
<td>1/2  -1/2  -1/2  1/2  1/2  -1/2  -1/2  1/2</td>
</tr>
<tr>
<td>1/4  -19/64  1/8  9/32  -9/32  1/8  19/64  -1/4</td>
</tr>
<tr>
<td>0  3/16  -1/2  1/2  -1/2  1/2  -3/16  0</td>
</tr>
</tbody>
</table>

Figure 4.21: The intDCT-C transform matrices.

Figure 4.22: The intDCT-C fast implementation.

// Trac D. Tran, 2000

template< typename S , typename T >
inline void gFastDCT( S* outData , const T* inData )
{
    const int* i = inData;
    int x[8];
    gButterflyInit( inData[0] , i[7] , x[0] , x[7] );

    gButterfly( x[0] , x[3] );
    gButterfly( x[1] , x[2] );
    gButterfly( x[4] , x[5] );
    gButterfly( x[7] , x[6] );


58
Another problem with this implementation is that there is no simple way of creating the inverse transforms needed for mipmap creation. Instead, we need to manually adjust and implement the fast implementation graph seen in Figure 4.22. These can be constructed by
4. IMPLEMENTATION

Figure 4.23: The timings of the intDCT implementation, compared to the first DCT conversion.

taking the inverse scheme and assuming the higher coefficients to be zero, whilst averaging the inputs. Note that the scalar for each consecutive mipmap is halved to $1/4$, $1/8$ and $1/16$ to account for the additive averaging step at the end. The resulting schemes of the four mipmaps are shown in Figure 4.24.

Figure 4.24: The implementation schemes of all four mipmap DCT inverse functions.

4.3.3 Vectorized implementation

For an additional performance increase, we should opt for a vectorized implementation; because we are using a 16-bit representation for the DCT coefficients, eight coefficients will fit
in one 128-bit SIMD register. Since our 2D DCT coefficient matrix is 8x8, a full 8-column transform can be performed in a single pass. A drawback is that we can only perform the transform on columns, so a transpose has to be performed between the row- and column transform to implement the full 2D row-column DCT transform. The implementation of the vectorized inverse intDCT transform is shown in Listing 4.16.

Listing 4.16: IntDCT-C inverse transform SIMD implementation.

```c
void gIntDCTInverseSIMD ( Tile8< int16 >& outBlock, const Tile8< int16 >& inData )
{
    const _m128i i = &inData.mData128[0]; // shorthand for input data
    _m128i X[8]; // intermediate values
    _m128i temp;
    _m128i const3 = _mm_set1_epi16(3);
    _m128i const5 = _mm_set1_epi16(5);
    _m128i const7 = _mm_set1_epi16(7);
    // X4.6
    // X[4] = -(i[0] + (i[0] / 2));
    temp = _mm_srai_epi16(i[0], 1);
    X[4] = _mm_sub_epi16(temp, i[4]);
    // X[0] = i[0] - X[4];
    X[0] = _mm_sub_epi16(i[0], X[4]);
    // X6.2
    // X[2] = (i[2] - (i[0] + (i[0] / 2))) / 8;
    temp = _mm_srai_epi16(i[0], 1);
    X[2] = _mm_sub_epi16(temp, i[2]);
    temp = _mm_sub_epi16(i[0], X[2]);
    X[6] = _mm_add_epi16(i[6], temp);
    // X5.3
    temp = _mm_srai_epi16(i[5], 1);
    X[3] = _mm_add_epi16(i[3], temp);
    temp = _mm_sub_epi16(i[3], X[5]);
    X[5] = _mm_add_epi16(i[5], temp);
    // X7.2
    // X[1] = i[1];
    temp = _mm_add_epi16(X[1], 3);
    X[7] = _mm_add_epi16(i[7], temp);
    gButterflySIMD(X[4], X[6]);
gButterflySIMD(X[0], X[2]);
gButterflySIMD(X[7], X[5]);
gButterflySIMD(X[1], X[3]);
    // X5.3
    temp = _mm_srai_epi16(X[3], 1);
    temp = _mm_sub_epi16(temp, X[5]);
    X[5] = _mm_sub_epi16(temp, X[5]);
    temp = _mm_srai_epi16(X[3], 1);
    temp = _mm_sub_epi16(temp, X[5]);
    X[3] = _mm_sub_epi16(X[3], temp);
}
```
4. IMPLEMENTATION

```c
4. IMPLEMENTATION

```ButterflySIMD(X[0], X[1]);
ButterflySIMD(X[4], X[3]);
ButterflySIMD(X[6], X[5]);
ButterflySIMD(X[2], X[7]);

outBlock.mData128[0] = _mm_srli_epi16(X[0], 1);
outBlock.mData128[1] = _mm_srli_epi16(X[4], 1);
outBlock.mData128[2] = _mm_srli_epi16(X[6], 1);
outBlock.mData128[3] = _mm_srli_epi16(X[2], 1);
outBlock.mData128[4] = _mm_srli_epi16(X[7], 1);
outBlock.mData128[5] = _mm_srli_epi16(X[5], 1);
outBlock.mData128[6] = _mm_srli_epi16(X[3], 1);
outBlock.mData128[7] = _mm_srli_epi16(X[1], 1);
```

```c
} // uint8 -> int16 casting using SIMD
for (int x = 0; x < 4; x++) {
    temp2.mData128[x*2] = _mm_unpacklo_epi8(inData.mData128[x], _mm_set1_epi32(0));
    temp2.mData128[x*2+1] = _mm_unpackhi_epi8(inData.mData128[x], _mm_set1_epi32(0));
}
```

```c
gIntDCTForwardSIMD(temp, temp2);
```

```c
// transpose
for (int x = 0; x < 64; x++) {
    int r = x >> 3;
    int c = x & 7;
    temp2.mDataLinear[x] = temp.mDataLinear[(c << 3) + r];
}
```

```c
gIntDCTForwardSIMD(outBlock, temp2);
```

The timings show a large improvement over the linear implementation. The results can be seen in Figure 4.25.

