Bauhaus-Universität Weimar
Faculty of Media
Degree Medieninformatik

Real-Time Extraction of Cache-friendly Isosurfaces from Volumetric Data

Bachelor's Thesis

Anton Benjamin Lammert
b. 5th October 1997 in Weimar

Matriculation Number 118592

First Referee: Prof. Dr. Bernd Fröhlich
Second Referee: Prof. Dr. Rhadamés Carmona

Submission date: 23.10.2020

Declaration of Academic Honesty

I hereby confirm that I worked on this thesis with the title
Real-Time Extraction of Cache-friendly Isosurfaces from Volumetric Data
independently and that I did not use sources of any kind other than the ones that I have acknowledged and cited.

Furthermore, I confirm that this is the first time that this thesis - in any form - is presented to a supervising staff.

Weimar, 23.10.2020

ANTON BENJAMIN LAMMERT

Abstract

Real-time isosurface extraction is a well-researched area in the field of computer graphics and scientific visualization. One of the main reasons for this is that scalar volumetric data is frequently produced and needs to be analysed interactively, especially in the domain of medical applications. The interactive visualization, however, does not only require a reasonably fast extraction of isosurfaces from volumetric data, but also the optimization of the mesh data itself to make effective use of modern graphics hardware features during rendering time. The aim of this thesis is two-fold: Firstly, we want to provide an overview of efficient mesh representations and evaluate their performance using state-of-the-art graphics hardware. Secondly, we want to make use of the findings from the first part of this thesis and apply it to our goal to allow for the real-time extraction of mesh data that can be efficiently rendered on modern graphics hardware. In the first part of this thesis we not only evaluate the efficiency of existing mesh representations but also propose new efficient mesh representation schemes, which are based on recently gained knowledge about the behavior of modern GPUs and are more efficient than other mesh representation schemes tested. In the second and main part of the thesis, we propose two modifications for marching-cubes-based isosurface extraction schemes, which are able to perform a more cache-friendly isosurface extraction with respect to a well-known representative baseline implementation on the fly.

Table of Contents

1 Introduction

The creation and analysis of volume data is crucial for the investigation of structures encoded within the volume data, that are occluded and cannot be directly detected such as internal organs or geological structures. Scalar volumetric data is created in many different domains, e.g. for medical [1] and geological application [2], real-time reconstruction from range images [3, 4] as well as physics-based simulations [5]. To visualize and analyse volume data, direct volume rendering techniques like volume raycasting, or indirect volume rendering techniques like marching cubes are commonly used. Despite recent efforts to make raycasting feasible for commodity hardware [6], the technique itself does not scale well with increased display or volume resolution. Therefore, indirect volume rendering based on extracted isosurfaces is generally still desirable. Furthermore, having an explicit polygonal surface representation allows for surface-based analysis, simulation and modification.

It is desirable to achieve high frame rates for the visualization of extracted isosurfaces in order to allow for interactive data exploration. If the volume data and the specified isovalue are static, the isosurface has to be extracted and optimized for rendering only once. Consequently, the frame rate of the visualization depends on the time spent on rendering the extracted isosurface. Since the extraction of the isosurface and the optimization of the polygonal surface representation have to be done only once, it is not necessary that they are performed in real-time. If the volume data or the specified isovalue change frequently, e.g. in interactive data exploration, real-time isosurface extraction schemes might be needed. In this case the frame rate of the visualization depends on the time spent on the isosurface extraction as well as the time spent on rendering the extracted isosurface. However, the rendering efficiency of the extracted isosurface is often disregarded since the focus of most real-time isosurface extraction schemes is on the efficiency of the extraction process. Especially in cases where the extracted isosurface is rendered multiple times, e.g. from different perspectives in the case of stereoscopic rendering, or a server-client

scenario where a powerful server extracts the mesh and a hardware-weak client renders the extracted mesh, the rendering efficiency of the extracted mesh still plays a major role. In an ideal scenario the isosurface is extracted in real-time and rendered efficiently as well. For this reason, this thesis is dedicated to the investigation of real-time extraction of isosurfaces that can be rendered efficiently.

To tackle this challenge, we first evaluate the efficiency of different mesh representation schemes. Afterwards, based on our insights about efficient mesh representations, we propose two modifications to marching-cubes-based isosurface extraction schemes, which allow for the real-time extraction of isosurfaces that can be efficiently rendered.

This thesis is structured as follows:
Chapter 2 gives an overview of articles relevant for this thesis, which mainly deal with efficient mesh representation and real-time extraction of isosurfaces.
In **Chapter 3** the reader is introduced to the topic of mesh representation as well as isosurface extraction. The introduced topics are referred to throughout the following chapters.
Chapter 4 evaluates the efficiency of existing mesh representation schemes. In addition, novel mesh representation schemes are proposed, based on recently gained insights into the behavior of modern GPUs, which improve upon already existing schemes.
In **Chapter 5** two different real-time isosurface extraction schemes that are able to produce cache-friendly isosurfaces are proposed and evaluated. To this end, a representative baseline implementation is modified in order to create cache-friendly isosurfaces in real-time.
Chapter 6 summarises the results of this thesis and reflects the limitations of different approaches. In addition, different approaches for future work are proposed.

2 Related Work

Over the years much research has been conducted to find efficient mesh representations with respect to different criteria. Early approaches viewed the problem of creating an efficient mesh representation as the problem of efficient mesh compression and decompression, in order to use less storage space and reduce the transmission time of meshes over networks. Since the geometry and the topology of a triangulated polygonal mesh can be separately represented, schemes that compress the geometry and schemes that compress the topology of triangulated polygonal meshes have been proposed [7, 8, 9].

Deering [7] proposed one of the first schemes including a compression of mesh geometry. To encode geometry efficiently, Deering uses the delta difference in vertex positions between the last vertex and the new vertex to encode the new vertex more efficiently. Furthermore, Deering introduced the concept of a generalized triangle mesh, which uses generalized triangle strips and a mesh buffer to encode the topology efficiently. Using this approach Deering was able to reduce the number of bits needed for the mesh representation by a factor of 6 to 10. The mesh buffer proposed by Deering is similar to a post-transform vertex cache in the sense that both store recently used vertices in order to avoid unnecessary operations. The post-transform vertex cache is used by the GPU to store the result of transforming a vertex from model space to view space. If the vertex reappears and the transformation result is still stored in the post-transform vertex cache, the result can be reused and the vertex does not need to be transformed again, thus avoiding calculations on the GPU. We refer the reader to **Chapter 3.3** of this thesis for a more detailed explanation of the post-transform vertex cache and the concept of triangle strips.

Hoppe [10] introduced the concept of a transparent post-transform vertex cache in combination with indexed triangles and indexed triangle strips. Transparency refers to the fact that each vertex is uniquely referenced by an index and can be directly accessed by it. This is not possible with the approach by Deering [7], because the vertex positions are encoded

by delta differences and cannot be accessed directly. The proposed post-transform vertex cache uses a FIFO-queue to store processed vertices. Furthermore, Hoppe [10] proposed a triangle stripping algorithm that produces cache-optimized triangle strips.

The concept of a transparent post-transform vertex cache that can be used with indexed triangles led to the development of cache-optimization algorithms for indexed triangles meshes. The size of the post-transform vertex cache is crucial for the creation of cache-efficient meshes, but often unknown. Forsyth [11] proposed a linear-speed cache-optimization scheme for indexed triangle meshes, that assumes an exponential falloff for the probability of a cache-hit. Due to this assumption the scheme is able to optimized the indexed triangle mesh for post-transform vertex caches of different size.

Over the years many algorithms to generate efficient triangle strips have been developed [12, 13, 14, 15, 16]. Vaněček and Kolingerová [17] compared several important triangle stripping algorithms and evaluated the rendering performance of the created triangle strips. Furthermore, Vaněček and Kolingerová evaluated the performance of cache-optimized triangle meshes. In their tests the triangle strip meshes outperformed the cache-optimized triangle meshes when a display list [18] was used for rendering. However, when an index and vertex buffer was used for rendering, the cache-optimized triangle meshes outperformed the triangle strip meshes.

Kerbl et al. [19] showed that modern GPUs do not have a global post-transform vertex cache that can be used by all threads of the GPU. Instead groups of threads, so called warps, share a local post-transform vertex cache. Furthermore, Kerbl et al. [19] showed that the post-transform cache of NVIDIA GPUs behaves similarly to a FIFO-queue of size 42 while the post-transform cache of AMD GPUs behaves similarly to a LRU-queue of size 15. For NVIDIA GPUs Kerbl et al. [19] noticed that the vertex caching behaves differently than expected when two indices that are processed within the same warp are in between different multiples of 2^{16}.

In **Chapter 4** we compare the rendering and cache efficiency of triangle strips with the rendering and cache efficiency of recent cache-optimization schemes in order to verify the findings of Vaněček and Kolingerová [17]. To investigate the efficiency of triangle fan meshes we propose and evaluate a simple triangle fanning algorithm. Furthermore, we investigate the behaviour of NVIDIA GPUs identified by Kerbl et al. [19] in more detail

and propose novel schemes for efficient mesh representations.

Since the publication of the marching cubes algorithm by Lorensen and Cline [20] many modifications to the original algorithm have been proposed to increase its efficiency, in order to make the algorithm suitable for specific use cases such as real-time isosurface extraction. Since often only few of the voxels of the volume generate geometry, many modifications to the marching cubes algorithm try to avoid processing voxels that will not generate geometry in order to extract the isosurface more efficiently [21, 22, 23]. Due to the massive parallelism of the marching cubes algorithm, many parallel processing approaches have been proposed to execute the algorithm more effectively, often making use of the SIMD nature of the GPU. To be able to extract isosurfaces from large volume data using parallel approaches, out-of-core schemes have been proposed [24]. Lakshmipathy et al. [25] proposed a scheme that extracts triangle strips from the volume data in order to improve the rendering performance of the extracted isosurface. However, their approach is not designed for efficient parallelization. Scholz et al. [26] proposed a GPU-friendly level-of-detail scheme for the marching cubes algorithm using tetrahedral and hexahedral cells. They noticed that the rendering performance of the extracted isosurface can be improved if the topology of the generated triangles inside each hexahedral cell is stored in a more cache-efficient way inside the index buffer. However, the surface extraction and hierarchy updates of the scheme proposed by Scholz et al. [26] run on the CPU. Furthermore, modifying the isovalue at run-time requires up to one second of processing time.

Chen et al. [27] proposed a GPU-based scheme using NVIDIA CUDA that is able to run in real-time and extract indexed isosurfaces. To index the generated vertices efficiently, they use a parallel prefix sum implementation. However, Chen et al. [27] do not investigate the rendering efficiency of the extracted isosurface.

A similar GPU-based scheme that extracts indexed isosurfaces using a parallel prefix sum implementation was proposed by Liu et al. [28]. The scheme proposed uses a blocking hierarchy and is able to process larger volumes than other state-of-the-art GPU algorithms, while also needing less time for the isosurface extraction. The authors showed that the extraction of indexed isosurfaces is also beneficial for storage and transmission, reducing the size of an extracted isosurface without indices from 273.6 MB to 72 MB in the indexed case. Similarly to Chen et al. [27], Liu et al. [28] do not investigate the rendering

efficiency of the extracted isosurfaces.

In **Chapter 5** we propose GPU-based isosurface extraction schemes with focus on the cache efficiency of the extracted isosurface. Similar to the work of Chen et al. [27] and Liu et al. [28] we make use of the prefix sum in order to index vertices efficiently. We define a simple cube-wise voxel processing scheme that allows for the extraction of cache-friendly isosurfaces. To the best of our knowledge, there is no published research investigating the real-time extraction of cache-friendly isosurfaces.

3 Background

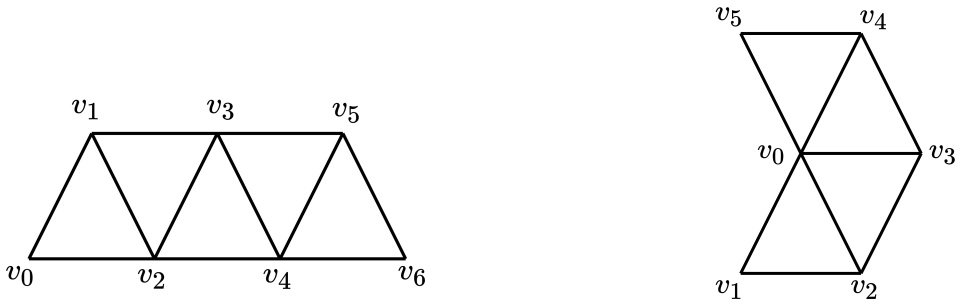
This chapter provides the reader with a non-exhaustive overview of concepts related to efficient mesh representations, namely *triangle strip encoding*, *triangle fan encoding* and *vertex cache optimization*, as well as a short overview of *isosurface extraction* with focus on the marching cubes algorithm. The core themes of the aforementioned concepts are covered, such that our approaches for efficient mesh representation in general in **Chapter 4**, and for extraction of cache-friendly isosurfaces in **Chapter 5** can be mostly understood without the necessity to know the related work in detail.

3.1 Triangle Strips

A basic polygonal mesh consists out of its geometry and its topology, which can both be represented independently of each other. To represent the topology of a triangulated mesh efficiently, the concept behind triangle strips and triangle fans is often used [9, 29, 30]. The triangulated mesh is represented as a set of non-overlapping triangle strips. A triangle strip consists of a sequence of triangles, where consecutive triangles in the strip share a common edge. This knowledge can be used to encode the topology of a single triangle strip efficiently. In practice, an index buffer is often used to encode the topology of triangle strips, but the usage of an index buffer is not compulsory. To encode a triangle strip using an index buffer the index buffer is created in such a way that three consecutive indices in the index buffer always encode a triangle of the triangle strip. The triangles in the triangle strip shown in **Figure 3.1a** can be encoded by the index sequence $v_0 - v_1 - v_2 - v_3 - v_4 - v_5 - v_6$. One can see that all triangles of the triangle strip shown in **Figure 3.1a** are encoded in this way.

Triangle strips that include zero area triangles are known as generalized triangle strips

[31]. The advantage of including zero area triangles into the triangle strip encoding is the fact that it allows for more freedom in the creation of triangle strips. For example the topology shown in **Figure 3.1b** can be encoded using the single triangle strip sequence $v_1 - v_2 - v_0 - v_3 - v_0 - v_4 - v_5$ using one zero area triangle.



(a) Triangle strip example. **(b)** Generalized triangle strip example.

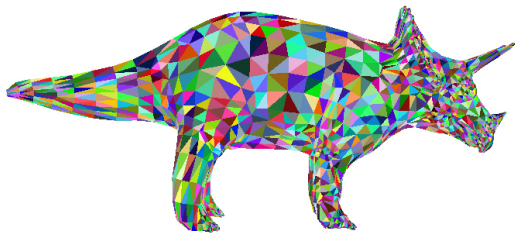
Figure 3.1: The topology shown in **(a)** can be encoded using a single triangle strip without zero area triangles. The index sequence from v_0 to v_6 encodes all triangles of the triangle strip. The topology shown in **(b)** can be encoded with a single generalized triangle strip.

To encode a triangle strip with n triangles only $n + 2$ indices are needed. In comparison $3n$ indices are needed to encode all n triangles without any information about the triangle topology. One can see that triangle strips can reduce the size of the index buffer up to a third of the original size which is beneficial for storing or transmitting the triangulated mesh. Furthermore, the GPU can utilize the topology information to process triangles more efficiently since it is clear that the two vertices which were previously worked on can be cached and reused for setting up the next triangle.