```
Figure 4.25: Timings of the SIMD implementation of the intDCT-C transform.
```

62
4.3.4 Q-tables and quality

In order to compress the DCT coefficient data more effectively, the coefficients are quantized according to Q-tables. These quantization tables define the precision for each of the 64 DCT values. The quantization is a trade-off between compression ratio and quality. Creating a good Q-table involves the creation of heuristics to model the human eye response and analyzing image properties. However, this is outside the scope of this thesis and therefore we will be using the existing Quantization tables used by Adobe Photoshop, a photo edition suite. These tables are available online[13] and are defined separately for Chroma and Luma channels and available for various quality/size tradeoffs.

For the following sections, we will use a default set of Q-tables shown in Table 4.1. The data used is based on the mid-quality JPEG Q-tables used by Adobe Photoshop. In the full analysis in Chapter 5, we will use a more extensive set of q-tables to get a better overview of the compression properties of our streaming pipeline.

4.3.5 Coefficient Reordering

Before Run-Length Encoding can be used on the data, the coefficients have to be reordered, as shown in Figure 3.11. In the initial implementation, this was done by simply iterating over all DCT data 64 times, retrieving the DCT coefficients one by one and adding them to the output buffer containing the reordered data. However, this proved to be extremely slow due to the cache-unfriendliness of the many repeated traversals. A different implementation was designed that would do less traversals across all data by using an intermediate buffer to improve cache efficiency. The two schematics and their performances can be seen in Figure 4.26. Multiple versions of the latter schema were tried, with various buffer sizes; a buffer size containing four coefficients per block had the best results with an average of 40ms for a 2048 × 2048 pixel image.

![Figure 4.26: Two Coefficient reordering schemas, the left one does more operations but performs better](image-url)
4. IMPLEMENTATION

4.4 Run-Length Encoding

The next phase in the pipeline is the compression of the DCT coefficients. Since the coefficients are already integers, we can compress them directly through Run-Length Encoding without additional quantization: We take advantage of recurring coefficients and store these spans of repeating coefficients as a marker/length/coefficient triplet. For data that contains repeating data with a length of three or more, this results in a net compression of data, with an additional repeat-token added to the alphabet. Because the next step is Huffman encoding, we can freely use the word-size of our data at this point; using signed short values will be enough to encode all possible coefficients with enough room to add an additional RLE token. Listing 4.17 shows the RLE encoding and decoding function.

Listing 4.17: Run-Length Encoder and Decoder

typedef short RLEToken;  // should be at least as large as the encoded word-size
const RLEToken cRLERepeatToken = 0x7FFF;  // should never appear in our raw data(*)
template<typename T>
uint RunLengthEncode(RLEToken* outData, const T* inData, uint inLength)
{
    int running_length = 0;
    int running_token = cRLERepeatToken;
    RLEToken* data_ptr = outData;
    for (uint c = 0; c < inLength; c++)
    {
        const T& current = inData[c];
        if (current == running_token)
        {
            running_length ++;
        }
        else
        {
            // lengths below 4 are simply outputted as is
            if (running_length < 4)
                for (int c = 0; c < running_length; c++)
                {
                    *(data_ptr++) = running_token;
                }
            // lengths above are encoded with delim/runtime/token
            else
                {
                    *(data_ptr++) = cRLERepeatToken;
                    *(data_ptr++) = running_length;
                    *(data_ptr++) = running_token;
                }
            running_length = 1; running_token = inData[c];
        }
    }
    // finalize any running_length left
    if (running_length < 4)
        for (int c = 0; c < running_length; c++)
            *(data_ptr++) = running_length;
    else
        {
            *(data_ptr++) = cRLERepeatToken;
            *(data_ptr++) = running_length;
            *(data_ptr++) = running_token;
        }
    return (data_ptr- outData);
}

template<typename T>
uint RunLengthDecode(T* outData, RLEToken* inData, uint inLength)
Run-Length Encoding depends on repeating data to compress effectively. The most repeating data can be found in the higher frequencies, where the coefficients are relatively low, especially after quantization, most high frequencies will be zero. In order to leverage this fact, we sort all DCT data by coefficient during RLE compression, creating large sections of zeros, improving compression. In order to effectively store and retrieve the (compressed) DCT data, it needs to be stored in such a way that each LOD-level can be loaded separately. This is why we store the DCT data in JZag order, as described in Section 3.5.2. Figure 4.27 shows the compressed RLE data: bright cells describe high coefficients, while red cells denote repeat tokens in the data.

```
T* data_ptr = outData;
for (uint c = 0; c < inLength; c++)
{
    RLEToken& current = inData[c];
    if (current == rREpeaToken)
    {
        c++;
        uint repeat_count = (uint) inData[c];
        c++;
        T repeat_value = (T) inData[c];
        for (uint r = 0; r < repeat_count; r++)
        {
            *(data_ptr++) = repeat_value;
        }
    }
    else
    {
        *(data_ptr++) = (T) current;
    }
} return (data_ptr - outData);
```

Figure 4.27: Showing the RLE compressed data. From left to right: unsorted, sorted by coefficient, sorted and quantized with a default Q-table. Compression ratio goes up from 1:2 to 1:5.

Grouping all coefficients greatly improves compression, but DCT-DXT transcoding is not necessarily done in a single pass due to memory constraints. The minimal granularity for decompressing the data is 16 × 16 blocks: four 8x8 luminance channel blocks and two subsampled chromatic channels can be converted into a 16 × 16 block of RGB data. The most optimal way of storing data would therefore be to sort by coefficients, group them by mipmap, and interleave the coefficients in three channels in a 4:1:1 interleaving scheme. This would enable the decoding pass to convert any amount of 16 × 16 RGB blocks at one time, depending on available memory. However, this interleaving of channels would
4. IMPLEMENTATION

Table 4.1: Chromaticity and Luminance Q-table.

<table>
<thead>
<tr>
<th>Luminance</th>
<th>Chrominance</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 8 13 21 26 32 24 17</td>
<td>13 13 17 27 20 20 17 17</td>
</tr>
<tr>
<td>8 9 12 20 27 23 12 12</td>
<td>13 14 17 14 14 12 12 12</td>
</tr>
<tr>
<td>13 12 16 26 23 12 12 12</td>
<td>17 17 14 14 12 12 12 12</td>
</tr>
<tr>
<td>21 20 26 23 12 12 12 12</td>
<td>27 14 14 12 12 12 12 12</td>
</tr>
<tr>
<td>26 27 23 12 12 12 12 12</td>
<td>20 14 12 12 12 12 12 12</td>
</tr>
<tr>
<td>32 13 12 12 12 12 12 12</td>
<td>20 12 12 12 12 12 12 12</td>
</tr>
<tr>
<td>34 12 12 12 12 12 12 12</td>
<td>17 12 12 12 12 12 12 12</td>
</tr>
<tr>
<td>17 12 12 12 12 12 12 12</td>
<td>17 12 12 12 12 12 12 12</td>
</tr>
</tbody>
</table>

be detrimental to the RLE compression ratio. As initially proposed in Chapter 3, we will therefore pick a middle-road, where we group sets of coefficients in 16-pixel wide scanlines. Figure 4.28 shows the resulting compression data of an image interleaved in scanlines. The compression ratio for this test image is 0.16 compared to raw RGB - in comparison, DXT also has a fixed ratio of 6:1, or 0.16 for non-alpha images. This is a promising ratio, since it is already our target compression ratio and we still need to apply the Huffman compression layer, which will be discussed in the next section.

Figure 4.28: Showing the final encoded data; sorted by coefficient, with a 4:1:1 interleaving of its channels.

4.4.1 Alternative RLE encoding

Instead of using a separate token to denote a repetition, we can use another way of delimiting runs; we use the duplication of a character as a token; this means that we lose the extra token in our heuristic, which can simplify our histogram generation code in the next subsection. However this creates additional overhead: a “natural” occurring duplicate now needs an explicit count of two appended, resulting in an increase in size instead of a reduction in
Huffman Coding

size. Compared to the original RLE, this creates slightly larger data sets, however we gain compression during the Huffman phase, resulting in a net gain in compression and speed. The compression code is listed in Listing 4.18.

Listing 4.18: Alternate RLE Encoder

template <typename T>
uint RunLengthEncode(RLEToken* outData, const T* inData, uint inLength)
{
  int running_length = 1;
  int running_token = inData[0];
  RLEToken* data_ptr = outData;
  for (uint c = 1; c < inLength; c++)
  {
    const T& current = inData[c];
    if (current == running_token)
    {
      running_length++;
    }
    else
    {
      // single characters are output as is
      if (running_length == 1)
        *(data_ptr++) = running_token;
      // lengths above 1 are encoded with a double running token followed by length
      else
        {
          *(data_ptr++) = running_token;
          *(data_ptr++) = running_token;
          *(data_ptr++) = running_length;
        }
      running_length = 1; running_token = inData[c];
    }
  }
  // finalize any running_length left
  if (running_length == 1)
    *(data_ptr++) = running_token;
  // lengths above are encoded with delim/runtime/token
  else
    {
      *(data_ptr++) = running_token;
      *(data_ptr++) = running_token;
      *(data_ptr++) = running_length;
    }
  return (data_ptr - outData);
}

4.4.2 Performance

Figure 4.29 shows the performance of the RLE implementation in milliseconds. The data set that is being compressed is a 2048x2048 image. As can be seen, the decoding speed is slightly slower than the encoding speed.

4.5 Huffman Coding

The next step in the pipeline is to use Huffman coding to further compress the data. Huffman coding is an entropy encoding, based on the histogram of the encoded data. Firstly, the data is analyzed and a histogram is made from the data: Figure 4.30 shows the histogram of the interleaved RLE data shown in Figure 4.28. The next step is to create a balanced tree from the histogram data by recursively merging the smallest/lightest leaves and branches
into larger ones until the root-node has been created. The third step is to assign each node a Huffman code by recursively assigning each left child node the code of the parent node with a zero appended, and each right child node the parent node with a one appended. When all nodes have been visited, these codes describes routes from the root node to each individual other node. Since the length of each node is equal to the depth of that node, heavy leaves (containing often recurring RLE tokens) have short codes, while infrequent nodes have long nodes, resulting in an effective compression.

Figure 4.30: Encode and Decode speed of the RLE implementation in ms.

Figure 4.30: Histogram of compressed data.
4.5.1 Histogram Generation

The efficiency of creating a histogram is dependent mostly on the so-called binning algorithm that is used. In this case, this translates to the efficiency of the algorithm that checks if a token has already been encountered and that adds or increases the weight of the proper bin. Since the huffman decoding should be loss-less, each unique token is a separate bin. Pre-allocate bins for all possible token values will be too inefficient, so the algorithm will need to dynamically allocate bins as we encounter new tokens. However this approach would be slow and inefficient as we would need to search through all dynamic bins each time we process a new token. Although this can be made more efficient by sorting the bins using injection sorts and using binary searches, the proposed implementation leverages the expected distribution to create a hybrid version where we pre-allocate bins for a certain range of tokens for direct indexing while dynamically binning the outliers outside this range. The distribution peaks around zero, we dynamically allocate \( c_{\text{StaticRange}} \) bins to the left and to the right of zero. The schematic representation of this algorithm can be seen in Figure 4.31. The code that implements this principle can be found in Listing 4.19. After experimentation, a \( c_{\text{StaticRange}} \) of around 128 seems to give the best results for our set of test cases. Figure 4.31 shows this binning algorithm.

![Histogram Generation Flowchart](image)

Figure 4.31: Histogram Generation Flowchart with the static and dynamic bins below. The values in the boxes denote the tokens represented with each bin. The dynamic bins are allocated to any outliers that are encountered in the binning process.


```c
struct HuffmanBin
{
    uint16 mToken;
    int    mCount;
    HuffmanBin()
    {}
    HuffmanBin(uint16 inToken, int inCount)
    : mToken(inToken), mCount(inCount)
    {}
}
```

69
4. IMPLEMENTATION

};

int GenerateHuffmanHistogram(
    RLEToken* inData,
    uint inElementCount,
    HuffmanBin* inBins
)
{
    // static
    const int cStaticRange = 128;
    const uint cStaticBins = cStaticRange+2;
    uint extra_tokens = 0;

    int c = 0;
inBins[cStaticRange] = HuffmanBin(cRLERepeatToken, 0);
    for (int p = 0; p < cStaticRange; p++)
    {
        inBins[p] = HuffmanBin(-cStaticRange + p, 0);
        inBins[cStaticRange+1+p] = HuffmanBin(p, 0);
    }

    for (uint n = 0; n < inElementCount; n++)
    {
        RLEToken t = inData[n];
        assert(t != RLEToken(0xcdcd));
        uint h;
        if (t > -cStaticRange && t < cStaticRange)
        {
            // static
            inBins[t+cStaticRange];
        }
        else
        {
            // dynamic
            for (h = cStaticBins; h < cStaticBins+extra_tokens; h++)
            {
                if (inBins[ht].mToken == t)
                {
                    inBins[ht].mCount++; break;
                }
            }
            if (h == extra_tokens+cStaticBins)
            {
                inBins[cStaticBins + (extra_tokens++)] = (HuffmanBin(t, 1));
            }
        }
    }
    return cStaticBins+extra_tokens; // amount of histogram data generated
}

4.5.2 Balanced Tree Generation

The next step is the generation of a balanced (Huffman) tree. This can be done using a bottom-up approach whereby the tree is initiated using the histogram data as leaves. The branches are generated by iteratively joining the lightest leaf and branch nodes into new nodes until the entire tree is constructed. Figure 4.32 shows such a construction in steps.

To keep the code simple and fast, we use the existing histogram data structure; the HuffmanBin is expanded to include child references mLeft and mRight. The full code is listed in Listing 4.20.

Listing 4.20: Huffman Tree Generation Function.

int GenerateHuffmanTree(RLEToken* inData, uint inElementCount)
{
    // static one-time init of data; using STL so we can use std::sort
    static std::vector<HuffmanBin> bins;
    bins.resize(10000); // histogram data size

70
Huffman Coding

Figure 4.32: Constructing a balanced tree from histogram data. The blue entries denote leaf nodes, green entries denote branch nodes and white entries denote nodes that have already been merged.

```cpp
uint leaf_count = FillHistogramData(inData, inElementCount,)
std::sort(bins.begin(), bins.begin() + cStaticBins + extra_tokens);

// skip zero-entries
uint bgn_leaf_idx;
for (bgn_leaf_idx = 0; bgn_leaf_idx < leaf_count && bins[old_values].mCount == 0; bgn_leaf_idx++)
{
}
uint bgn_branch_idx = leaf_count;
uint node_count = leaf_count;

// while we have leaves or at least two branches left
while (bgn_leaf_idx != end_leaf_idx || bgn_branch_idx != node_count - 1)
{
    int child_left, child_right;

    // get lightest leaf or branch for first child
    if (end_leaf_idx == bgn_leaf_idx ||
        (ttl_values > bgn_branch_idx && bins[bgn_branch_idx] < bins[bgn_leaf_idx]))
        child_left = bgn_branch_idx++;
    else
        child_left = bgn_leaf_idx++;

    // get lightest leaf or branch for second child
    if (end_leaf_idx == bgn_leaf_idx ||
        (ttl_values > bgn_branch_idx && bins[bgn_branch_idx] < bins[bgn_leaf_idx]))
        child_right = bgn_branch_idx++;
    else
        child_right = bgn_leaf_idx++;

    bins[ttl_values] = HuffmanBin(child_left, node_count, child_right, node_count,
        bins[child_left].mCount + bins[child_right].mCount); node_count++;
}
```
4. IMPLEMENTATION

4.5.3 Tree code generation

Now that the tree has been constructed, we can encode each leaf and output the binary data. The encoding of each leaf can be done using a recursive tree walk whilst encoding each child node with its parent’s Huffman code and appending a one or zero, depending on which child it is. Listing 4.21 shows the full implementation of the HuffmanBin object along with the EncodeNode() function that recursively assigns a Huffman code to each node.

Listing 4.21: Fully implemented Huffman Bin.