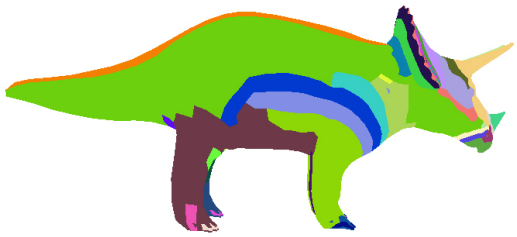
Triangle strips are supported as rendering primitives by many major graphic APIs such as OpenGL [32]. OpenGL allows the use of a single index buffer to store all triangle strips. In order to allow rendering separate triangle strips with a single draw call, OpenGL allows the definition of so called restart indices, to tell the graphics driver not to connect vertices across the restart index. We refer the reader to Shreiner et al. [32] for a more detailed explanation about primitive restarts in OpenGL.

The problem of finding an optimal separation of a triangulated polygonal mesh into triangle strips has been proven to be NP-complete [33]. However, many efficient triangle

stripping algorithms have been proposed that are able to produce good results in practice [12, 13, 34]. For an exhaustive overview of different triangle stripping algorithms we refer the reader to Vaněček and Kolingerová [17]. To evaluate the efficiency of triangle strips, two different triangle stripping algorithms are chosen in **Chapter 4.1** to generate triangle strips: STRIPE [12] and Patchification [31]. Both algorithms are greedy and were chosen because they were able to produce good results in practice, use different approaches and their implementations were made publicly available. The implementation of STRIPE that was made publicly available by Evans et al. [12] is used. For the Patchification algorithm the implementation included in the Meshoptimizer framework [35] is used. In **Figure 3.2** triangle strips generated by STRIPE and Patchification are visualized. The algorithm STRIPE tries to create long triangle strips encoding zero area triangles as well while the algorithm Patchification tries to create triangle strips in such a way that the number of encoded zero area triangles is minimized.



(a) Individually encoded triangles.
Triangle count: 16,980



(b) Triangle strips encoded by STRIPE [12].
Average strip length: 132.1 triangles.

(c) Triangle strips encoded by Patchification [31].
Average strip length: 12.5 triangles.

Figure 3.2: Visualization of jointly encoded triangles across the surface of the example model *Triceratops*. Same colors show parts of the surfaces, which are encoded in the same rendering primitive. While using triangles as rendering primitive unavoidably leads to inefficient representation of the surface with redundant encoding of vertices which could potentially be shared **(a)**, the triangle stripping algorithms STRIPE **(b)** and Patchification **(c)** lead to more compact encoding of surface parts, which leads to smaller memory footprints within a triangle strip.

3.2 Triangle Fans

The concept of triangle fans is similar to the concept of triangle strips in the sense that triangle fans also use the knowledge about the topology to encode the topology efficiently. The triangulated polygonal mesh is represented as a set of non-overlapping triangle fans. A triangle fan consists out of multiple triangles that share a common center vertex. Furthermore, consecutive triangles in the triangle fan share a common edge as as can be seen in the example in **Figure 3.3**. Similar to triangle strips triangle fans are often encoded using an index buffer but the usage of an index buffer is not compulsory. To encode triangles in a mesh as a triangle fan using an index buffer, the index of the common center vertex is placed first in the index buffer. Afterwards, the vertex indices are added to the index buffer, such that two consecutive indices always create a triangle with the index of the common center vertex. This means that the triangle fan shown in **Figure 3.3** would be encoded by the index sequence $v_0 - v_1 - v_2 - v_3 - v_4 - v_5$. One can see that all triangles of the triangle fan shown in **Figure 3.3** are encoded in this way.

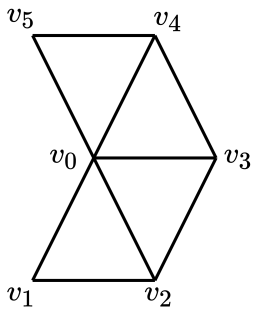


Figure 3.3: Example of a topology that can be encoded with a single triangle fan. The index sequence from v_0 to v_5 encodes all triangles of the triangle fan.

Similar to the usage of triangle strips, $n + 2$ indices are needed to encode a triangle fan with n triangles. Likewise, triangle fans are supported as rendering primitives by major graphic libraries as well, and restart indices can be used in OpenGL to store multiple triangle fans in a single index buffer.

Compared to triangle strips, triangle fans often contain a lower number of triangles. This is due to the fact that maximum number of triangles in a triangle fan is limited by the total number of triangles that share the common vertex. For closed manifold triangle

meshes the number of triangles with a common vertex is around 6 on average, which can be shown using Euler's formula [36]. Consequently, this limits the maximum size of a triangle fan to 6 triangles per fan on average. Triangle strips, however, are not limited in such a way and can achieve strips with more than 100 triangles per primitive as can be seen in **Figure 3.2b**. If the aim is to reduce the size of the index buffer for a triangulated mesh it often makes more sense to use triangle strips instead of triangle fans because of the fan size limitation.

3.3 Vertex-cache-optimal Mesh Representations

To render a polygonal mesh every vertex of the mesh has to be transformed from model to view space at least once. To avoid unnecessary recomputation GPUs often use a so-called post-transform vertex cache [37]. For the GPU to be able to reuse results of the vertex transform, it needs to be able to uniquely identify vertices. Because of this an index buffer is required to make the usage of the post-transform cache possible [1].

The post-transform vertex cache is used to store the result of the transformation from the model into view space for processed vertices. If a vertex has already been transformed and the result is still in the post-transform cache, the GPU can reuse the previously computed result. We refer to this case in general as *cache hit*. If the vertex has not been processed before or the result of the processed vertex is not contained in the post-transform cache anymore, the result has to be computed and is consequently stored in the post-transform cache. We refer to this case in general as *cache miss*.

Information about the implementation of the post-transform cache in recent GPUs is often not publicly available [19]. However, the size of the post-transform cache is often assumed to be limited. In many cases it is assumed that the post-transform cache works like a first-in-first-out (FIFO) queue, but different schemes like least-recently-used (LRU) queues are also not uncommon [19]. The performance of vertex cache optimization schemes is strongly dependent on the properties of the post-transform cache used by the GPU.

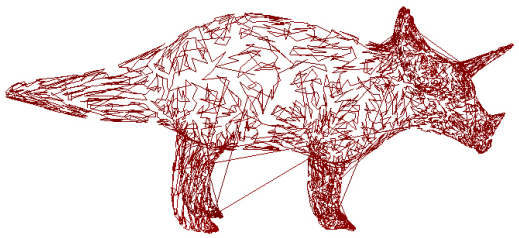
A vertex cache optimal mesh representation minimizes the number of cache misses, or, to

1 `https://www.khronos.org/opengl/wiki/Post_Transform_Cache` (Accessed on 10/08/2020)

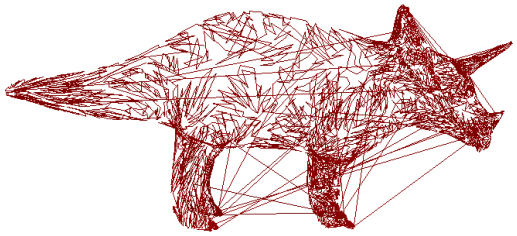
phrase it positively, maximizes the number of cache hits. In order to improve the cache-coherency, vertex cache optimization algorithms often modify the order of the indices within the index buffer. In general, reducing the distance between the occurrences of same indices in the index buffer improves the chances of having the referenced vertex still in the vertex post-transform cache.

In **Chapter 4.3** two different cache optimization algorithms are used and evaluated. The algorithm Tipsify from Sander et al. [37] that is included in the Assimp mesh library [2] and the cache optimization algorithm included in the Meshoptimizer project [35] which is based on the linear-speed mesh optimization algorithm from Forsyth [11] and the algorithm Tipsify [37]. The results of the cache optimization from both algorithms is visualized in **Figure 3.4** by connecting the center points of the triangles, which lie next to each other in the index buffer, by a line.

(a) Non cache optimized version.
Number of cache misses: 14,678.



(b) Cache optimized version using Tipsify [37].
Number of cache misses: 4,810.



(c) Cache optimized version using Meshop. [35].
Number of cache misses: 4,816.

Figure 3.4: Triangle order visualized for different versions of the *Triceratops* model. Here each line connects the center points of two consecutive triangles in the index buffer, revealing the triangle order within the index buffer.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

2 `http://assimp.sourceforge.net/lib_html/postprocess_8h.html` (Accessed on 10/09/2020)

One can see that the order of triangles in the index buffer was modified by the cache optimization algorithms in such a way that consecutive triangles in the index buffer are in spatial proximity to each other in most cases. When adjacent triangles are next to each other in the index buffer the likelihood of possible cache hits is increased due to the fact that both triangles share two vertices at the common edge.

3.4 Vertex Caching in modern GPUs

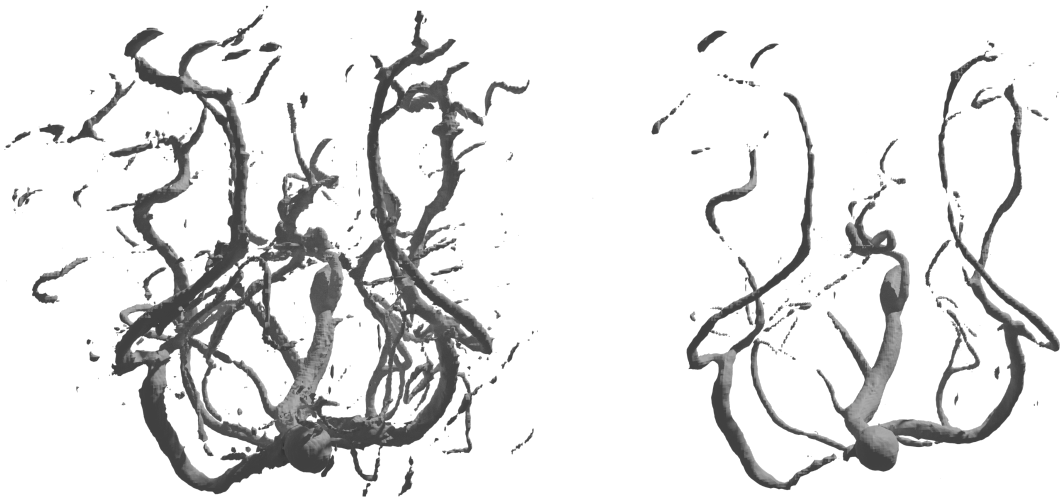
Recent research in the field of vertex caching revealed that the caching behaviour of modern GPUs is different to the caching behaviour of a global FIFO or LRU queue of a specific size [19]. Kerbl et al. [19] showed that the post-transform cache is not global and cannot be accessed by all threads of the GPU. Instead, groups of threads, so called *warps*, seem to share a common post-transform cache. Between warps it is not possible to reuse transformation results. Such a behaviour makes sense since the overhead of thread-communication for the cache access is smaller and the thread groups can operate independently from each other.

For NVIDIA GPUs a warp consists out of 32 threads. Kerbl et al. [19] showed that the post-transform vertex cache for NVIDIA warps behaves in some way similar to a FIFO-queue of size 42. However, warps can process only up to 32 unique vertices or 32 triangles. Furthermore, the vertex caching behaves differently for NVIDIA GPUs if the indices i and j are processed within the same warp and there exists a k, which is a multiple of 2^{16}, such that $(i-k)\cdot(j-k) < 0$ or respectively $\lfloor i/2^{16} \rfloor \neq \lfloor j/2^{16} \rfloor$. For AMD GPUs Kerbl et al. [19] found out that the caching behaviour can be best represented as a LRU-queue of size 15.

Using the knowledge about the vertex caching behaviour, Kerbl et al. [19] proposed a vertex cache optimization algorithm that was able to improve upon existing cache optimization schemes. In **Chapter 4.3** the findings from the aforementioned study are used to propose two different vertex cache optimization schemes.

3.5 Isosurface Extraction

In scientific research and medical applications scalar volumetrical data is frequently produced and analysed. One way to analyse scalar volumetrical data is the extraction of the isosurface of a specified isovalue. The isosurface is a three dimensional representation of all occurrences of the isovalue in the scalar volume. To approximate, extract and visualize isosurfaces, there are different approaches as presented in detail by Engel et al. [38]. One of the more commonly used approaches in indirect rendering is the marching cubes (MC) algorithm proposed by Lorensen and Cline [20] that generates a triangulated polygonal mesh representing the isosurface of a specified isovalue. The MC algorithm is popular since the surface test and the extraction itself can be efficiently implemented and the approach itself is embarrassingly parallel, because the extraction is essentially local and therefore has barely any data dependency except between eight neighboring cells. The isosurfaces shown in **Figure 3.5** were extracted by the MC algorithm in real-time on the GPU using NVIDIA CUDA demonstrating the massive parallelism of the MC algorithm.



(a) Isovalue = 0.2 **(b)** Isovalue = 0.25

Figure 3.5: Phong-shaded extracted isosurfaces of the *Vertebra* volume with 512x512x512 voxels. The isosurfaces were extracted in real-time using the NVIDIA CUDA MC implementation from the NVIDIA CUDA samples [39]. Comparing **(a)** and **(b)**, it can be seen that the extracted geometry depends strongly on the specified isovalue.

Because of its massive parallelizability many real-time isosurface extraction algorithms make use of the concept behind MC. The MC algorithm processes each voxel independently and generates geometry for each voxel depending on the scalar values at the voxel corners and the specified isovalue. Depending on the scalar value at each corner and the isovalue an index is generated that is then used to lookup the triangles that the voxel will generate in a lookup table. The vertex positions of the triangles are determined using linear interpolation along the voxel edges in such a way that the value of the scalar volume at the vertex position is equal to the isovalue. This can be seen in **Figure 3.6**.

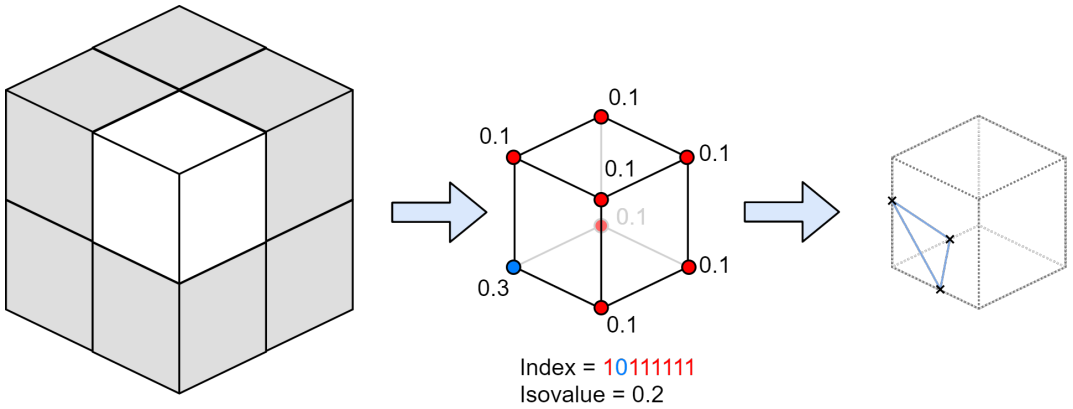


Figure 3.6: Simplified visualization of the steps of the Marching Cubes algorithm. For every voxel the sampled scalar values at the voxel corner and the isovalue determine the triangle index of the voxel. The index is then used to generate the geometry that represents the isosurface inside the voxel. Note, that the vertex positions are determined by linear interpolation along the voxel edges.

In **Chapter 5** we combine the insights of cache optimization and efficient triangle encoding in order to propose a modification to existing and already performant MC algorithms. The modification allows the extraction of cache-friendly isosurfaces in real-time, increasing the rendering performance of extracted meshes with only little reconstruction overhead.