```c
struct HuffmanBin
{
    union
    {
        struct // leaf
        {
            RLEToken mToken;
            uint16 mPadding; // will be zero for a leaf node
        }
        struct // branch
        {
            int16 mLeft;
            int16 mRight; // will be non-zero for branch node
        }
    }
    int mCount;
    int mCode;
    int mCodeSize;

    bool IsLeaf() { return mRight == 0; } // mRight = mPadding

    void EncodeNode(int inCode, int inCodeSize)
    {
        mCode = inCode;
        mCodeSize = inCodeSize;
        if (mPadding != 0)
        {
            HuffmanBin* left = this + mLeft;
            HuffmanBin* right = this + mRight;
            left->SetCode((inCode << 1), inCodeSize + 1);
            right->SetCode((inCode << 1)+1, inCodeSize + 1);
        }
    }

    bool operator<(const HuffmanBin& inOther) { return mCount < inOther.mCount; }
    bool operator>(const HuffmanBin& inOther) { return mCount > inOther.mCount; }

    HuffmanBin(RLEToken inToken, int inCount) : mCode(0), mToken(inToken), mCount(inCount), mCodeSize(0), mPadding(0) {}
    HuffmanBin(int16 inLeft, int16 inRight, int inCount) : mCode(0), mCount(inCount), mLeft(inLeft), mRight(inRight) {}
    HuffmanBin() : mCode(0), mToken(0), mCount(0), mCodeSize(0), mPadding(0) {}
};
```

4.5.4 Binary Encoding and Decoding

Using the generated tree, the input data can be compressed by rewriting it as Huffman codes and storing the resulting binary data along with the Huffman tree that generated it. To speed up the lookup of these Huffman codes, the static-dynamic hybrid approach from the histogram creation phase can be used.

Using the tree that was created during the encoding can also be used to decode the bit streams. This is done by simply traversing the tree using the bit stream to decide if to go left
or right during traversal. Whenever a leaf node is reached, a token has been decoded and traversal is started again from the root until the original data has been fully reconstructed.

Listing 4.22 shows an implementation of the Huffman encoding and decoding algorithms. The inHistogramData object contains the tree data generated during tree construction.

Listing 4.22: RLE Encoding and Decoding functions

```c
uint HuffmanEncode
{
    uint32_t outData ,
    RLEToken* inData ,
    uint inElementCount ,
    HistogramData& inHistogramData
}

    // retrieving generated histogram data
    inHistogramData.CreateCodeLookup();
    CodeLookups lookups = inHistogramData.GetCodeLookup();
    uint leafCount = inHistogramData.GetLeafCount();
    uint buffer[2] = {0, 0}; // double output buffer
    uint buffer_size = 0; // amount of bits in output buffer
    uint32_t outPtr = outData; // output stream
    for (uint n = 0; n < inElementCount; n++)
    {
        RLEToken t = inData[n];

        // fast lookup in histogram data for tokens in the static index
        uint bin_idx = GetStaticIndex(t);

        // slow lookup for outliers
        if (bin_idx == cInvalidStaticIndex)
        {
            for (uint h = cStaticBins; h < leafCount; h++)
                if (lookups[h].mToken == t)
                    break;
        }

        // retrieve code
        uint code = lookups[bin_idx].mCode;
        uint code_length = lookups[bin_idx].mCodeSize;
```
4. IMPLEMENTATION

// output binary stream through double buffer
buffer[0] += (code&Mask(code_length)) << buffer_size;
if (buffer_size + code_length >= 32) // guard for negative shifts
    buffer[1] += (code&Mask(code_length)) >> (32-buffer_size);
buffer_size += code_length;
// when the first buffer is full, flush it to output and swap in the second one
if (buffer_size >= 32)
    {
        *(outPtr++) = buffer[0];
        buffer[0] = buffer[1];
        buffer[1] = 0;
        buffer_size -= 32;
    }

// flush buffer
if (buffer_size > 0)
    *(outPtr++) = buffer[0];
return outPtr-outData;

uint HuffmanDecode(RLEToken*outData,
    const uint32_t inData,
    uint inTokenCount,
    HistogramData& inHistogramData)
{
// retrieving quick-lookup tree data
QuickTree* root = inHistogramData.GetQuickRoot();
QuickTree* tree_traverse = root;
const uint32_t* inptr = inData;
// output
RLEToken* outptr = outData;
// bitstream buffer
uint buffer_size = 0;
uint buffer = 0;
uint to_go = inTokenCount;
while (to_go > 0)
    {
        // fetch data from stream
        // when buffer is empty
        if (buffer_size == 0)
            {
                buffer = *inptr++;
                buffer_size = 32;
            }
        // consume one bit to traverse tree
        bool in_bit = (buffer&0x00000001) != 0;
        buffer >>=1;
        buffer_size--;;
        // traverse branch left or right
        tree_traverse &= in_bit ? tree_traverse->mLeft+1 : tree_traverse->mLeft;
        // when leaf is reached
        if (tree_traverse->IsLeaf())
            {
                // output token
                *(outptr++) = tree_traverse->mToken;
                tree_traverse = root;
                to_go--;
            }
    }
return outptr - outData;
4.6 Performance

In order to maximize performance, the hybrid lookup used during the histogram generation was used during the decoding process. Additionally, the tree was stored in a more compact way to increase memory coherence and reduce cache misses. The used tree representation during decoding only contained the \(\text{mLeft}, \text{mRight}\) and \(\text{mToken}\) members. The resulting performance is shown in Figure 4.34.

![Figure 4.34: Huffman Encoding and Decoding Performance of a 4 megapixel image in milliseconds.](image)

4.7 DXT Encoding

The final step in decoding is the generation of the DXT encoded image data. There are several ways of encoding a raw image into a DXT format. Squish[2] is a high performance library that gives the user access to several iterative and heuristic methods of generating optimal DXT encodings of an image. However, in the case of run-time trans-coding, speed is favored over quality. Because of this, the implementation contains a simple algorithm that maps all colors to one dimension before sorting and binning the 16 values. One of the drawbacks of this method is poor performance on several edge cases where luminosity is stable but chromaticity changes. Figure 4.35 shows an extreme example of this: the gradient colored image has a PSNR of 30 and the more organic texture on the right side of the image compresses with a PSNR of 40. For future implementations, this can be mostly negated by using the fact that the method is fully deterministic: by adding hinting and/or replacement data, any low-fidelity DXT blocks can be patched or replaced with pre-generated high fidelity alternatives. For more information about the layout of the DXT format, see Appendix A. Listing 4.23 shows the compression algorithm used. It accepts 16 bits of RGB data, packed similarly to the raw YCoCg data, this way no additional data swizzling has to be performed before transcoding from transformed RGB data. The performance of this algorithm is shown in Figure 4.36.
Figure 4.35: The DXT compression algorithm does not handle chromaticity changes without luminosity changes very well.

Figure 4.36: DXT conversion speeds of a 4 megapixel image in milliseconds

Listing 4.23: RGB-DXT Conversion function

```c
void gConvertToDxt16(const RGBColor16& inColor, DXTBlock& outBlock)
{
    int lums[cDxtCount];
    int lum_min_idx = 0;
    int lum_max_idx = 0;
    lums[0] = inColor.mRed[0] + inColor.mGreen[0] + inColor.mBlue[0];
    int lum_min = lums[0];
    int lum_max = lums[0];
    for (uint i = 1; i < cDxtCount; i++)
    {
        int lum = inColor.mRed[i] + inColor.mGreen[i] + 2 * inColor.mBlue[i];
        lums[i] = lum;
        if (lum < lum_min) { lum_min = lum; lum_min_idx = i; }
        if (lum > lum_max) { lum_max = lum; lum_max_idx = i; }
    }
    gColor24To16(outBlock.mColor0, inColor.mRed[lum_min_idx], inColor.mGreen[lum_min_idx], inColor.mBlue[lum_min_idx]);
    gColor24To16(outBlock.mColor1, ...
```
With the DXT algorithm, all stages and algorithms of the streaming system have been implemented and tested for performance. Chapter 5 shows the results of several streaming operations on a representative texture set.
Chapter 5

Measurements and Results

For a final measurement, a set of textures is streamed from the compressed representation to the DXT output. In order to get representative results, texture site CGTextures.com was contacted to provide a set of the most downloaded textures. The compression and timing results can be seen in Table 5.1. The measurements were performed on a Pentium Core2Duo processor running at 3.0 Ghz, running on a single thread. The processor speed is comparable to the 3.2 Ghz processor that runs the Xbox 360, or the Cell processor that runs the PS3. More in-depth results of the separate streaming and compression stages can be found in Chapter 4.

The results show that the current implementation can stream in textures at around 40 megapixel per second, compressing them at a ratio from 1:16 up to 1:50. The performance can be further manipulated by exploring different Q-Tables and optimizing the various stages of the streaming pipeline. This satisfies the goal set in Section 3.6.

As part of the thesis research, the entire pipeline as laid out here has also been implemented on a Xbox 360 in 2009. The system was able to run at sufficient speed in a retail-quality environment and has been added to the technical portfolio of Triumph Studios. This Xbox 360 implementation also included run-time defragmentation and additional memory management. The system has not yet been implemented on the PS3 architecture.
### 5. Measurements and Results

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Size</td>
<td>2.56</td>
<td>2.56</td>
<td>2.56</td>
<td>2.56</td>
<td>1.25</td>
<td>0.80</td>
<td>1.86</td>
<td>2.16</td>
<td>4.0</td>
</tr>
<tr>
<td>Huffman Decode</td>
<td>18.1</td>
<td>18.6</td>
<td>18.6</td>
<td>14.4</td>
<td>8.2</td>
<td>6.8</td>
<td>6.8</td>
<td>7.9</td>
<td>19.4</td>
</tr>
<tr>
<td>RLE Decode</td>
<td>6.8</td>
<td>6.8</td>
<td>7.2</td>
<td>7.4</td>
<td>2.5</td>
<td>1.6</td>
<td>3.9</td>
<td>3.6</td>
<td>10.3</td>
</tr>
<tr>
<td>Coefficient Reordering</td>
<td>7.0</td>
<td>9.0</td>
<td>9.0</td>
<td>8.5</td>
<td>5.0</td>
<td>2.5</td>
<td>4.9</td>
<td>5.0</td>
<td>5.6</td>
</tr>
<tr>
<td>Inverse Int-DCT</td>
<td>8.2</td>
<td>7.9</td>
<td>7.9</td>
<td>7.9</td>
<td>4.9</td>
<td>2.8</td>
<td>5.8</td>
<td>8.3</td>
<td>18.2</td>
</tr>
<tr>
<td>Inverse YCoCg</td>
<td>6.2</td>
<td>5.7</td>
<td>6.3</td>
<td>4.3</td>
<td>2.9</td>
<td>1.5</td>
<td>4.3</td>
<td>5.7</td>
<td>11.1</td>
</tr>
<tr>
<td>DXT conversion</td>
<td>15.0</td>
<td>18.5</td>
<td>18.3</td>
<td>18.3</td>
<td>9.0</td>
<td>5.7</td>
<td>12.5</td>
<td>14.9</td>
<td>29.8</td>
</tr>
<tr>
<td>Total</td>
<td>61.3</td>
<td>66.5</td>
<td>67.8</td>
<td>58.1</td>
<td>32.5</td>
<td>20.9</td>
<td>38.2</td>
<td>45.4</td>
<td>94.4</td>
</tr>
<tr>
<td>Throughput (megapixel/sec)</td>
<td>41.7</td>
<td>38.4</td>
<td>37.7</td>
<td>44.0</td>
<td>38.4</td>
<td>38.4</td>
<td>48.8</td>
<td>47.6</td>
<td>42.3</td>
</tr>
<tr>
<td>PSNR</td>
<td>34.2</td>
<td>32.2</td>
<td>39.9</td>
<td>30.5</td>
<td>33.5</td>
<td>33.4</td>
<td>33.4</td>
<td>33.4</td>
<td>32.5</td>
</tr>
<tr>
<td>Compression (percentage of raw RGB)</td>
<td>6.0</td>
<td>7.6</td>
<td>7.6</td>
<td>4.0</td>
<td>4.0</td>
<td>6.4</td>
<td>4.2</td>
<td>1.9</td>
<td>1.9</td>
</tr>
</tbody>
</table>

Table 5.1: A set of representative textures, their streaming times and signal to noise ratios.
Chapter 6

Conclusion

The design goals listed in Chapter 1, sets up various comparisons with the standard DXT compression. Now that the system has been implemented, we can evaluate each point separately.

Looking at compression ratio, the proposed streaming system outperforms DXT by a relatively large margin. The compression is three to almost ten times more effective while still keeping a PSNR above 30 dB. Defining a speed comparison is more difficult: simply streaming DXT mipmaps requires almost no CPU power but requires more bandwidth from the optical disk than the proposed progressive streaming solution.

Depending on the amount of CPU power available, the proposed streaming system can out-perform simple DXT streaming due to its superior compression and progressive nature: the 12x speed DVD drive that is built into the Xbox 360 has a maximum read speed of around 15 MB/s. Using DXT compression, this means a maximum throughput of 30 megapixels per second. Our proposed streaming system has a higher throughput in almost all cases, as seen in Table 5.1. However, this assumes that the streaming process can run uninterrupted, utilizing all the available processing power of a single core. In most game engines, this is not possible as many subsystems run on the same processing unit, reducing the effective CPU time per frame. Similarly, the DVD player will only perform at peak throughput if all data can be read consecutively from disk. In the case of streaming texture data, this is probably not the case. Additionally, the DVD drive has to stream in other data during game-play such as background music, sounds and movies. This reduces the effective bandwidth that can be used by the streaming system significantly. In the end it depends on the amount of CPU time that can be allocated to the streaming subsystem.

All things considered, I recommend Triumph Studios to adopt the Progressive Streaming: the current engine runs primarily on one or two threads, while the rest of the CPU cores are mainly used by third party libraries. When I was testing the streaming system on the Xbox 360, I had no problems finding enough processing power for the streaming system to run at significant speed. The results in Chapter 5 and the implementation on the Xbox 360 show that the current design and implementation is a viable way of streaming and compressing texture data in a real-time setting. The streaming solution will solve the memory issues, making it possible for Triumph Studios to present the player with larger game worlds, further increasing the quality of their games.
6. CONCLUSION

6.0.1 Future Work

One the biggest drawbacks of the current implementation is the fact that the system needs the DCT coefficients of the smaller mipmap maps in order to reconstruct the higher mipmap maps. This results in some additional memory overhead: instead of simply storing the DXT representation of a mipmap, the original DCT data has to be kept in memory or streamed again as soon as the higher mipmap maps need to be loaded. A solution to this problem would be the ability to use the already converted DXT mipmap as a basis. Since the DCT conversion is an additive process, this should be possible to implement this using research laid out in this paper.

There is not a lot of room for additional performance gain in the current streaming pipeline, except for small cache optimizations and other small algorithmic improvements. It is most likely not cost efficient to look into further optimizing the current design. However, there are large gains to be made by moving the computation to the GPU completely. Using pixel and compute shaders, color conversion and DCT transforms can be be run in parallel on the GPU. However, Huffman and RLE encoding is more difficult to implement in a classic shader model.

The current implementation only looked at streaming RGB textures; adding an additional alpha channel should not prove to be a problem. The additional channel would be compressed in a similar fashion as the Y (luminance) channel.
Appendix A

DXT compressed texture format

The DXT format is used by most modern graphical pipelines to store texture data in compressed form. The DXT compression scheme is known under many different names such as the S3TC (S3 texture compression), and is a subset of the BCn texture compression formats currently supported by modern GPU chip sets. It utilizes a block-based, fixed-ratio lossy compression scheme.

The format is a block based compression and has a fixed compression ratio. It was developed specifically to be easily implementable on hardware, and has subsequently been implemented in all modern 3D-acceleration hardware. The reduced texture size size results in decreased data bandwidth, which is a bottleneck for almost all graphical processing done in modern games. Because of this, the use of the DXT format results in a significant performance gain, even though the hardware decompression adds complexity to the rendering pipeline. The DXT format is the preferred format to use for almost all types of textures used in modern games, across all gaming platforms.