4 Intermediate Evaluation

In this chapter different ways to efficiently render polygonal triangle meshes are tested and evaluated. **Chapter 4.1** evaluates the usage of triangle strips for efficient mesh representation. In **Chapter 4.2** a triangle fanning scheme is proposed and evaluated. Based on the findings of Vaněček and Kolingerová [17] that cache optimization can outperform the use of triangular strips, **Chapter 4.3** evaluates various existing cache optimization schemes. Furthermore, based on knowledge about the cache behavior of modern GPUs, a warp-based cache optimization scheme and an approach to improve the rendering performance of cache-optimized meshes are proposed and evaluated. In **Chapter 4.4** two approaches to improve the rendering performance by improving the vertex access on the GPU are proposed and evaluated. **Chapter 4.5** puts the findings from the previous sections into context and evaluates them.

To evaluate approaches to efficient mesh representations, a set of models from the Stanford Scanning repository (SCR) [1] is being used. The models from the SCR are often used in scientific research. This makes it possible to compare our results to the results of other related articles. Besides those models, two additional models have been used for testing. The *Triceratops* model from Viewpoint Animation Engineering (VAE) [2] which was used in several other papers [15, 34] and the *Obelisk* model courtesy of the Computer Vision in Engineering Group at Bauhaus-Universitat Weimar (BUW). All models are listed in **Table 4.1** and displayed in **Figure 4.1**.

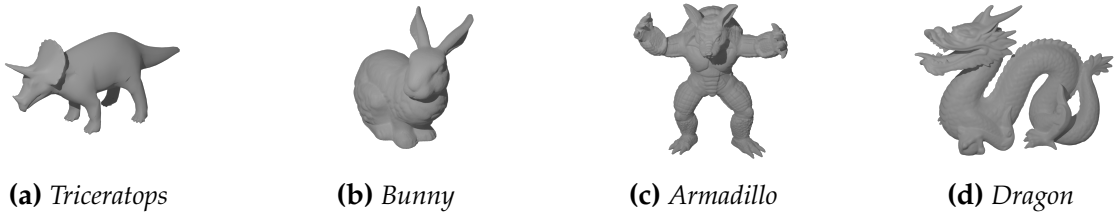
The framework used for testing was implemented in C++ and uses the OpenGL API, a widely used industry standard for hardware accelerated 3D graphics programming. In order to estimate the effectiveness of different mesh representations in terms of rendering

1 `http://graphics.stanford.edu/data/3Dscanrep/` (Accessed on 09/16/2020)
2 `https://www2.cs.duke.edu/courses/cps124/spring04/code/data/animals/triceratops.obj` (Accessed on 09/16/2020)

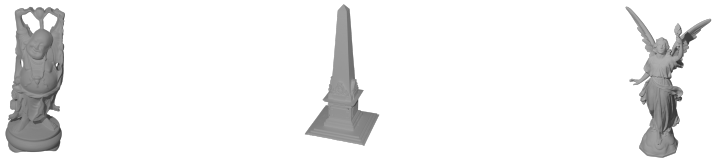
performance, we record the time spent on the graphics card for rendering the models in different representations. To measure the rendering time (RT) a OpenGL timer query is being used [40]. To have a stable result, the average RT over 1000 successive rendering calls is computed. In order to measure the number of vertex cache misses (CM) an atomic counter is being used in the vertex shader. The atomic counter is only incremented if a vertex has to be transformed because of a cache miss. After the rendering call is finished, the value of the atomic counter is read and reset to 0. The number of CMs is being measured since it is a concrete measurement for the amount of work the GPU has to do considering the vertex transform.

Model name	Vertex count	Triangle count	Source
Triceratops	2,832	5,660	VAE
Bunny	34,834	69,451	SCR
Armadillo	172,974	345,994	SCR
Dragon	434,856	871,414	SCR
Budda	546,955	1,087,716	SCR
Obelisk	11,141,077	21,462,382	BUW
Lucy	15,015,873	28,055,728	SCR

Table 4.1: Models used for the evaluation of mesh representations.



(a) *Triceratops* **(b)** *Bunny* **(c)** *Armadillo* **(d)** *Dragon*



(e) *Budda* **(f)** *Obelisk* **(g)** *Lucy*

Figure 4.1: Phong-shaded renderings of the models used for evaluation.

In the following sections the performance of different mesh representations will be evaluated. To calculate the performance difference the original rendering time and the new rendering time are used as shown in **Equation 4.1**.

$$\Delta\textbf{performance} = \frac{t_{\text{old}} - t_{\text{new}}}{t_{\text{new}}} \cdot 100\% \tag{4.1}$$

4.1 Triangle Strip performance

As explained in **Chapter 3.1** the usage of triangle strips is well known and researched. Triangle strips allow for a compression of the mesh topology and can greatly reduce the size of the index buffer. Because of this, the usage of triangle strips can improve the rendering performance and will be evaluated in the following. In the following evaluations triangle strips are be encoded in a single index buffer using a primitive restart index to separated consecutive triangle strips in the index buffer. The primitive index usage is explained in more detail by Shreiner et al. [32].

4.1.1 STRIPE

The implementation of the algorithm STRIPE that was provided by provided by Evans et al. [12] was used to create a triangle stripped model from the normal model. The algorithm takes the topology encoded through triangle indices as input and outputs triangle strips encoded using the vertex indices. Different parameters can be chosen by the user to specify the behaviour of the algorithm. The algorithm was used with its default parameters which produced the best results for the authors of STRIPE [3].

As can be seen in **Table 4.2** triangle strips improve the rendering time for nearly all test models. The triangle stripped version of the *Triceratops* model performed worse than the normal version. This might be due to an overhead created by the necessary primitive restarts in the index buffer. Since the difference in rendering time for the *Triceratops* is only small it could also be due to to variance of the rendering time.

3 `https://www3.cs.stonybrook.edu/~stripe/` (Accessed on 10/01/2020)

For the *Lucy* model the triangle strip performance was also worse than the performance of the normal version. To evaluate how triangle strips perform on a GPU with more memory, the GPU was switched to one of the newest GPUs available at the moment.

Model name	Mean Strip Length	RT Normal	RT STRIPE	ΔPerformance
Triceratops	132.1	0.0071 ms	0.0077 ms	-7.79%
Bunny	113.5	0.0678 ms	0.0369 ms	83.73%
Armadillo	76.7	0.2534 ms	0.0938 ms	170.14%
Dragon	62.2	0.2315 ms	0.1989 ms	16.39 %
Budda	62.7	0.2685 ms	0.2484 ms	8.09%
Lucy	86.5	6.2138 ms	6.5920 ms	-5.73%

Table 4.2: Triangle strip rendering performance for strips produced by STRIPE.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA GTX 1080 8GB

Model name	Mean Strip Length	RT Normal	RT STRIPE	ΔPerformance
Triceratops	132.1	0.0041 ms	0.0049 ms	- 16.32%
Bunny	113.5	0.0233 ms	0.0206 ms	13.10 %
Armadillo	76.7	0.0797 ms	0.0444 ms	79.50%
Dragon	62.2	0.1043 ms	0.0751 ms	38.88%
Budda	62.7	0.1065 ms	0.0980 ms	8.67 %
Lucy	86.5	2.1060 ms	1.6619 ms	26.72 %

Table 4.3: Triangle strip rendering performance for strips produced by STRIPE.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

Model name	CM Normal	CM STRIPE
Triceratops	14,678	5,842
Bunny	147,216	72,945
Armadillo	945,897	459,527
Dragon	1,510,625	943,297
Budda	1,801,723	1,163,470
Lucy	40,953,240	30,314,734

Table 4.4: Triangle strip cache misses for strips produced by STRIPE.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

As can be seen in **Table 4.3** the triangle stripped version performed better than the normal

version for all cases except for the *Triceratops* model. This might again be due to an overhead produced by the usage of restart indices. Since the difference in rendering time is only small it could also be due to variance of the rendering time. The number of cache misses could be reduced compared to the normal model for the entire evaluation set, as can be seen in **Table 4.4**.

For static meshes a display list (DL) can be used for rendering in OpenGL. A DL is a series of OpenGL commands that is compiled once and executed later by the GPU. Using display lists minimizes data transmission between the CPU and GPU because index buffers are not required [18]. Vaněček and Kolingerová [17] noticed that triangle strips performed better with DLs compared to rendering calls using vertex and index buffers. Because of this, the performance of triangle strips with DLs was evaluated as well. To evaluate this in our test system, a DL is created and the triangle strip is defined using the OpenGL commands *glBegin* and *glEnd*.

Model name	RT Normal	RT STRIPE	RT STRIPE DL
Triceratops	0.0041 ms	0.0049 ms	0.0049 ms
Bunny	0.0233 ms	0.0206 ms	0.0209 ms
Armadillo	0.0797 ms	0.0444 ms	0.0407 ms
Dragon	0.1043 ms	0.0751 ms	0.0717 ms
Budda	0.1065 ms	0.0980 ms	0.0896 ms
Lucy	2.1060 ms	1.6619 ms	3.0345 ms

Table 4.5: Triangle strip DL rendering performance STRIPE.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

Model name	CM Normal	CM STRIPE	CM STRIPE DL
Triceratops	14,678	5,842	5,842
Bunny	147,216	72,945	73,247
Armadillo	945,897	459,527	354,550
Dragon	1,510,625	934,297	879,990
Budda	1,801,723	1,163,470	1,097,153
Lucy	40,953,240	30,314,734	28,997,735

Table 4.6: Triangle strip display list cache misses STRIPE.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

As can be seen in **Table 4.5** the usage of a display list improved the rendering time for all models except for the *Lucy* and *Bunny* model. A possible reason could be that the DL for the *Lucy* model is too large for the GPU and data has to be re-transmitted. Another reason could be an internal mechanism of the GPU that handles large display lists differently. The number of cache misses was reduced by the DL usage for all models except the *Bunny* model as can be seen in **Table 4.6**.

4.1.2 Meshoptimizer

The Meshoptimizer project [35] offers functions for triangle strip generation as well as vertex cache-, overdraw- and vertex fetch-optimization. Furthermore, it is able to automatically optimize glTF files. Because of the variety of different optimizations offered by the Meshoptimizer project it is included into the framework and evaluated.

To generate the triangle stripped version, the functions *meshopt_optimizeVertexCacheStrip* and *meshopt_stripify* have been used. The function *meshopt_optimizeVertexCacheStrip* optimizes the mesh for strips as well as for the vertex cache while the function *meshopt_stripify* takes the indices of the normal version as input and generates a vector of triangle strips separated by the primitive restart index.

Model name	Mean Strip Length	RT Normal	RT Meshop.	ΔPerformance
Triceratops	12.5	0.0071 ms	0.0074 ms	-4.05%
Bunny	18.7	0.0678 ms	0.0363 ms	86.77%
Armadillo	13.7	0.2534 ms	0.0904 ms	180.30%
Dragon	10.9	0.2315 ms	0.1794 ms	29.04%
Budda	10.7	0.2685 ms	0.2379 ms	12.86%
Lucy	13.2	6.2138 ms	6.3779 ms	-2.57%

Table 4.7: Triangle strip rendering performance Meshoptimizer.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA GTX 1080 8GB

Model name	Aver. Strip Length	RT Normal	RT Meshop.	ΔPerformance
Triceratops	12.5	0.0041 ms	0.0048 ms	-14.58%
Bunny	18.7	0.0233 ms	0.0208 ms	12.01%
Armadillo	13.7	0.0797 ms	0.0455 ms	75.16%
Dragon	10.9	0.1043 ms	0.1186 ms	-12.05%
Budda	10.7	0.1065 ms	0.1546 ms	-31.11%
Lucy	13.2	2.1060 ms	2.6456 ms	-20.39%

Table 4.8: Triangle strip rendering performance Meshoptimizer.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

Model name	CM Normal	CM Meshop.
Triceratops	14,678	5,870
Bunny	147,216	72,283
Armadillo	945,897	513,125
Dragon	1,510,625	956,087
Budda	1,801,723	1,199,498
Lucy	40,953,240	35,087,154

Table 4.9: Triangle strip cache misses Meshoptimizer.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

The Meshoptimizer results in **Table 4.7** are similar to the results from STRIPE in **Table 4.2**. The results measured with the NVIDIA Quadro GPU in **Table 4.8**, however are very different to the results from STRIPE in **Table 4.3**. While the performance was increased by STRIPE for all models besides the *Triceratops* on the NVIDIA Quadro, the Meshoptimizer stripped version improved the performance in only two cases. However, the number of CMs is relatively similar between the STRIPE and the Meshop. stripped versions. See **Table 4.4** and **Table 4.9**. A possible reason for this might be that the GPU has some overhead when working on a new triangle strip. Since the average strip length is shorter for the Meshoptimizer stripped mesh, it contains more triangle strips than the model that was created with STRIPE. If there is some overhead for processing a new triangle strip, then the decrease in performance makes sense.

For the sake of completeness, the Meshoptimizer stripped model with DL is tested as well.

Model name	RT Normal	RT Meshop.	RT Meshop. DL
Triceratops	0.0041 ms	0.0048 ms	0.0048 ms
Bunny	0.0233 ms	0.0208 ms	0.0208 ms
Armadillo	0.0797 ms	0.0455 ms	0.0406 ms
Dragon	0.1043 ms	0.1186 ms	0.1175 ms
Budda	0.1065 ms	0.1546 ms	0.1435 ms
Lucy	2.1060 ms	2.6456 ms	4.4166 ms

Table 4.10: Triangle strip display list rendering performance Meshoptimizer.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

Model name	CM Normal	CM Meshop.	CM Meshop. DL
Triceratops	14,678	5,870	5,940
Bunny	147,216	72,283	73,156
Armadillo	945,897	513,125	367,200
Dragon	1,510,625	956,087	911,110
Budda	1,801,723	1,199,498	1,137,158
Lucy	40,953,240	35,087,154	29,880,623

Table 4.11: Triangle strip display list cache misses Meshoptimizer.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

As can be seen in **Table 4.10** the Meshoptimizer stripped model with DL performs similar with respect to the RT of the meshoptimizer stripped without DL. The DL however, performed worse in regards of the RT for the *Lucy* model, similar to the measurements from STRIPE with DL in **Table 4.5**. The DL reduced the number of cache misses for the *Armadillo*, *Dragon*, *Budda* and *Lucy* model as can be seen in **Table 4.11**.

4.2 Triangle Fan performance

Often triangle strips are used instead of triangle fans, since triangle strips often contain more triangles than triangle fans as explained in **Chapter 3.2**. However, long triangle strips might not be cache efficient. If a strip is long, a vertex that was processed in the beginning of the strip might not appear again until much later in the strip. This means that it might not be in the cache anymore when it reappears. Since the cache optimization algorithm Tipsify [37] traverses along the mesh in a fan-based manner and is very cache efficient, the question arises if it might be possible to combine both advantages. The reduced number of cache misses from the cache optimization and the reduced size of the index buffer from the triangle fan encoding.

Based on the idea of the Tipsify cache optimization algorithm we propose a simple triangle-fanning algorithm. It takes the cache optimized index buffer that was optimized by the Tipsify algorithm and finds the triangle fans within. For this it chooses the fan centers in a greedy way and then creates and encodes the fans accordingly. A visualization of the triangle fans created by the triangle fanning scheme can be seen in **Figure 4.2**.

Model name	RT Normal	RT FAN	ΔPerformance
Triceratops	0.0070 ms	0.0063 ms	12.00%
Bunny	0.0720 ms	0.0364 ms	98.02%
Armadillo	0.2528 ms	0.0957 ms	164.15%
Dragon	0.2335 ms	0.2400 ms	-2.70%

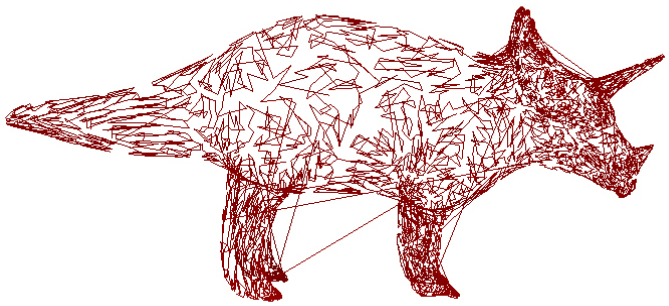
Table 4.12: Fan-based approach rendering performance.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA GTX 1080 8GB

Model name	RT Normal	RT FAN	ΔPerformance
Triceratops	0.0041 ms	0.0046 ms	-10.86%
Bunny	0.0233 ms	0.0202 ms	15.34%
Armadillo	0.0797 ms	0.1003 ms	-20.53%
Dragon	0.1043 ms	0.2157 ms	-51.64%

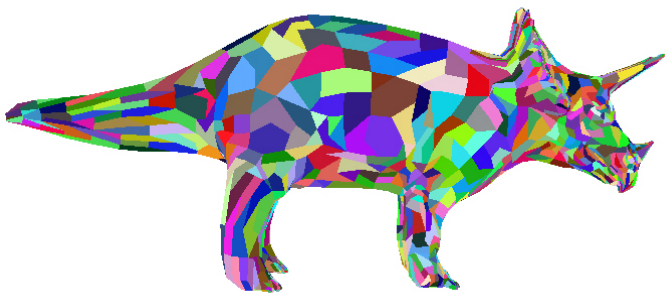
Table 4.13: Fan-based approach rendering performance.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

Model name	CM Normal	CM STRIPE	CM FAN
Triceratops	14,678	5,842	4,853
Bunny	147,216	72,945	60,609
Armadillo	945,897	459,527	510,611
Dragon	1,510,625	934,297	826,627

Table 4.14: Fan-based approach cache misses.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB



(a) Tipsify optimized triangle order of the *Triceratops* model.



(b) Fanned *Triceratops* model with 3.07 triangles per fan on average.

Figure 4.2: The triangle fans visualized in **(b)** have been created using the Tipsify cache optimized triangle order shown in **(a)**. To visualize the order of triangles in the index buffer the center points of two consecutive triangles in the index buffer are connected using a line in **(a)**. In **(b)** local patches of uniform color indicate surface regions contained in the same triangle fan. The size of the index buffer of the *Triceratops* model was reduced by the fanning from 16,980 to 11,270.

As can be seen in **Table 4.12** the fan based approach works well for the *Triceratops*, *Bunny* and *Armadillo* model but not on the *Dragon* model. On the Quadro GPU it perform worse

for all models except the *Bunny* as can be seen in **Table 4.13**. However, as can be seen in **Table 4.14** the number of cache misses is being reduced by the fanning approach compared to the cache misses of the normal model. This is similar to the behaviour of the Meshoptimizer stripped version. A possible reason could again be an internal overhead when starting a new primitive, in this case a new fan. Such an overhead could explain the drop in performance even though the number of cache misses is being reduced. If this would be the case, triangle fans might not be an option for all models anymore, since the size of a triangle fan is limited by the number of triangle around the center vertex. For closed manifold triangle meshes it is assumed that this is on average around 6, which can be shown using Euler's formula [36]. Since the triangle fanning scheme performed worse with respect to the rendering time for the *Dragon* model compared to the normal version, no larger models were used for testing.

4.3 Cache-based Optimization performance

Vertex cache optimization schemes can improve the rendering performance for triangle meshes as explained in **Chapter 3.3**. As shown by Vaněček and Kolingerová [17], they are able to outperform triangle strips in terms of rendering time. This motivates the comparison and evaluation of different vertex cache optimization schemes.

4.3.1 Tipsify

The implementation of the Tipsify cache optimization algorithm [37] that is included in the Assimp mesh library [4] was used for testing. The Tipsify algorithm runs fan-based along the surface of the mesh, processing triangles in a cache efficient way.

Model name	RT Normal	RT Tipsify	ΔPerformance
Triceratops	0.0071 ms	0.0072 ms	-1.38%
Bunny	0.0678 ms	0.0354 ms	91.52%
Armadillo	0.2534 ms	0.0907 ms	179.38%
Dragon	0.2315 ms	0.1739 ms	33.12%
Budda	0.2685 ms	0.2263 ms	18.64%
Lucy	6.2138 ms	6.0193 ms	3.23%

Table 4.15: Tipsify rendering performance.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA GTX 1080 8GB

Model name t	RT Normal	RT Tipsify	ΔPerformance
Triceratops	0.0041 ms	0.0047 ms	-12.76%
Bunny	0.0233 ms	0.0199 ms	17.08%
Armadillo	0.0797 ms	0.0457 ms	74.39%
Dragon	0.1043 ms	0.0739 ms	41.13%
Budda	0.1065 ms	0.0893 ms	19.26%
Lucy	2.1060 ms	1.9830 ms	6.20%

Table 4.16: Tipsify rendering performance.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

4 `https://www.assimp.org/` (Accessed on 10/10/2020)

Model name	CM Normal	CM Tipsify
Triceratops	14,678	4,810
Bunny	147,216	60,168
Armadillo	945,897	528,967
Dragon	1,510,625	826,006
Budda	1,801,723	1,038,803
Lucy	40,953,240	33,128,706

Table 4.17: Tipsify cache misses.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

On both GPUs used for testing the cache optimization results in an improved rendering time for all models except the *Triceratops* model as can be seen in **Table 4.15** and **Table 4.16**. The number of cache misses is reduced by Tipsify for all models compared to the number of cache misses for the normal version as can be seen in **Table 4.17**.

Since DLs were able to improve the rendering performance of the triangle stripped version for some models the usage of DLs for triangle meshes was investigated as well. While investigating DLs with triangles instead of triangle strips it turned out that DLs use the cache if the number of unique vertices is below 2^{16}. Since no indices were defined, the expected number of cache misses is three times the number of triangles. However, the measured number of cache misses is lower than the expected number of cache misses if less than 2^{16} vertices are used. This suggests that for the usage of DLs that use less than 2^{16} vertices some kind of internal indexing mechanism is being used by the NVIDIA GPU. If more than 2^{16} vertices are being used the measured number of cache misses is equal to the expected number of cache misses.

4.3.2 Meshoptimizer

As mentioned in **Chapter 4.1.2** the Meshoptimizer project offers functions for vertex cache-, overdraw- and vertex fetch-optimization. The functions *meshopt_optimizeVertexCache*, *meshopt_optimizeOverdraw* as well as *meshopt_optimizeVertexFetch* were used to optimize the mesh. As can be seen in **Table 4.18** and **Table 4.19** the Meshoptimizer optimization improved the rendering time for all models except the *Triceratops* model. Similar to the

results from the Tipsify cache optimization in **Table 4.17** the number of cache misses was reduced by the optimizations of the Meshoptimizer framework for all models as can be seen in **Table 4.20**.

Model name	RT Normal	RT Meshop.	ΔPerformance
Triceratops	0.0071 ms	0.0072 ms	-1.38%
Bunny	0.0678 ms	0.0364 ms	86.26%
Armadillo	0.2534 ms	0.0733 ms	245.70%
Dragon	0.2315 ms	0.1750 ms	32.28%
Budda	0.2685 ms	0.2263 ms	18.64%
Lucy	6.2138 ms	6.0608 ms	2.52%

Table 4.18: Meshoptimizer rendering performance.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA GTX 1080 8GB

Model name	RT Normal	RT Meshop.	ΔPerformance
Triceratops	0.0041 ms	0.0045 ms	-8.88%
Bunny	0.0233 ms	0.0205 ms	13.65%
Armadillo	0.0797 ms	0.0412 ms	93.44%
Dragon	0.1043 ms	0.0882 ms	18.25%
Budda	0.1065 ms	0.1042 ms	2.20%
Lucy	2.1060 ms	1.9807 ms	6.32%

Table 4.19: Meshoptimizer rendering performance.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

Model name	CM Normal	CM Meshop.
Triceratops	14,678	4,816
Bunny	147,216	60,648
Armadillo	945,897	360,424
Dragon	1,510,625	1,044,079
Budda	1,801,723	1,228,452
Lucy	40,953,240	30,180,313

Table 4.20: Meshoptimizer cache misses.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

4.3.3 Warp-based Cache Optimization

Based on the knowledge about the warp behaviour from Kerbl et al. [19] we propose a warp-based cache optimization algorithm. This is motivated by the fact that Tipsify and Meshoptimizer do not take the concrete behaviour of an NVIDIA warp into consideration. The basic idea behind the optimization is similar to the idea behind the Tipsify algorithm from Sander et al. [37]. It uses a queue to keep track of unprocessed vertices that are part of an already processed triangle. A vertex is processed if all triangles that use the vertex have been processed. However, unlike the Tipsify algorithm, the scheme searches for the vertex that was referenced most often and has not yet been processed. The idea behind this behaviour is to process vertices as fast as possible. This can be beneficial because a cache miss can occur if the vertex is not processed and re-appears later again. Furthermore, the scheme also checks if processing a triangle would mean that a new warp would be used. If this is the case we select the next vertex that is the new best option and do not process the triangle. The cache optimized mesh is then split into parts of 2^{16} unique vertices.

Model name	CM Normal	CM Tipsify	CM Warp-based
Triceratops	14,678	4,810	4,920
Bunny	147,216	60,168	60,166
Armadillo	945,897	528,967	550,628
Dragon	1,510,625	826,006	858,605
Budda	1,801,723	1,038,803	1,087,865
Lucy	40,953,240	33,128,706	34,791,735

Table 4.21: Own Warp-optimization scheme cache misses.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

The warp-based optimization scheme does not outperform the cache optimization of the Tipsify algorithm as one can see in **Table 4.21**. However, schemes that are based on the knowledge about the vertex caching behaviour of the GPU have the potential to improve upon existing schemes. This can be seen by the warp-based cache optimization algorithm created by Kerbl et al. [19] that outperformed existing vertex cache optimization algorithms in some cases.

4.3.4 Mesh Splitting

To reproduce the results of Kerbl et al. [19] mentioned in **Chapter 3.4**, a simple test was created to measure the vertex caching behaviour of the GPU.

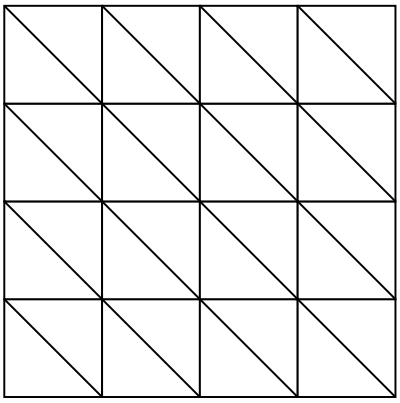
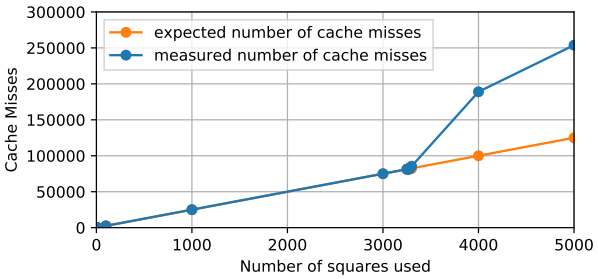


Figure 4.3: 4x4 square grid made out of 25 vertices and 32 triangles. Each NVIDIA warp can process up to 32 triangles or 32 vertices [19] meaning that the square grid can be processed by exactly one warp.

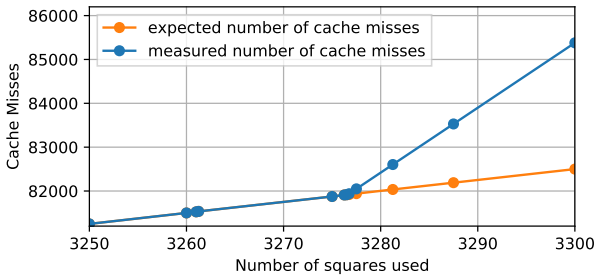
The vertex cache testing program renders meshes formed from 4x4 square grids, divided into 32 triangles as shown in **Figure 4.3**. Using the knowledge from Kerbl et al. [19] we know that the square shown in **Figure 4.3** will be processed by exactly one warp. Since each warp has a cache that behaves similar to a FIFO queue of size 42, we can arrange the triangles of each square grid in the index buffer such that each square grid generates 25 cache misses, which is the number of unique vertices used to create the square grid. When multiple squares are used they will connect with each other at the bottom/top of the square. If we render 2 squares they will consist out of 45 vertices and 64 triangles. For 3 squares 65 vertices and 96 triangles will be used. We will render multiple squares and record the number of cache misses and compare it to the expected number of cache misses.

If less than 2^{16} unique vertices are being used, the measured number of cache misses is equal to the expected number of cache misses, with 25 cache misses per square. However, the number of cache misses increases around 2.9 times as fast as expected if more than 2^{16} unique vertices or respectively 3,276 squares are being used as can be seen in **Figure 4.4**. When more than 17,500 squares are being used the measured number of cache misses per square was 72 which is much more than the expected 25 cache misses per square.

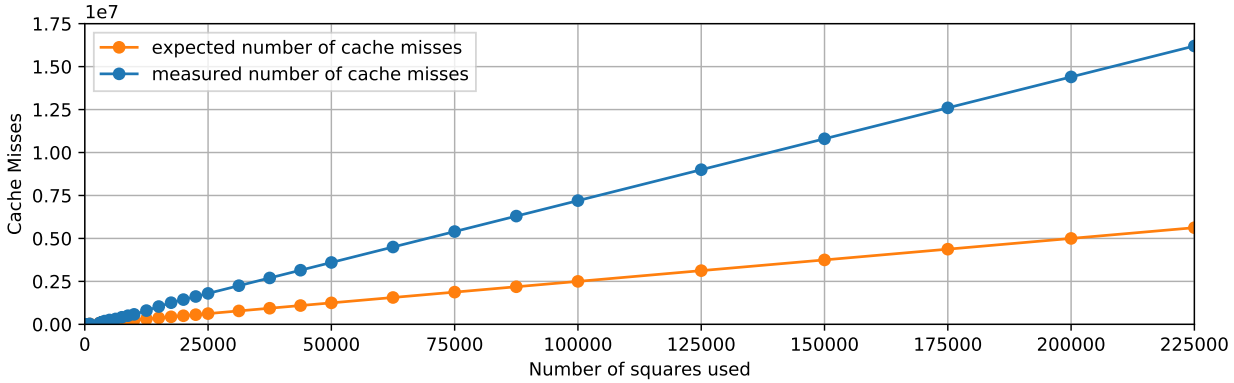
Using 4 horizontal triangle strips to build up each square and measuring the number of cache misses produced similar results to the measured cache misses from the square test with triangles. However, the number of cache misses per square was slightly lower with 69.77 cache misses per square. One reason for this might be the fact that the GPU can use the topological knowledge that it can cache the last two processed vertices for the next triangle if the current triangle strip is not left.



(a) Expected and measured number of cache misses with up to 5,000 used squares.



(b) Expected and measured number of cache misses between 3,250 and 3,300 used squares.



(c) Expected and measured number of cache misses with up to 225,000 used squares.

Figure 4.4: Expected and measured number of CMs for the 4x4 triangulated grid test. The measured number of cache misses is equal to the expected number of cache misses if less than 3,276 squares are used as can be seen in **(a)** and **(b)**. If more squares are used the number of measured cache misses is around 2.9 times the number of expected cache misses as can be seen in **(c)**.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA GTX 1080 8GB

This increased number of cache misses if the mesh has more than 2^{16} vertices motivates the idea to split the model into parts of 2^{16} or less unique vertices. For this, the result of the cache optimization from Tipsify is being used. The splitting works in a greedy way by going through the index buffer. We add elements from the index buffer into the index

buffer of the split part until 2^{16} unique vertices are used by the split part. The indices and the vertex buffer are then adjusted accordingly. We then start creating a new split part and continue going through the index buffer of the original mesh that was optimized by Tipsify. Here GL_UNSIGNED_INT is being used as type for the indices in the index buffer.