There are five base DXT compression schemes that make up the full DXT specification; DXT1 gives a fixed ratio compression of 4bpp (bits per pixel) while all other schemes give a compression of 8bpp. The compression is achieved by taking advantage of localized correlation between colors (and alpha values where applicable): each scheme uses $4 \times 4$ block layout, where the texture data is subdivided in blocks and each pixel block is processed individually.

A.1 DXT 1

The DXT1 scheme is identical to the BC1 block layout and is the most basic of the BCn and DXT layouts. It has a fixed compression ratio of 1:6 compared to 24-bit RGB. A 4 block layout of a BC1 format is shown in Figure A.1. The two initial color entries create the local palette in which the indices select the color for each of the 16 pixels of the block. The indices are two bits, which makes it possible to choose from four colors $c_0$, $c_1$, $c_2$ and $c_4$. The first two entries $c_0$ and $c_1$ are given as two 16-bits colors at the start of the block.

1Exceptions to this are textures that need high fidelity or use High Dynamic Range.
A. DXT COMPRESSED TEXTURE FORMAT

while $c_3$ and $c_4$ are defined by interpolation of the first two colors:

$$c_2 = \frac{2}{3}c_0 + \frac{1}{3}c_1$$

When looking at these functions, a twofold degeneracy can be observed; by swapping $c_0$ and $c_1$ and inverting the indices, any image can be represented in two different ways. This equals an additional one bit of storage; this is used by the DXT1 scheme to denote a different way of interpreting the color palette. This special case is defined as being present when $c_0$ has a bigger numerical value than $c_1$: the palette is then interpreted as having only three colors, where the last index is reserved for a fully transparent pixel; this creates support for a one-bit alpha channel for DXT1 blocks, with local loss of color fidelity. The third palette entry $c_3$ is defined in this case as $c_2 = \frac{1}{3}c_0 + \frac{2}{3}c_1$.

A.2 DXT 2, 3

The layout of DXT2 and DXT3 is similar to that of DXT1, but with an additional block of alpha information. A 64 bits data block is added before the DXT1 block, containing 4-bits alpha values for each pixel. Also, the color block is always interpreted as having four-color palette, retaining the degeneracy described above. DXT2 and DXT3 use exactly the same data structure, with the difference being the interpretation of the color values during rendering; the DXT2 colors are assumed to be pre-multiplied by alpha, where the DXT3 colors are assumed to be unprocessed. In most current rendering pipeline APIs, the sampling of textures and the blending of the resulting values are fully separated, and both DXT schemes result in the same sampled values when fed into the hardware samplers. The layout is equal to the BC2 block layout and has an fixed compression ratio of 1:4 compared to 32-bit RGBA.

A.3 DXT 4, 5

Similar to the previous DXT layouts, the DXT4 and DXT5 formats also contain a 64 bits alpha block. Instead of sixteen 4-bits entries, the layout describes the alpha data in the same way the color data is stored: using a two palette values and sixteen index values. The layout is equal to the BC3 block layout and has a 1:4 compression ratio similar to the DXT2 and DXT3 schemes. Because the alpha block contains only one channel, the two palette values are eight bits each, using 3-bits per pixel to select between eight alpha values $\alpha_0...\alpha_7$. The degeneracy is exploited similarly to the BC0 format; when $\alpha_i > \alpha_0$, the alpha values of $\alpha_6...\alpha_7$ are calculated in a similar way as the color values:

$$\alpha_n = \frac{8-n}{6}alpha_0 + (n-1)alpha_1$$

However, when the indices are in reversed order $\alpha_i < \alpha_0$, a different interpolation is used:

$$\alpha_n = \frac{8-n}{6}alpha_0 + (n-1)alpha_15 = 2 \cdot 5$$

The last two indices are then used to denote the minimum and maximum alpha values. $\alpha_6 = 0$, $\alpha_7 = 255$. 

84
For completeness, we quickly list the other BCn schemes that are widely supported: BC4 is a single channel format with the same size as BC1; it uses the same storage scheme as the alpha section of BC3. BC5 is a similar format which stores two channels in a similar way. BC6 and BC7 are relatively new and support floating point data; they use a more complex encoding scheme and can define a wide range of compression modes.

A.4 DXT Compression

Although the decompression of a DXT block is straight forward; the compression of DXT is far more involved; Finding the optimal local palette is an expensive operation which makes DXT not well suited for run-time compression. Section 4.7 details the implementation of a fast, real-time DXT encoder.

Figure A.1: The three schemes used by the DXT compression standard. Note that the fifth alpha index is split over the 32-bit boundary.

Figure A.2: A short 2D graphical representation of the basic principle of DXT compression. Two endpoints $c_0$ and $c_1$ are chosen, and two points in the middle are interpolated from those points. Finally, the color values of the 16 pixels are mapped to $c_0...c_3$. 
A.5 Software

In order to test the implementation of the DXT format, a small program was created that can encode DXT images in various ways and analyze them for any error or bias from the original image; it was used to test and evaluate the various encoding algorithms that were used.

Figure A.3: The decompression software used to perform and analyze various real-time DXT conversion algorithms.
Appendix B

Discrete Cosine Transform

The Discrete Cosine Transform (DCT) expresses any given discrete signal as a finite sum of cosine functions of different frequencies. This translates a signal over time into a frequency domain, which is useful in various applications. In the context of this document, it is used to create a lossy compression scheme; the image is transformed to the frequency domain, compressed using a lossy compression algorithm and translated back during decompression.

Figure B.1: A representation of the 64 base functions belonging to a 2D DCT transform on the left. There are various types of DCT, each describing its own set of base functions. The coefficients shown are from the DCT described in the JPEG standard.[5] On the right two images with their DCT coefficients shown beneath them.

There are multiple DCT transforms functions, each having different boundary equations. Below are the definitions of DCT-I through DCT-III. It can be seen that all three equations are similar but have differences in scaling factors and offsets. The main difference of these equations are the boundary conditions set for the various transforms. Each transform also maps on a special case of a Discrete Fourier Transform (DFT).
B. DISCRETE COSINE TRANSFORM

B.0.1 DCT-I transform

The equations below show the first three forms of DCT.

\[ X_k = \frac{1}{2} \left( x_0 + (-1)^k x_{N-1} \right) + \sum_{n=1}^{N-2} x_n \cos \left( \frac{\pi}{N-1} nk \right) \quad k = 0, \ldots, N - 1. \]  

\[ X_k = \sum_{n=0}^{N-1} x_n \cos \left( \frac{\pi}{N} \left( n + \frac{1}{2} \right) k \right) \quad k = 0, \ldots, N - 1. \]  

\[ X_k = \frac{1}{2} x_0 + \sum_{n=1}^{N-1} x_n \cos \left( \frac{\pi}{N} \left( k + \frac{1}{2} \right) n \right) \quad k = 0, \ldots, N - 1. \]

In this thesis the most common form is used: the DCT-II transform; the DCT-III transform can be used as-is as the inverse transformation. However, in this paper, the standard transform is further modified to contain a \( \frac{1}{\sqrt{2}} \) factor on the DC component and an additional overall scaling to make the transform completely orthogonal; this means the same function can be used for the inverse operation, saving in code to be optimized. However, this additional scaling breaks the direct link between the DCT-II transform and Discrete Fourier transforms. In researching the various types of DCT, I made a program that not only implements the various DCT forms and their inverses, but also shows the implicit signal symmetries and conditions that lie at their basis.

Figure B.2: DCT analysis tool, showing the various expansions of the signal using the various DCT functions.
B.1 Multidimensional DCTs

In order to process two dimensional data, a 2D version of the DCT is used. This is implemented with a simple row-column scheme in which we transform the data twice, once per rows and the resulting data per column. This straight-forward method is used in our implementation, however there are various ways of optimizing these computations further if more performance should be required at run-time.
Appendix C

Mipmapping