Model name	Triangle count	Number of split parts
Triceratops	5,660	1
Bunny	69,451	1
Armadillo	345,994	3
Dragon	871,414	8
Budda	1,087,716	10
Lucy	28,055,728	261

Table 4.22: Number of mesh parts after the splitting procedure.

Model name	RT Normal	RT Tipsify	RT Tipsify & Split
Triceratops	0.0071 ms	0.0072 ms	0.0072 ms
Bunny	0.0678 ms	0.0354 ms	0.0367 ms
Armadillo	0.2534 ms	0.0907 ms	0.0810 ms
Dragon	0.2315 ms	0.1739 ms	0.1684 ms
Budda	0.2685 ms	0.2263 ms	0.2236 ms
Lucy	6.2138 ms	6.0193 ms	6.0139 ms

Table 4.23: Split GL_UNSIGNED_INT rendering performance.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA GTX 1080 8GB

Model name	RT Normal	RT Tipsify	RT Tipsify & Split
Triceratops	0.0041 ms	0.0047 ms	0.0046 ms
Bunny	0.0233 ms	0.0199 ms	0.0202 ms
Armadillo	0.0797 ms	0.0457 ms	0.0403 ms
Dragon	0.1043 ms	0.0739 ms	0.0746 ms
Budda	0.1065 ms	0.0893 ms	0.0880 ms
Lucy	2.1060 ms	1.9830 ms	1.7741 ms

Table 4.24: Split GL_UNSIGNED_INT rendering performance.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

Model name	CM Normal	CM Tipsify	CM Tipsify & Split
Triceratops	14,678	4,810	4,810
Bunny	147,216	60,168	60,168
Armadillo	945,897	528,967	298,478
Dragon	1,510,625	826,006	740,672
Budda	1,801,723	1,038,803	925,546
Lucy	40,953,240	33,128,706	24,448,505

Table 4.25: Split GL_UNSIGNED_INT cache misses.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

The number of parts with up to 2^{16} vertices produced by the splitting can be seen in **Table 4.22**. One can see in **Table 4.23** and **Table 4.24** that the splitting improved the performance for some models. The increase in performance is most noticeable for the *Lucy* model in **Table 4.24**. In **Table 4.23** the performance of the *Bunny* model was not improved. However, since the model consists out of less than 2^{16} unique vertices it was not split. The difference in rendering time might be due to variance. As can be seen in **Table 4.25** the number of cache misses was reduced for all models with more than 2^{16} vertices that have been split.

If the mesh is split into parts of 2^{16} or less vertices, we can use GL_UNSIGNED_SHORT as type for the indices in the index buffer instead of GL_UNSIGNED_INT. This is possible since GL_UNSIGNED_SHORT uses 16 bits and thus allows for indices up to 2^{16}. Switching to a different type might potentially change the behaviour of the GPU, which motivates the following measurements.

Model name	RT Normal	RT Tipsify & Split Short	ΔPerformance
Triceratops	0.0041 ms	0.0047 ms	-12.76%
Bunny	0.0233 ms	0.0197 ms	18.27%
Armadillo	0.0797 ms	0.0412 ms	93.44%
Dragon	0.1043 ms	0.0657 ms	58.75%
Budda	0.1065 ms	0.0595 ms	78.99%
Lucy	2.1060 ms	1.1534 ms	82.59%

Table 4.26: Split with short rendering performance compared to normal version.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

Model name	RT Tipsify & Split	RT Tipsify & Split Short	ΔPerformance
Triceratops	0.0046 ms	0.0047 ms	-2.12%
Bunny	0.0202 ms	0.0197 ms	2.53%
Armadillo	0.0403 ms	0.0412 ms	-2.18%
Dragon	0.0746 ms	0.0657 ms	13.54%
Budda	0.0880 ms	0.0595 ms	47.89%
Lucy	1.7741 ms	1.1534 ms	53.81%

Table 4.27: Split with short rendering performance compared to Split with int. Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

Switching the index type from GL_UNSIGNED_INT to GL_UNSIGNED_SHORT improved the rendering-time for nearly all models as can be seen in **Table 4.27**. The *Lucy* and *Budda* model performed much better when GL_UNSIGNED_SHORT instead of GL_UNSIGNED_INT was used as can be seen in the same table. It has to be mentioned that the rendering time was not reduced for the *Armadillo* and the *Triceratops* model. However, the difference in performance was only small as can be seen in **Table 4.27**. A difference in cache misses was not measured. The performance improvement might be due to the fact that the GPU is not required to execute some kind of mechanism that it employs if the number of unique vertices is larger than 2^{16}. The existence of such an (as of yet unidentified) mechanism was presumed by Kerbl et al. [19]. As one can see in **Table 4.26** the rendering time for the *Lucy* model was nearly halved by the split version with GL_UNSIGNED_SHORT.

Since splitting and using GL_UNSIGNED_SHORT produces good results, the question arises if the triangle strip version can be improved in such a way as well. For this, we split the model in advance into parts of roughly 2^{16} faces as is shown for the *Armadillo* model in **Figure 4.5**. The split parts are then being stripped by the STRIPE algorithm [12]. Afterwards, the stripped results are loaded into the framework and GL_UNSIGNED_SHORT is being used as index type.

Model name	RT Normal	RT STRIPE	RT Split Short & STRIPE
Armadillo	0.0797 ms	0.0444 ms	0.0434 ms
Dragon	0.1043 ms	0.0751 ms	0.0708 ms
Budda	0.1065 ms	0.0980 ms	0.0857 ms

Table 4.28: First split then strip rendering performance.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

Model name	CM Normal	CM STRIPE	CM Split Short & Strip
Armadillo	945,897	459,527	355,459
Dragon	1,510,625	943,297	880,351
Budda	1,801,723	1,163,470	1.096,185

Table 4.29: First split then strip cache misses.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

(a) *Armadillo* model before splitting. **(b)** *Armadillo* model after splitting.

Figure 4.5: Phong-shaded visualization of the mesh splitting used for the split and strip test for the *Armadillo* model. The original *Armadillo* model is shown in **(a)**. In **(b)** the different mesh parts created by the splitting are visualised. Triangles that belong to the same split part are colored with the same color.

As can be seen in **Table 4.28** the rendering time of the triangle strips has been improved by splitting. The number of cache misses is being reduced as well as can be seen in **Table 4.29**. Since the split and then triangle stripped version with short indexing did not outperform the cache optimized and then split version with short indexing (see **Table 4.26**) no further models were evaluated.

Splitting the mesh into parts of 2^8 vertices and using GL_UNSIGNED_BYTE for the vertex indexing was tested as well. Since it performed very badly on larger meshes as can be seen in **Table 4.30**, no further tests were done.

Model name	RT Normal	RT Tipsify	RT Split Int	RT Split Short	RT Split Byte
Obelisk	1.3001 ms	1.2920 ms	1.1831 ms	0.8277 ms	30.1976 ms

Table 4.30: Rendering time of different representation versions compared with each other for the *Obelisk* model. Note that the cache optimized, split version with unsigned byte indices performs worse than all other versions.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

4.4 Improving Vertex Fetching performance

Before a warp can process geometry, the vertices of the geometry have to be fetched from the vertex buffer. For NVIDIA GPUs threads in a warp are executed in lock-step [5]. If the vertex access is executed in lock-step as well, vertices that are used by a warp should be close to each other in the vertex buffer. If this is not the case the threads would have to execute multiple vertex fetch instructions to fetch all necessary vertices. Spending more time on the vertex fetching means that the performance will not be optimal. This motivates the idea of improving the vertex fetch in order to improve the performance.

Reordering vertices

A possible approach to improve the vertex access is to reorder the vertices within the vertex buffer. To reorder the vertices a greedy algorithm is used. It goes through all indices in the index buffer and creates a index-mapping for each index. The index that appears first will be mapped to 0, the index that appears second to 1 and so on as can be seen in **Figure 4.6**. Since the input is cache optimized the occurrences of an index in the index buffer are most likely close to each other. If this is the case, vertices that will be fetched by a warp should also be close to each other in the vertex buffer. To test the vertex reordering, the cache optimized result of Tipsify that was split into part of 2^{16} unique vertices and uses GL_UNSIGNED_SHORT is being used as input.

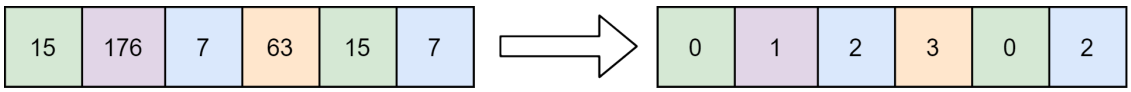


Figure 4.6: Visualization of the idea behind the vertex reordering process. On the left a part of the original index buffer is shown. On the right the index buffer after the vertex reordering is show. Indices that refer to the same vertex are colored in the same color. Note that the vertices referenced in the index buffer are closer to each other in the vertex buffer after the mapping.

5 `https://developer.nvidia.com/content/life-triangle-nvidias-logical-pipeline` (Accessed on 09/16/2020)

Model name	RT Normal	RT Split short	RT Split short reordered
Triceratops	0.0041 ms	0.0047 ms	0.0046 ms
Bunny	0.0233 ms	0.0197 ms	0.0195 ms
Armadillo	0.0797 ms	0.0412 ms	0.0397 ms
Dragon	0.1043 ms	0.0657 ms	0.0644 ms
Budda	0.1065 ms	0.0595 ms	0.0573 ms
Lucy	2.1060 ms	1.1534 ms	1.1648 ms

Table 4.31: Tipsify split and reordered rendering performance.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

Model name	CM Normal	CM Split short	CM Split short reordered
Triceratops	14,678	4,810	4,810
Bunny	147,216	60,168	60,168
Armadillo	945,897	298,478	298,478
Dragon	1,510,625	740,672	740,672
Budda	1,801,723	925,546	925,546
Lucy	40,953,240	24,448,505	24,448,505

Table 4.32: Tipsify split and reordered cache misses.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

The vertex reordering does not affect the number of cache misses for the cache optimized and split mesh as can be seen in **Table 4.32**. In all but one case the rendering time was improved by the vertex reordering as can be seen in **Table 4.31**. For *Lucy* the rendering time did not improve.

If the model is only being cache optimized but not split into part of 2^{16} unique vertices the reordering influences the caching behaviour. As can be seen in **Table 4.34** the number of CMs is sometimes being reduced by the reordering if the model has more than 2^{16} unique vertices. For the *Budda* and *Dragon* model the number of cache misses was increased by the reordering. This can be seen in **Table 4.33** as well, since the *Budda* and *Dragon* model perform worse after the reordering in regards of the rendering time. For The *Armadillo* and the *Lucy* model the rendering time was improved by the reordering.

Model name	RT Normal	RT Tipsify	RT Tipsify reordered
Triceratops	0.0041 ms	0.0047 ms	0.0047 ms
Bunny	0.0233 ms	0.0199 ms	0.0203 ms
Armadillo	0.0797 ms	0.0457 ms	0.0403 ms
Dragon	0.1043 ms	0.0739 ms	0.0784 ms
Budda	0.1065 ms	0.0893 ms	0.0922 ms
Lucy	2.1060 ms	1.9830 ms	1.8380 ms

Table 4.33: Tipsify and reordered rendering performance.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

Model name	CM Normal	CM Tipsify	CM Tipsify reordered
Triceratops	14,678	4,810	4,810
Bunny	147,216	60,168	60,168
Armadillo	945,897	528,867	341,802
Dragon	1,510,625	826,006	882,676
Budda	1,801,723	1,038,803	1,069,286
Lucy	40,953,240	33,128,706	26,687,766

Table 4.34: Tipsify and reordered cache misses.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

The difference in cache misses between the cache optimized version and the cache optimized version with reordered vertices shows that the order of the vertices in the vertex buffer can strongly influence the number of cache misses when using NVIDIA GPUs. Interestingly, the reordering only affects the number of cache misses of models with more than 2^{16} vertices.

Duplicating vertices

The idea of duplicating vertices comes from the fact that different warps do not share a common vertex post-transform cache. This means if vertex *i* is used in two different warps, it has to be transformed in each warp at least once. Because of this, the question arises if it makes sense to duplicate vertices that are used in different warps in order to have a better vertex fetch at the cost of a larger vertex buffer. This means if the same vertex is used by two or more warps it will be duplicated in the vertex buffer in such a way that the vertex fetching is improved. This can be done by iterating through the index buffer and assigning all vertices used in a warp a new index as can be seen in **Figure 4.7**. The model that was cache optimized with the Tipsify algorithm has been used as input for the vertex duplication scheme.

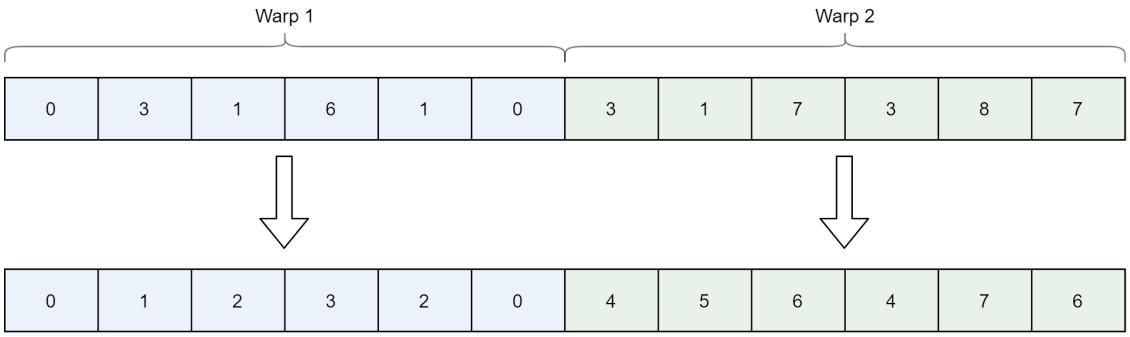


Figure 4.7: Visualization of the vertex duplication process. Here each warp processes six indices or respectively two triangles. One can see that the vertices that are used within each warp are next to each other in the vertex buffer after the duplication process. Note, that after the duplication, indices 1 and 4 refer to the duplication of the vertex which originally had the index 3.

Model name	Vertex Count Normal	Vertex Count Tipsify duplicated
Triceratops	2,832	4,787
Bunny	34,834	59,958
Armadillo	172,974	297,211
Dragon	434,856	736,135
Budda	546,955	919,342
Lucy	15,015,873	24,352,428

Table 4.35: Vertex count of the original model and the model optimised by Tipsify and modified by the duplication process.

Model name	RT Normal	RT Tipsify	RT Tipsify duplicated
Triceratops	0.0041 ms	0.0047 ms	0.0045 ms
Bunny	0.0233 ms	0.0199 ms	0.0206 ms
Armadillo	0.0797 ms	0.0457 ms	0.0409 ms
Dragon	0.1043 ms	0.0739 ms	0.0732 ms
Budda	0.1065 ms	0.0893 ms	0.0886 ms
Lucy	2.1060 ms	1.9830 ms	1.9492 ms

Table 4.36: Tipsify and duplicated vertices rendering performance.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

Model name	CM Normal	CM Tipsify	CM Tipsify duplicated
Triceratops	14,678	4,810	5,195
Bunny	147,216	60,168	65,318
Armadillo	945,897	528,967	323,704
Dragon	1,510,625	826,006	787,804
Budda	1,801,723	1,038,803	985,628
Lucy	40,953,240	33,128,706	26,275,942

Table 4.37: Tipsify and duplicated vertices cache misses.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

As can be seen in **Table 4.36** the rendering time was improved by the vertex duplication for all models but the *Bunny* model. The measured cache misses in **Table 4.37** show a similar behaviour. For all models with more than 2^{16} vertices the number of cache misses was reduced by the vertex duplication scheme. For the *Triceratops* and *Bunny* models the number of cache misses was increased. It has to be taken into consideration that the vertex buffer for the duplicated version holds a lot more elements than for the base model as can be seen in **Table 4.35**. If duplicating vertices for improved fetching can reduce the number of cache misses if the model has more than 2^{16} unique vertices it can be presumed that the unknown mechanism that NVIDIA GPUs employ depends on the vertex fetching in some way. This would be supported as well by the measurements from **Table 4.34**.