With the introduction of mipmapping [28], textures can increase in resolution (detail) with relatively little impact on processing power. Whereas a basic texture is stored as a single image in video memory, a mipmapped texture is stored in video memory as a so-called mipmap chain, consisting of the texture image along with a list of smaller images called mipmap levels. Starting with the texture image as mipmap 0, each following mipmap in the chain is half the size of the previous one until the lowest mipmap is reached, which consists of a single pixel. When a triangle is rasterized by the rendering hardware, the effective UV derivative between pixels is computed to select the appropriate mipmap from the mipmap chain. Mipmapping increases both quality and data-access efficiency of texture sampling: mipmap levels contain pre-filtered versions of the texture which reduces aliasing artifacts and the sampling itself is done in a more cache-friendly fashion because it is guaranteed to sample adjacent pixels at any sampling frequency. Because of these benefits, mipmapping is implemented and used by default on almost all rendering systems today.

The main drawback of mipmapping is the additional memory used by the mipmap chain; a mipmapped texture is a third larger than a non-mipmapped version: 

\[
\left(\frac{1}{4} + \frac{1}{16} + \frac{1}{64} + \frac{1}{256} + \ldots\right) = \frac{1}{3}
\]

Trilinear filtering

Switching from different detail (mipmap) levels can lead to popping artifacts. Since the used mipmap is dependent on the rendering distance, these artifacts happen in screen space rather than only between two frames. To reduce these artifacts, interpolation is used between two mipmap levels. This is called trilinear filtering, and causes the renderer to start introducing lower mipmap levels before switching over completely. Although this removes artifacts, it causes textures to become unnecessarily blurry.

Anisotropic Filtering

Anisotropic Filtering tries to resolve this problem at the cost of more video memory. For textures that are placed at oblique viewing angles with respect to the camera, the compres-
C. MIPMAPPING

Figure C.1: Example of a mipmapped texture. Notice that the second level takes up only one fifth of the texture data.

Expansion in one axis causes the filter to choose a lower mip map level, automatically losing detail along both directions. To remedy this, Anisotropic Filtering creates additional mipmaps by creating additional images per level that are only scaled differently over both axis. What ratios are generated is dependent on user settings. Higher anisotropy give sharper images on oblique angles, but increase memory usage.
Appendix D

Proposed prediction and control system

For this document we provide a baseline prediction system to base our design decisions on. This system has a coarse and fine prediction system, we will first describe the coarse system below: Note that this system is only superficially implemented, and merely used as a contextual reference during the design of the texture streaming system.

D.1 Cell based prediction

The starting point of this prediction is to split the scene into streamable sections called cells (borrowed from a similar system in portal engines [3][18]. Only a select amount of other cells is directly visible from any given cell. A cell is defined to be visible from another cell if a line of sight exists between any two points within these cells. This is translated into a cell visibility graph, where each node presents a cell, and each edge denotes a line of sight between any two points between two cells. Using this representation, we set the following rules for our streaming prediction:

1. Every visible cell should be fully loaded.
2. We assume no other state information, except the current cell the camera is in.
3. When the camera is within a cell, we know what other cells are visible.
4. We assume that given any cell, each adjacent cell is also a visible cell.
5. For each possible pair of cells, we know a minimum bound for the time it takes to travel from one to another. (subsequently, adjacent nodes have a travel time of zero)

Given these rules and the visibility graph $G$ with cells $C_x$, visibility edges and adjacency edges, we can define the Current Set $CS_x$ for each cell as containing all cells in $G$ that need to be loaded to have full detail from any view point in $C_x$ as follows.

Since we do not take the camera position within a cell into account, we have to assume the camera can move to an adjacent cell any time. Therefore, all directly adjacent cells need
to be loaded. We first define this set as the adjacency set $A_{S_c}$. The final Current set can be defined as:

$$\forall x \in A_{S_c} : A(x, c)$$  \hspace{1cm} (D.1)

Where $A(x, c)$ denotes the adjacency relation. Additionally, all cells visible from the adjacency set should be loaded; by extending the adjacency set along the visibility edges, we define the full Current Set (where $V(x, c)$ denotes the visibility relation, as we can assume the Adjacency set is a subset of the Current set).

$$\forall x \in C_{S_c} : \exists a \in V_{S_c} : V(x, c)$$  \hspace{1cm} (D.2)

However, implementing this simple rule-based system on existing scenes without proper consideration will create impractical cells and bad streaming performance. Figure D.1 shows two examples of scenes (as seen from above). Next to each scene is the visibility graph, showing visibility and adjacency between the cells. On the far right, the streaming table is shown. Each column shows a list of minimum time will take for each cell to become visible to the camera, given a current cell. It becomes clear that the layout of a scene can greatly affect the streaming performance.

Using this time-table, it is a trivial task of streaming in the data optimally. Any data that is needed immediately should be streamed in, and any room that is left should be filled with the cells that have the lowest time bound. Although this will do for a simple streaming system, we cannot define a proper LOD heuristic for our content, resulting in low detail efficiency. As can be seen from the streaming tables, a large portion of assets needs to be streamed in at any given time. In the next subsection we will describe a more fine-grained prediction model, which results in a more efficient streaming model.

### D.2 Camera based prediction

The cell based system is a good starting point for streaming when the scene is composed of clearly defined areas, and visibility graphs are relatively unconnected. In scenes that
are more open, or in cases where the streamable assets within one area are too big to be streamed in as one streamable dataset, more fine grained prediction system is required. In this section we look into a prediction model that is based on camera behavior and rendering distance rather than scene layout.

In game and other interactive 3D applications, the camera movement is directly controlled by user input (through mouse, keyboard and/or game-pad). Despite this rather unpredictable external input, the behavior of the camera is usually predictable within certain bounds. These predictions are two-fold: Firstly, there are limits that the application poses on the camera movement, such as maximum speeds and collision systems preventing the camera from moving through walls or solid objects (something that we already implicitly take into account for the minimum time bounds in the cell based prediction). Secondly, the behavior of the user is predictable up to a certain degree; for instance we can assume the user will be looking mostly forward when moving, and the camera will be focused on visual cues that are presented by the environment.

Typically, camera movement is relatively stable, with defined probability functions and maximum bounds for both translation and rotation speeds. For this paper we will assume a simple distribution: the camera we will be using in our streaming model is that of a typical navigating camera, moving mostly forwards, usually with a fixed maximum movement speed. Translating these behaviors into streaming priorities can be done in different ways, but we will use a straight-forward way of doing so: we construct various volumes around the camera, where each volume has a visibility factor. For each asset, its streaming priority is determined by visibility factor of the volume it is intersecting. The visibility factor of each volume is based on the visibility of the volume at the current time and the probability that it will be visible in the future, given a set of rotation and translation distributions.

This results in objects that are close and in front of the camera getting priority over objects that are far away or outside of the viewing frustum. The model can be enhanced by factoring in other inputs from cell based or trigger based prediction mechanics.

![Prediction volumes for directional prediction and distance prediction](image)

|Figure D.2: Prediction volumes for directional prediction and distance prediction|

Image D.2 shows fine two sets of these volumes, one where there is a set maximum rotation speed and one where there is not. The first is typical of racing games, where the viewing direction is very predictable in both movement and viewing direction. The second is closer to 3D modeling applications and games with a first-person viewpoint; the camera can move in any direction at a set maximum speed, but the viewing direction is unpredictable without a limit on rotation speed.
D.3 Interface