4.5 Evaluation

Out of all of the mesh representation schemes that were evaluated, the version that was cache optimised by Tipsify, split into parts of 2^{16} vertices, encoded with unsigned short integer indices, and with reordered vertices, performed best regarding the rendering time for all but two models. For *Lucy*, the Tipsify cache optimized split version with unsigned short integer indexing performed best and for the *Triceratops* model the normal version performed best. The reason why the normal version performed best for the *Triceratops* model might be the small size of the model. Furthermore, the differences between the rendering times of all versions was small for the *Triceratops* model and could be due to variance.

The Tipsify cache optimized split version had the lowest number of cache misses for all models but one. For the *Bunny* model, our proposed warp-based cache optimization scheme performed best. However, the difference in cache misses between the warp-based approach and the Tipsify cache optimized version for the *Bunny* model was only small.

The optimal number of cache misses is exactly the number of vertices of the model that are used since every vertex needs to be transformed at least once. We can compare the vertex count with the cache misses measured for the Tipsify cache optimized split version to see how close the results are to the optimal vertex reuse. For this we compute a factor as follows: $\textbf{Factor} = \frac{\textbf{CM measured}}{\textbf{CM optimum}}$. The closer the computed factor is to 1 the closer the results are to the optimum.

Model name	CM optimum	CM Tipsify Split short	Factor
Triceratops	2,832	4,810	1.698
Bunny	34,834	60,168	1.727
Armadillo	172,974	298,478	1.725
Dragon	434,856	740,672	1.703
Budda	546,955	925,546	1.692
Lucy	15,015,873	24,448,505	1.628

Table 4.38: CM optimum compared to measured number of CM.

As can be seen in **Table 4.38** the Factor is around 1.7 for all tested models.

Long triangle strips as produced by STRIPE performed well on the Quadro RTX GPU for all models. Display lists seemed to perform poorly for large models. Triangle strips with and without display lists were outperformed by the cache optimization and splitting scheme with unsigned short integer indexing, confirming the findings of Vaněček and Kolingerová [17] that triangle strips can be outperformed by cache optimization schemes on modern GPUs.

The unknown mechanism in NVIDIA GPUs that is employed if more than 2^{16} vertices are being used seems to depend on the vertex ordering inside the vertex buffer. Investigating the behaviour further might yield valuable insight into the behaviour of the unknown mechanism that could be used to improve the performance of models with more than 2^{16} vertices.

5 Cache-friendly Isosurface Extraction

Isosurface extraction plays a significant role in many research areas and areas of application more specifically, it plays a role in visualising large volume datasets generated from medical scans. We refer the reader to **Chapter 3.5** for a brief introduction about isosurface extractions and the marching cubes algorithm.

In the following chapter we introduce changes to an existing baseline implementation of the marching cubes algorithm from Lorensen and Cline [20], that improve the rendering performance of the reconstructed isosurface. To explain the necessary changes in the baseline implementation the pipeline of the baseline implementation is explained in **Chapter 5.1**. Different schemes to improve the rendering and extraction performance are proposed and evaluated in **Chapter 5.2**. In **Chapter 5.3** the proposed schemes are put into context and evaluated regarding their applicability.

5.1 Real-time Volumetric Data Extraction using Marching Cubes

A vast number of GPU-based isosurface extraction algorithms [41, 42, 43] are available on the internet. Besides different extraction approaches, they differ in the level of control they give the developer over the graphics card. For the sake of working with a well-crafted baseline implementation and having as much control over the graphics hardware as possible, we decided to investigate and modify the marching cubes CUDA sample which is provided with current packages of the NVIDIA CUDA SDK [39].

The marching cubes (MC) implementation writes the generated geometry into a vertex buffer without the use of indices. However, related work and our experiments (see

Chapter 4) both reveal, that modern graphics hardware benefits from indexed meshes in order to achieve greatly increased rendering performance compared to plain rendering from vertex buffers without additional indices. To understand the modifications we introduce to our reference implementation, it is necessary to understand the main steps of the baseline MC implementation in the first place. In the following we will illustrate (see **Figure 5.1**) and provide an overview of these main stages. We assume that a few basic concepts of the CUDA execution model, such as thread- and block-ID are known. For an introductory explanation of the CUDA terminology, we refer the reader to Sanders and Kandro [44].

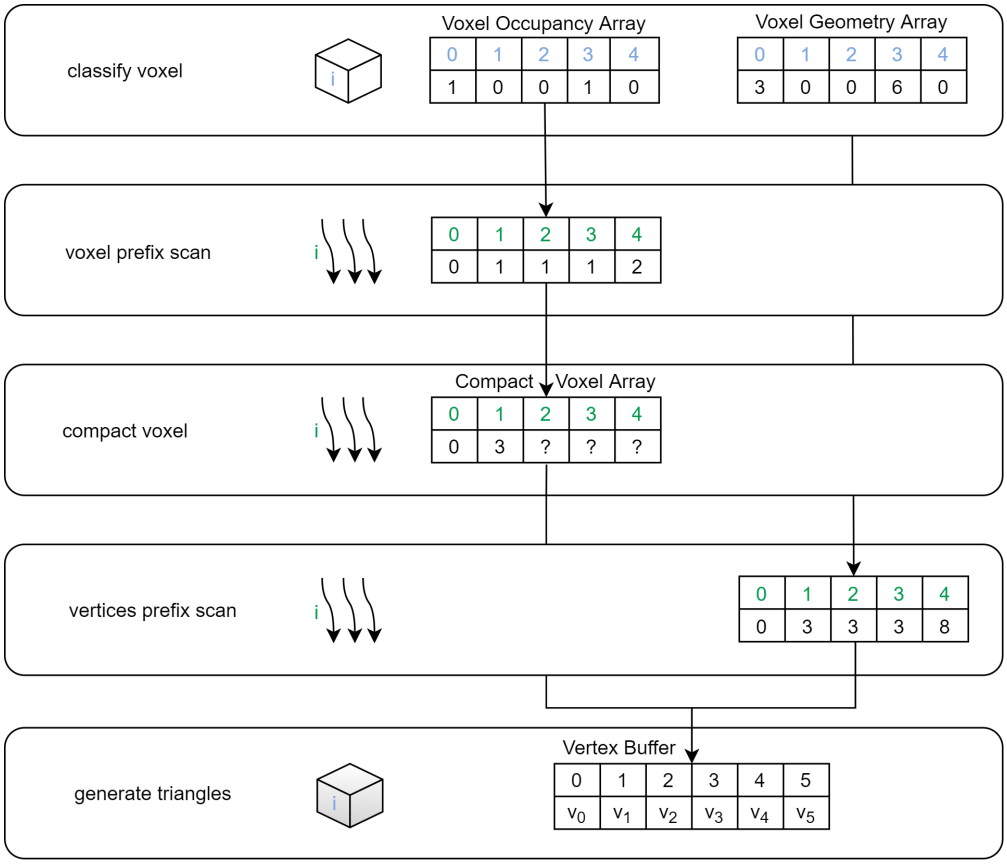


Figure 5.1: Stages of the MC CUDA baseline implementation.

In the first step the volume data is being loaded. Without loss of generality the maximum value of the volume in each dimension is limited to be a power of 2. This is due to the fact that that throughout the different CUDA kernels bitwise operations are being used to replace modulo operations that might be inaccurate for larger modulo values [45]. For the voxel access a unique ID is derived from the block- and thread-ID of a CUDA

thread. This unique voxel ID (vID) is then being used to compute the voxel position (vPos) at which the thread performs the isosurface extraction as can be seen in **Figure 5.2**.

```
1 vPos.x = vID & (max_x - 1);
2 vPos.y = (vID >> ld_max_x) & (max_y - 1);
3 vPos.z = (vID >> (ld_max_x + ld_max_y)) & (max_z - 1);
```

Figure 5.2: Voxel ID to voxel position mapping, where max_x, max_y, max_z are the maximum number of voxels in x-,y- and z-direction and ld_max_x, ld_max_y, ld_max_z are the base 2 logarithmic results of the maximum values in x-,y- and z-direction. Note that max_x, max_y and max_z are a power of 2.

For this mapping a substring of the binary representation from the voxel ID is used to derive the position of the voxel for each dimension. This can be best explained by the example shown in **Figure 5.3**.

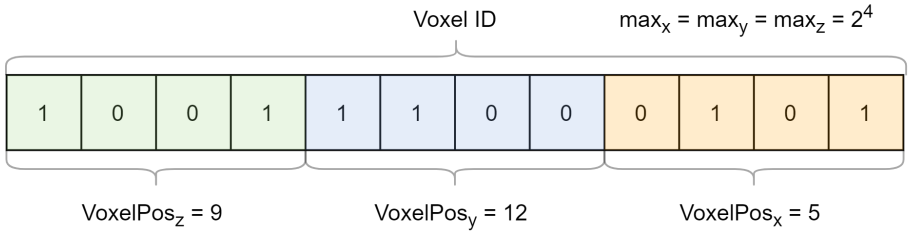


Figure 5.3: NVIDIA MC CUDA voxel ID to voxel position mapping example. Here the voxel position (5,12,9) is being derived from the voxel ID value 2501 or 100111000101 in binary.

To analyse which voxels will generate geometry and how many elements in the vertex buffer each voxel will generate the *classify voxel* kernel (see **Figure 5.1**) is being used. The information if a voxel will generate any geometry is stored in the voxel occupancy array. If the voxel will produce geometry the value of the voxel ID in the voxel occupancy array will be set to 1, else to 0. A voxel geometry array stores the information about how many elements in the vertex buffer each voxel will use. As an example in **Figure 5.1**, the voxel with ID 3 (blue labels) is flagged to produce geometry since the bit in the occupancy array is set to one, and the voxel geometry array contains the number 6 since the voxel will write as many vertices into the vertex buffer. The voxel with the ID 0 will also produce geometry and write 3 elements into the vertex buffer.

To be able to work only with voxels that generate geometry, an exclusive prefix sum scan is performed on the voxel occupancy array. The CUDA implementation uses the exclusive prefix sum scan function from the Thrust library [46]. The Thrust library is a library providing high-level access for massively parallel execution of an assortment of algorithms on the GPU.

The result from the exclusive prefix sum scan is then used by the *compact voxel* kernel that creates a compact voxel array which stores the voxel IDs of all voxels that will produce geometry. The position of the voxel ID in the compact voxel array is given by the value of the occupied voxel in prefix sum array. In **Figure 5.1** the voxels with the voxel IDs 0 and 3 are occupied. The voxel with voxel ID 0 will be stored at position 0 and the voxel with voxel ID 3 will be stored at position 1 in the compact voxel array.

To determine the positions in the vertex buffer that each voxel will use, an exclusive prefix sum scan is performed over the voxel geometry array. The result of this prefix sum scan is an array containing the positions at which each voxel will start writing the associated extracted geometry data into the vertex buffer. In the example in **Figure 5.1** the thread working on the voxel with ID 0 will start writing at position 0 in the vertex buffer. The thread working on the voxel with ID 3 will start writing at position 3 in the vertex buffer.

The *generate triangles* kernel is then used to generate the vertices for each occupied voxel and write them into the vertex buffer. Afterwards OpenGL interopability is being used to work with the vertex buffer.

Although the MC CUDA sample is quite flexible the current implementation comes with some limitations. The MC implementation is limited for volumes up to size 512x512x512. This is due to the fact that the CUDA 1D-textures which are used to store the volume data can only contain up to 2^{27} elements [47]. Larger volumes can be sampled by switching to CUDA 3D textures instead which allow to store up to 2048x2048x2048 or respectively 2^{33} elements [47], as we verified within the sample application.

Testing the MC CUDA baseline sample, the Thrust prefix sum scan was not able to process the voxel occupancy array for volumes larger than 512x512x512. Since prefix

sum scans are necessary to compute the position in the vertex buffer that will be used, custom solutions might be needed for larger volumes. This can be avoided if an array with the voxel IDs of all occupied voxels is given a priori. Such a priori knowledge could either be generated in an offline volume preprocessing stage, or generated by real-time online extraction pipelines, e.g. the brick occupancy structure by Jakob Wagner [48].

5.2 Cache-efficient Real-Time Isosurface Extraction

Cache efficiency plays an important role for the MC algorithm in several ways. Since the volume data has to be accessed frequently but only part of the volume can be stored in the cache the topic of cache-efficient volume access is well known and researched [49]. To improve the cache efficiency of the volume data access the volume can be organized into so-called bricks as introduced by Parker et al. [50]. Since the data of neighbouring voxels is better localised by the bricks the number of cache misses for the volume access is reduced. This allows for a faster extraction since less data has to be loaded into the cache.

Furthermore, cache efficiency is important when the result of the MC algorithm is being rendered. Isosurface extractions that produce a cache-efficient mesh representation are not well researched. The importance of creating cache-efficient representations however, seems not to be unknown and was addressed by Scholz et al. [51]. Scholz et al. [51] propose a level-of-detail isosurface extraction algorithm based on tetrahedral and hexahedral cells. The hexahedral cells are being reordered along a three-dimensional Hilbert curve to improve the cache efficiency of the rendering. Scholz et al.[51] were able to moderately reduce the rendering time using this approach. However, their implementation offloads the isosurface extraction as well as the hierarchy updates to the CPU. Furthermore, updating the isovalue requires up to 1 second of computation time on the CPU.

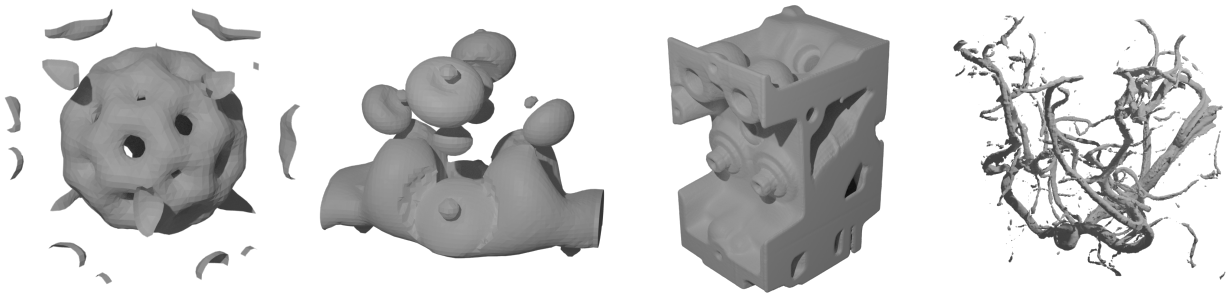
For the use case of streaming cache optimized meshes extracted from live extractions or interactive isovalue modification for indirect volume rendering, mesh optimization during isosurface extraction becomes crucial. To the best of our knowledge, no scalable solution has been proposed for those scenarios.

In the following, we will propose a set of approaches addressing the lack of such algorithms which not only offer a good trade-off between optimal isosurface extraction and rendering time, but are also possible to implement as an add-on to existing marching cubes implementations with a reasonable amount of effort, as we demonstrate in this thesis.

Different approaches to create cache-efficient meshes in real-time using MC will be proposed and evaluated. To evaluate the performance of those approaches volumes of different size from the volume library - VL [1] have been used with a fixed isovalue (iv) of 0.2f and the number of cache misses (CM) as well as the rendering time (RT) and extraction time averaged over 10.000 frames have been measured. The used volumes listed in **Table 5.1** all use 8-bit rectilinear grids. The phong-shaded reconstructed isosurfaces can be seen in **Figure 5.4**. To measure the number of cache misses an atomic counter was used (see **Chapter 4** for a short explanation). For the measurement of the rendering time a OpenGL timer query was used [40].

Volume name	Dimensions	Triangle Count at iv = 0.2f	Source
Bucky	32x32x32	14,501	VL
Neghip	64x64x64	30,698	VL
Engine	256x256x256	622,168	VL
Vertebra	512x512x512	591,958	VL

Table 5.1: Volumes used for testing.



(a) Bucky **(b)** Neghip **(c)** Engine **(d)** Vertebra

Figure 5.4: Phong-shaded renderings of the extracted isosurfaces from the volumes used for testing at iv = 0.2f.

1 `http://schorsch.efi.fh-nuernberg.de/data/volume/` (Accessed on 09/16/2020).

5.2.1 Global Indexing

To be able to use the vertex post-transform cache efficiently the use of vertex indices is required[2]. Since the MC CUDA baseline implementation does not use indices they have to be introduced into the pipeline.

To index each vertex the position of the vertex in the volume can be used. This mapping from 3D to 1D is shown in **Figure 5.5**.

```
1 index = (2 * x) + (2 * y * max_x) + (2 * z * max_x * max_y)
```

Figure 5.5: Mapping from the vertex position to the vertex index. Here x,y and z are the position of the vertex in the volume and either whole numbers or whole numbers $+0,5$. Max_x and max_y are the volume dimensions (in voxels) in x- and y-direction.

The indices defined in **Figure 5.5** can be used to save the vertices in the vertex buffer and reference the vertices using the index buffer. However, one can see that this comes with the disadvantage of having mostly unused vertex slots in the vertex buffer. Furthermore, the maximum number of indices grows in a cubic manner with respect to the dimensions of the volume. This means that the approach does not scale well since very large vertex buffers might be required. Since number of voxels that actually produce geometry is often significantly smaller than the number of total voxels in the volume, we propose an approach to index the vertices that is less conservative with the required space in the vertex buffer. The indexing scheme that we propose is similar to the indexing scheme proposed by Chen et al. [27] in the sense that both make use of an additional prefix sum scan in order to index vertices efficiently.

To index only vertices that are used one can use an exclusive prefix sum scan. For this an index usage array is used to keep track of whether an index will be used. If the index will be used, the value of the index is set to 1 and to 0 otherwise. The prefix sum scan on the index usage array provides a mapping from the old indices to the new indices in such a way that the vertex buffer only consists out of vertices that will be used. In **Figure 5.6** this mapping is shown using a simple example. I.e. here the index 5 will be mapped to the new index 3.

2 `https://www.khronos.org/opengl/wiki/Post_Transform_Cache` (Accessed on 10/08/2020)

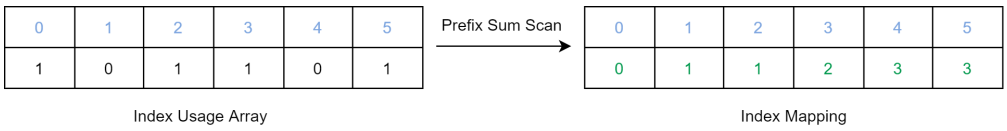


Figure 5.6: Visualization of the index mapping with an exclusive prefix sum scan. In this example the original index 5 is mapped to the new index 3.

After the integration of the additional index prefix scan and auxiliary structures, the modified MC isosurface extraction pipeline may be illustrated as shown in **Figure 5.7**. The *generate triangle* kernel was adjusted accordingly to create an index and vertex buffer as shown in the same figure. The information generated by the vertex prefix scan is used in this cache-friendlier marching cubes extraction to determine the positions in the index buffer used by the threads working on the voxels.

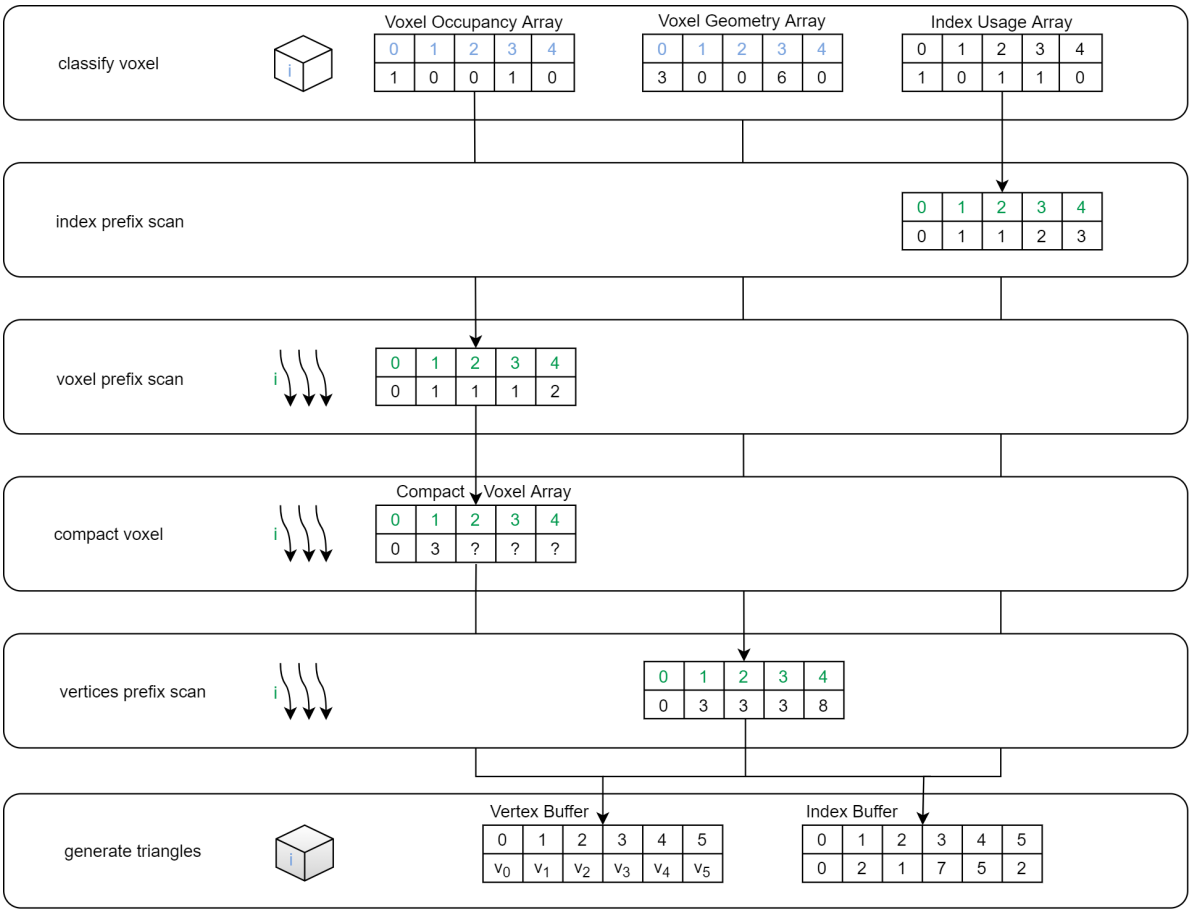


Figure 5.7: Cache-friendly MC extraction pipeline created by modifying the baseline implementation. Note that all vertices in the vertex buffer are being used in the index buffer.

To compare the performance of the global indexing scheme the number of cache misses was measured and compared with the number of cache misses for the baseline implementation of the MC algorithm. For the baseline implementation the number of cache misses is equal to the size of the vertex buffer since no indices are being used. To have a baseline the extracted isosurface was exported into an .obj file that was then optimized offline with the Tipsify cache optimization algorithm [37] as implemented by the Assimp mesh library [3]. This allows for a comparison between the cache efficiency of the real-time extraction with the result of an offline vertex cache optimization.

Model name	CM Baseline	CM global indexing	CM Tipsify
Bucky	43,524	18,928	12,315
Neghip	92,094	38,973	25,263
Engine	1,866,504	861,597	564,613
Vertebra	1,775,874	747,830	473,832

Table 5.2: Measured cache misses for the global indexing scheme compared to the number of cache misses for the baseline CUDA implementation and the Tipsify cache optimized mesh.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

Model name	RT Baseline	RT global indexing
Bucky	0.02682 ms	0.02737 ms
Neghip	0.02097 ms	0.02039 ms
Engine	0.14002 ms	0.07381 ms
Vertebra	0.07024 ms	0.03749 ms

Table 5.3: Measured rendering time of the baseline implementation and the global indexing scheme in ms.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

As can be seen in **Table 5.2** the number of cache misses could be reduced by the global indexing scheme compared to the baseline implementation. The global indexing scheme also lead to improved rendering times for the Neghip, Engine and Vertebra volume as can be seen in **Table 5.3**. For the Engine volume the rendering time was nearly halved. However, the global indexing scheme is not cache optimal as can be seen by the lower number of cache misses for the Tipsify optimized meshes in **Table 5.2**. Although the

3 `https://www.assimp.org/` (Accessed on 10/10/2020)

measurements in **Table 5.3** partially show great improvements for the larger volumes, they do not incorporate the isosurface extraction time. Since the isosurface is being computed for every frame the processing time for the extraction is important as well and therefore will be discussed in the following.

Version	classify voxel	index scan	voxel scan	vertex scan	compact voxel	generate triangles
Bucky:						
Baseline	25 μs	0 μs	419 μs	198 μs	3 μs	4 μs
Global Index	28 μs	199 μs	409 μs	170 μs	4 μs	5 μs
Neghip:						
Baseline	29 μs	0 μs	402 μs	207 μs	4 μs	4 μs
Global Index	31 μs	242 μs	423 μs	217 μs	4 μs	5 μs
Engine:						
Baseline	26 μs	0 μs	1,422 μs	676 μs	6 μs	7 μs
Global Index	15 μs	4,216 μs	423 μs	557 μs	3 μs	4 μs
Vertebra:						
Baseline	10 μs	0 μs	5,694 μs	3,267 μs	6 μs	5 μs
Global Index	12 μs	30,775 μs	2,165 μs	3,108 μs	4 μs	6 μs

Table 5.4: Measured processing times for the different stages of the extraction pipeline of the baseline version and the global indexing scheme.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

As one can see in **Table 5.4** the index scan performs worse for larger volumes. The explanation for this is that the size of the index usage array grows cubic with the volume dimension. Since the index scan takes around 30 ms for the Vertebra volume one can see that this approach might become inefficient for larger volumes and therefore does not scale arbitrarily without further modification.

5.2.2 Cube-wise Voxel processing

As can be seen in **Table 5.2** the number of cache misses for the global indexing is non-optimal and and leaves room for improvement. To be more cache-coherent the order of triangles in the index buffer can be modified. In the baseline MC implementation, the order of triangles in the index buffer is defined by the voxel ID. This means that the geometry extracted from the voxel with ID 0 will use the first slots in the index buffer. The next slots are used for the geometry of the voxel with voxel ID 1 and so on. One can see that the mapping from the voxel ID to the voxel position is crucial, since it defines in which order triangles are located in the index buffer. Using the baseline mapping shown in **Figure 5.2** voxels are processed along lines parallel to the x-axis. This however, might not be cache-efficient since voxels along a line share vertices only with the next and previous voxel on the line. To be more cache-efficient we introduce a cube-wise processing manner. Here, the volume is split into non-overlapping cubes that contain multiple voxels. Consequently, processing a cube means processing all the voxels inside of a cube. The main idea behind the cube-wise processing is that voxels inside a cube share more vertices and are processed in such a way that the generated geometry is more compactly located in the index buffer compared to globally indexed voxels. This means that references to the same vertex might be closer to each other in the index buffer. A similar idea was used by Parker et al. [50] to create a more cache-efficient texture access. To avoid wrong results from the modulo-operator bit-wise operations are used in the mapping defined in **Figure 5.9**. This however, limits the cube dimensions to be a power of 2. To evaluate the cache efficiency cubes of different sizes were used and the number of cache misses were measured accordingly. One can see the cube-wise processing visualized in **Figure 5.8**.

Model name	CM Global Ind.	CM Cube 2x2x2	CM Cube 4x4x4	CM Cube 8x8x8	CM Cube 16x16x16
Bucky	18,928	14,888	14,795	15,749	16,492
Neghip	38,973	31,006	30,776	33,043	33,097
Engine	861,597	655,923	632,421	673,442	701,901
Vertebra	747,830	625,644	617,649	643,296	649,294

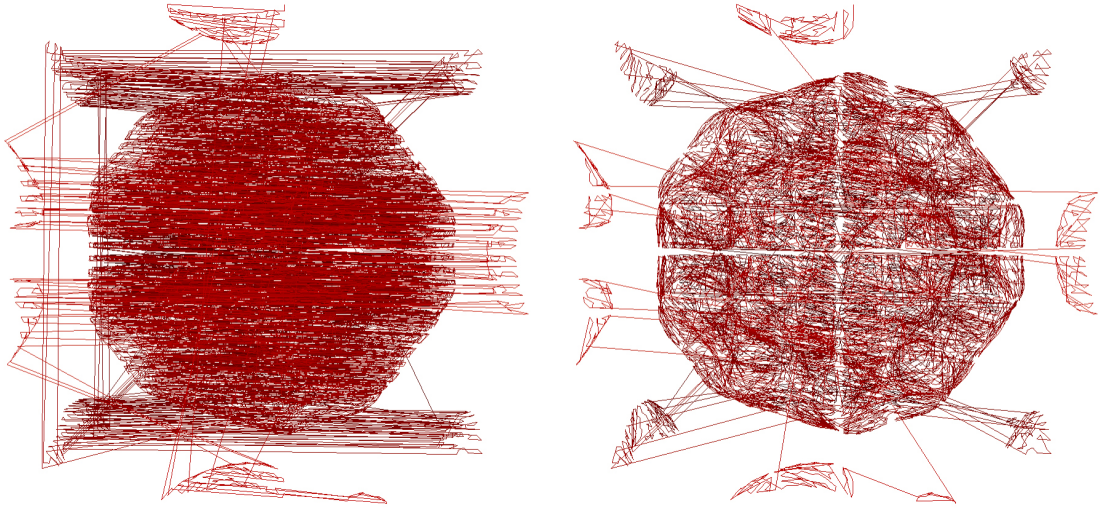
Table 5.5: Recorded number of cache misses for cubes of different sizes.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

As can be seen in **Table 5.5** the cube-wise processing defined by **Figure 5.9** lead to a reduction of the number of vertex post-transform cache misses for all tested cube sizes.

Cubes of size 4x4x4 performed best and reduced the number of cache misses from 861.597 to 632.412 for the Engine volume. Using cubes of size 4x4x4 the rendering time could be reduced further for all volumes as can be seen in **Table 5.6**. Since only the mapping from voxel ID to voxel position was modified the isosurface extraction times did not change.

Model name	RT Baseline	RT global indexing	RT Cube 4x4x4
Bucky	0.02682 ms	0.02737 ms	0.02463 ms
Neghip	0.02097 ms	0.02039 ms	0.01860 ms
Engine	0.14002 ms	0.07381 ms	0.06721 ms
Vertebra	0.07024 ms	0.03749 ms	0.03683 ms

Table 5.6: Measured rendering time in ms.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB



(a) global indexing **(b)** global indexing width cubes of size 4x4x4

Figure 5.8: Triangle order visualized using lines for the isosurface extraction of the Bucky volume with iv = 0.2f. Here the center points of two consecutive triangles in the index buffer are connected by a line.

```
1  // maximum number of cubes in x- and y-direction
2  x_max_cubes = max_x / cube_dim;
3  y_max_cubes = max_y/ cube_dim;
4
5  // maximum number of cubes in the xy plane
6  xy_max_cubes = x_max_cubes * y_max_cubes;
7
8  // x,y and z specify the local position of the voxel inside the cube
9  x = vID & (cube_dim - 1);
10 y = (vID / cube_dim) & (cube_dim - 1);
11 z = (vID /(cube_dim*cube_dim)) & (cube_dim - 1);
12
13 // the number of the cube that the voxel belongs to
14 cube_count = vID / (cube_dim * cube_dim * cube_dim);
15
16 // the offset values specify the global position of the cube
17 x_offset = (cube_count * cube_dim) & (max_x - 1);
18 y_offset = ((cube_count / x_max_cubes) * cube_dim) & (max_y - 1);
19 z_offset = ((cube_count / xy_max_cubes) * cube_dim) & (max_z - 1);
20
21 // the global voxel position is the combination of the local voxel
22 // position inside the cube and the global position of the cube
23 vPos.x = x + x_offset;
24 vPos.y = y + y_offset;
25 vPos.z = z + z_offset;
```

Figure 5.9: Voxel ID to voxel position mapping using cube-wise processing, where max_x, max_y, max_z are the maximum number of voxels in x-,y- and z-direction and cube_dim specifies the number of voxels for each cube in x-,y- and z-direction. Note that cube_dim as well as max_x, max_y and max_z are required to be a power of 2.