The described prediction system was used as a context to build our streaming engine on; the interface to our streaming system is a simple one, defined as two update functions; UpgradeTexture() and DowngradeTexture(). These were used to increase or decrease the streamable textures by one mipmap level; The system did not assume any knowledge of the internal timings of the texture streaming system, so the calls were asynchronous. Inversely, the streaming system had a simple interface for querying the current detail level of a texture GetTextureLevel(); the detail level was always defined as the internally registered desired level of detail, and not the actual or current level of detail. This way, the prediction system sees the streaming system as instantaneous, making it completely agnostic to any delays or implementation details. The prediction system therefore will not (and cannot) take memory or performance issues into account, these are handled internally by the detail management system.
Appendix E

Data Management

The main streaming design has established a data layout on a per-texture level. However, streaming every texture individually will result in a large amount of separate texture data requests, resulting in a large seek time overhead on the source media. Smart command queuing can remedy this to a point, but in the context of optical media access, data fragmentation should be avoided whenever possible. In the proposed data packing management, we use additional layers of packing and define a set of tools that can be used to create and analyze these layers.

E.1 Data Packing

We start by defining each separate use of a streamable mipmap as a Streamable Data Object (SDO). This means that each model that uses a certain texture has a separate SDO for each streamable mipmap of that texture, and SDOs of different objects can reference the same mipmap data. In addition to a reference to the mipmap data, each SDO has a footprint and a bounding box around the footprint. The footprint describes one or multiple 3D volumes that denote the areas where the SDO should be available in memory. When the camera enters the bounding box of an SDO, the streaming system should start streaming the mipmap data it is referencing. For every unique mipmap, the entire scene is subdivided in a 3D grid of boolean values called a layer. Each SDO resides in the layer of its mipmap, and all SDOs referencing the same mipmap will therefore reside in the same layer.

Multiple SDOs can be merged to form complex SDOs that contain multiple footprints connected to multiple layers. Complex SDOs will still only have a single bounding box. Non-complex SDOs are also called simple SDOs. The goal of the data packing phase is to iteratively merge SDOs until a certain ideal size per SDO is reached. Too many small SDOs will result in many small reads from disk, increasing seek times. But with each merge, the bounding box of the SDO will get less and less accurate, and the streaming less and less efficient. The idea is to create large packs with optimal correlation between the bounding box and its footprints. The bounding boxes are the structures that will eventually be used to queue the streaming of data, and will be used in the ordering of the SDOs on optical
disk. Better spatial correlation results in a more efficient streaming system, reducing the percentage of data that is loaded but not needed (at that time).

The process is separated into six steps.

**Rendering footprints** Each SDO marks the cells of its layer where it should be available in memory. All cells activated in this manner by a single SDO form its footprint.

**Bounding** Each rendered footprint is given a single axis aligned bounding box in grid coordinates.

**Flat Merge** The various SDOs within the same layer are merged into larger SDOs when possible.

**Full Merge** SDOs on different layers are merged into SDOs containing multiple footprints attached to multiple textures across multiple layers.

**SDO ordering** Before writing to disk, the packs are ordered along a Hilbert Curve, increasing correlation between linear distance of data location and spacial distance of footprints.

**Data overlaying** For each SDO, the previously ordered data is searched for possible overlap: when the same data is found in a relatively small data range, the data of the SDO is not discarded and a reference to the range is inserted instead.

![Convolution with a sphere to create footprints of the various texture levels](image)

Figure E.1: Convolution with a sphere to create footprints of the various texture levels

### E.1.1 Rendering footprints

In order to work with the large amount of texture footprints (thousands for a detailed environment), the footprints are rendered into 3D bitfields to make manipulation and correlation analysis easier. The resolution of the grid can be varied; in order to keep the amount of calculations relatively small, the gridsize should be kept to a minimum, but various resolutions should be tried to see which gives the best packing result versus rendering time. The footprint itself is a convolution between the surface geometry using the texture, and a sphere with a radius of \( d_{\min} + v_c \cdot t_d \) where \( d_{\min} \) is the minimum viewing distance for the given Level of Detail, \( t_d \) is the expected loading time of the texture, and \( v_c \) is the maximum speed.
of the camera. This ensures that from the time the camera reaches the footprint, the system has enough time to load the SDO before it is actually needed in the rendering pipeline. If the behaviour of the camera undergoes (predictable) changes such as is usually the case in cut-scenes, the footprints can vary in shape. All footprints of an SDO are rendered in a shared grid. After rendering, the footprint information is not needed anymore, and only the bounding box is stored.

![Figure E.2: Rasterizing footprints of the various mipmaps of a single textured polygon mesh](image)

**E.1.2 Flat Merge stage**

The first step in grouping SDOs is to merge simple SDOs on the same layer (thus referencing the same mipmap) into larger simple SDOs. The merging itself is as simple as just combining the bounding boxes into a new one by taking the minimum and maximum over all three axis. Merging decreases the amount of correlation so must be only applied when the reduction of the decrease in correlation weighs up against the removal of an SDO. This correlation is determined by simply counting the amount of marked cells against the amount of unmarked cells within the bounding box. The merge is only be finalized when the resulting SDO has a certain minimum correlation. Otherwise, the two SDOs are considered unmergeable at the current stage, but could be merged at a later stage, or in the Flat Merge stage after cut and shrink operations, which are described below. In addition to simply merging bounding boxes, other operations are performed in this stage. For instance, overlapping SDOs that cannot be merged are 'cut', removing overlapped pieces of the bounding box from one of the two SDOs. This means part of the footprint of one of the SDOs is reduced. Another possible operation is the 'shrink' operation, in which the bounding box of an SDO is 'shrunk' around its center when it does not accurately fits its footprint due to cut or merge operations.

These operations are performed wherever possible, and multiple iterations of the Flat Merge stage are performed until no more legal operations can be performed.
E.1.3 Full Merge stage

The Full Merge operation is basically the same as the Flat Merge, the only difference is that all operations are considered across all layers. Because of this increased complexity, an extra dimension is added to the correlation determination. In a Full Merge, the bounding box is merged in the same manner, except that the sets of layers in which the SDO is active is also merged by taking their union. The correlation ratio is then calculated by counting not the filled and empty cells in one layer, but in all layers in its active set. Instead of an average, a weighed average is calculated incorporating the data size of each mipmap. When the correlation ratio is above a certain minimum, the SDOs are merged into a larger, complex SDO. Similar to the Flat Merge stage, the Full Merge operations are repeated until no more legal operations can be performed.

E.2 Data overlaying

Before the final packs are written to disk, a final overlaying phase is designed to reduce data duplication. Because mipmaps are reused in disjoint areas of the scene, it is common that multiple complex SDOs are referencing the same mipmap data. Instead of simply duplicating the same data over several complex SDOs, the existing data is examined to see if data can reused. If the full set of mipmap data that is being referenced by a SDO already exists within a certain threshold range, the SDO data is not duplicated, but a reference to the data range is made instead. This reduces the efficiency of reading the data, but removes duplicate references. The overlaying pass can be repeated with varying threshold data ranges until the data fits within the disc space.
Data overlaying

Figure E.4: Two layers: after Flat Merges and Cuts, the boxes of different layers are merged using a Full Merge pass.

Figure E.5: Scanning the allocated data for a suitable overlay by keeping the last encountered positions of each data pack. In the figure, a minimum span of 8 blocks has been found. Instead of allocating another 3 blocks, the same data can be acquired by seeking over 8 existing blocks, reading 3 of them.
Bibliography