5.2.3 Local Indexing

As one can see in **Table 5.4** the index scan is computationally expensive for larger volumes. For this reason, it is necessary to find an alternative approach without the use of an additional prefix sum scan or an alternative approach to perform a prefix sum scan

efficiently for larger volumes sizes than the ones discussed so far. To avoid the additional prefix sum scan the indexing can be based on the compact voxel array. The compact voxel array allows to access only voxels that actually produce geometry. Since we know that each voxel will never use more than 12 vertices, we can assign each vertex inside an occupied voxel an index as shown simplified in **Figure 5.10**. Note, that voxels are still processed in a cube-wise processing manner.

```
1  // given the current thread-ID compute the base index that will be
2  // used for the voxel indexing
3  base_index = thread_ID * 12;
4  // given the current thread-ID get the voxel ID of the occupied voxel
5  // stored in the compact voxel array
6  current_voxelId = compact_voxel_array[thread_ID];
7  // for each vertex compute the index
8  for(int i = 0; i < 12; ++i){
9      vertex_index[i] = base_index + i;
10 }
```

Figure 5.10: Simplified local voxel indexing without vertex reuse across voxels.

The first occupied voxel will use the indices 0 to 11, the second occupied voxel the indices 12 to 23, and so on. Using this indexing scheme, it is possible to reuse the same vertices for triangles extracted from the same voxel. However, vertices are not shared across voxels. The vertex reuse across voxels can be enabled by introducing a scheme for the index selection that is based on the knowledge about neighbouring voxels. If a voxel and its neighbour are both occupied, both voxels might share one or more used vertices. If this is the case, vertex reuse across voxels can be implemented if both voxels agree on a common index for a shared vertex. To test the efficiency of this approach a basic scheme was used. Let (x,y,z) be the position of the current occupied voxel. If the voxel at position (x-1,y,z) is also occupied both voxels agree to use the indices from the voxel at position (x-1,y,z) for the shared vertices between both voxels. If the voxel at position (x,y-1,z) is occupied as well, the voxels at position (x,y,z) and (x,y-1,z) agree to use the indices from the voxel (x,y-1,z) for the vertices shared between both voxels. This scheme might be improved by checking more neighbouring voxels however, this will also increase the complexity of the kernel since each voxel can have up to 28 neighbouring voxels. To be able to utilize this scheme the inverse mapping for the mapping shown in **Figure 5.9** was used. This inverse mapping

is necessary since we need the voxel ID of the neighbouring voxel, given its position to check if the neighbouring voxel is occupied. The idea behind the proposed indexing scheme is similar to the idea behind the indexing scheme from Liu et al. [28] in the sense that indices are borrowed from neighbouring voxels.

Model name	CM Baseline	CM global indexing	CM Cube 4x4x4	CM Local Indexing
Bucky	43,524	18,928	14,795	19,287
Neghip	92,094	38,973	30,776	40,615
Engine	1,866,504	861,597	632,421	817,598
Vertebra	1,775,874	747,830	617,649	789,242

Table 5.7: Measured cache misses for the local voxel indexing.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

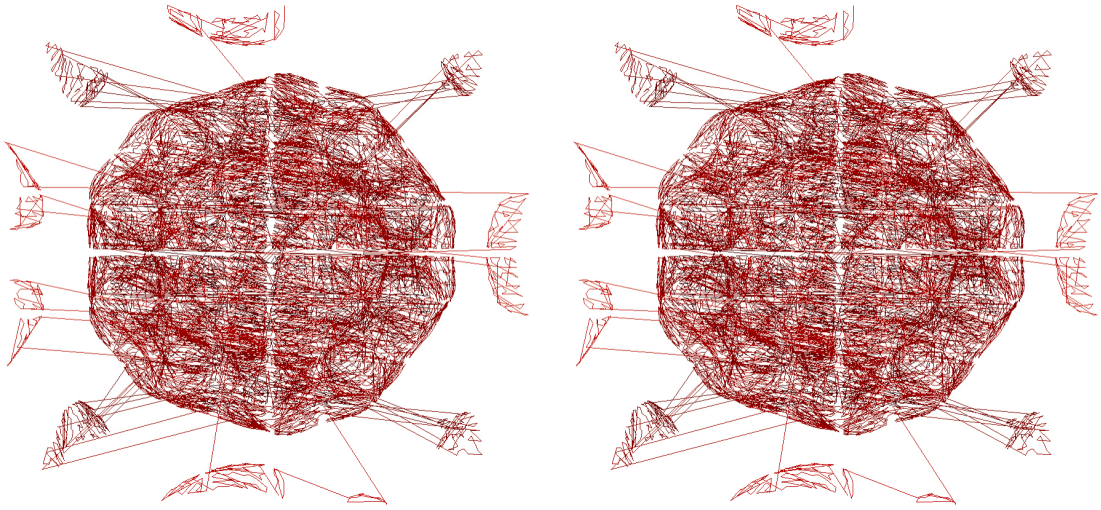
Model name	RT Baseline	RT global indexing	RT Cube 4x4x4	RT Local Indexing
Bucky	0.02682 ms	0.02737 ms	0.02463 ms	0.02460 ms
Neghip	0.02097 ms	0.02039 ms	0.01860 ms	0.01948 ms
Engine	0.14002 ms	0.07381 ms	0.06721 ms	0.12430 ms
Vertebra	0.07024 ms	0.03749 ms	0.03683 ms	0.08022 ms

Table 5.8: Measured rendering time in ms.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

Version	classify voxel	index scan	voxel scan	vertex scan	compact voxel	generate triangles
Bucky:						
Baseline	25 μs	0 μs	419 μs	198 μs	3 μs	4 μs
Local Index	30 μs	0 μs	399 μs	202 μs	4 μs	5 μs
Neghip:						
Baseline	29 μs	0 μs	402 μs	207 μs	4 μs	4 μs
Local Index	31 μs	0 μs	401 μs	214 μs	4 μs	5 μs
Engine:						
Baseline	26 μs	0 μs	1,422 μs	676 μs	6 μs	7 μs
Local Index	24 μs	0 μs	1,410 μs	668 μs	6 μs	7 μs
Vertebra:						
Baseline	10 μs	0 μs	5,694 μs	3,267 μs	6 μs	5 μs
Local Index	11 μs	0 μs	5,799 μs	3,271 μs	6 μs	6 μs

Table 5.9: Measured isosurface extraction processing times for the local voxel indexing scheme.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

As can be seen in **Table 5.7**, the local indexing scheme is similarly cache-efficient as the global indexing scheme. However, it is not as cache-efficient as the global indexing with cubes of size 4x4x4. The main reason for this is that the index selection scheme is not perfect and vertices might be duplicated. We can see in **Figure 5.11** that the triangle order for the global indexing with cubes and the local voxel indexing with cubes is identical. With a perfect index selection scheme the number of cache misses for the local voxel indexing might be reduced to the number of cache misses for the global indexing with cubes. Although the number of cache misses was reduced for all models compared to the baseline version the rendering times did improve only for 3 of the 4 volumes. The local voxel indexing reduced the number of cache misses for the Vertebra volume from 1.775.874 to 789.242. However, the rendering time shown in **Table 5.8** was increased from 0,07024 ms to 0,08022 ms by the local voxel indexing. A possible reason could be the fact that not all elements in the vertex buffer are used. This could mean that additional overhead for the vertex access is needed which could explain the increased rendering time even though the number of cache misses was reduced. As can be seen in **Table 5.9** the local indexing scheme did not produce significant overhead in the processing stages of the extraction process.



(a) global indexing with cubes of size 4x4x4. **(b)** local indexing with cubes of size 4x4x4.

Figure 5.11: Triangle order visualized using lines for the isosurface extraction of the Bucky volume with iv = 0.2f. Here the center points of two consecutive triangles in the index buffer are connected by a line. As can be seen, the triangle order in **(a)** and **(b)** is identical.

5.3 Evaluation

Using the global and local indexing scheme the number of cache misses was reduced for all tested volumes. The offline cache optimization algorithm Tipsify was able to reduce the number of cache misses further and shows that the cache optimization done by the global and voxel indexing might be improved upon. However, the global indexing with cubes of size 4x4x4 was able to get relatively close to the number of cache misses of the offline optimized mesh as can be seen in **Table 5.10**.

Model name	CM Baseline	CM Cubes 4x4x4	CM Tipsify
Bucky	43,524	14,795	12,315
Neghip	92,094	30,776	25,263
Engine	1,866,504	632,421	564,613
Vertebra	1,775,874	617,649	473,832

Table 5.10: Measured cache misses for the global indexing scheme with cubes of size 4x4x4 compared to the number of cache misses for the Baseline CUDA implementation and the Tipsify cache optimized mesh.
Measured on: Intel Core i7-7700k, 16 GB RAM, NVIDIA Quadro RTX 6000 24 GB

The rendering time for the Engine and Vertebra volume was approximately halved by the global indexing scheme with cubes of size 4x4x4, compared to the rendering time of the baseline implementation as can be seen in **Table 5.6**.

However, the global indexing scheme does not scale well with the volume size since the prefix sum scan quickly becomes expensive. In a server-client environment where a server extracts the isosurface and a hardware-weak client renders the extracted isosurface, such an approach might be viable. If weaker hardware is used to render the extracted isosurface, vertex post-transform cache efficiency is very important to achieve high rendering framerates or respectively low rendering times. For the engine volume the rendering time was nearly halved, compared to the rendering time of the baseline implementation which would double the framerate if a client would only render and display, but not extract the isosurface.

If the client extracts and renders the model, the global indexing scheme might become inefficient for larger volumes, since the framerate will be reduced by the computational cost

of the currently prohibitively expensive prefix sum scan. In such cases the local indexing scheme might be used. The extraction time of the local indexing scheme does scale well with the volume size. Furthermore, we were able to reduce the rendering time for Bucky, Neghip and the Engine model using the local indexing scheme.

6 Conclusion

In the course of this thesis, we showed that it is possible to extract cache-friendly isosurfaces in real-time on the GPU with little overhead. Furthermore, we evaluated efficient mesh representations in general, and proposed schemes based on recently gained insights about the vertex caching in modern GPUs [19], that are able to improve upon existing efficient mesh representation schemes [12, 31, 37]. The proposed mesh representation schemes can be used on arbitrary triangulated polygonal meshes and can be easily integrated into existing graphic pipelines.

Efficient Mesh Representation The mesh representation scheme that produced the best results in our tests applied cache optimization on the triangulated polygonal mesh first, split the mesh into parts of 2^{16} vertices based on the cache optimized index buffer, used unsigned short integers for the indexing of each part and reordered the vertices of each part to allow for better vertex fetching. Using this scheme the rendering time was nearly halved for models with more than 300,000 triangles (see **Table 4.31**) compared to the rendering time of the unoptimized triangulated polygonal mesh. The scheme can be easily integrated into existing graphics pipelines and can be used as the offline optimisation scheme of choice due to its simplicity and efficiency.

Our tests verified the findings of Vaněček and Kolingerová [17] that the use of cache optimization schemes can allow more efficient rendering than the use of triangle strips. We verified the identified warp behaviour of NVIDIA GPUs from Kerbl et al. [19] and found out that the number of cache misses for NVIDIA GPUs increases nearly three times faster than expected if the triangulated polygonal mesh, which is rendered using a single draw call, consists out of more than 2^{16} vertices (see **Figure 4.4**). Furthermore, we found out that the order of vertices in the vertex buffer has a strong influence on the number of

cache misses for NVIDIA GPUs if more than 2^{16} vertices are used in a single draw call (see **Table 4.34** and **4.37**).

Cache-friendly Isosurface Extraction Our proposed modifications to the marching cubes algorithm were able to reduce the rendering time for all tested volumes by up to half of the rendering time of the extracted isosurface from the baseline implementation. The proposed global indexing scheme with cube-wise processing using cubes of size 4x4x4 was the most efficient scheme in terms of the rendering time and number of cache misses of the extracted isosurface (see **Table 5.6** and **5.10**). The proposed local indexing scheme was able to reduce the number of cache misses for all volumes without the use and overhead of an additional prefix sum scan (see **Table 5.7** and **5.9**). Although the local indexing scheme was only able to reduce the rendering time for three of the four extracted isosurfaces, it can potentially be as efficient as the global indexing scheme with cube-wise processing and is a viable option for the extraction of cache-friendly isosurfaces with little or even no additional overhead.

However, the proposed real-time isosurface extraction schemes are not without limitations. Although both schemes were able to reduce the number of cache misses, it was not possible to beat an efficient offline cache optimization scheme with respect to the number of cache misses (see **Table 5.10**). The isosurface extraction process of the global indexing scheme turned out to be inefficient for larger volumes due to the high computational cost of the additional prefix sum scan. Furthermore, the CUDA implementation was unable to handle volumes with more than 2^{27} voxels and is limited to volumes and cubes for the cube-wise processing whose dimensions are a power of two. Due to time constraints, the gained insight about the advantage of splitting triangulated polygonal meshes into parts of 2^{16} vertices was not yet included into the isosurface extraction process. Splitting the isosurface into parts of 2^{16} vertices can potentially improve the rendering time of the extracted isosurface further.

All in all, we prove empirically with the work in the thesis that the real-time extraction of cache-friendly isosurfaces, which can be rendered efficiently, is possible.

Future Work

Although **Chapter 4** evaluates a lot of different mesh representation schemes, further mesh representation schemes, that were not evaluated in this thesis due to time-constraints, could be evaluated. It might be promising to evaluate the use of multi-draw rendering which can avoid unnecessary buffer bindings[1]. This could be beneficial for the model splitting scheme since the splitting can produce many parts as can be seen in **Table 4.22**. Furthermore, it makes sense to evaluate the mesh representation schemes on AMD and Intel GPUs in order to be able to give general advice for efficient mesh representations.

The rendering performance of the schemes proposed in **Chapter 5** for cache-friendly isosurface extraction can be improved using the insights gained about the caching behaviour of modern GPUs. Splitting the isosurface into parts of 2^{16} vertices might improve the rendering performance further. The volume splitting could be done using information from the voxel occupancy array of the reconstruction pipeline. Furthermore, improving the index selection algorithm of the local indexing scheme could increase the cache efficiency of the local indexing scheme. For the global indexing scheme, different prefix sum scans might be combined in order to reduce the computational cost of the reconstruction pipeline. To allow the usage of volumes larger than 512x512x512 an alternative to the necessary prefix sum scan function used in the NVIDIA CUDA example must be used, because the prefix sum scan was not able to process more than 2^{27} elements. Lui et al. [28] proposed an alternative approach for the isosurface extraction with MC using CUDA that is able to process larger volumes with up to 1024x1024x1546 voxels. For this Liu et al. [28] reduced the size of the input array for the prefix sum scan by using voxel blocks instead of individual voxels as elements of the array. It might be possible to integrate a similar approach into our proposed schemes making it possible to work with larger volumes as well. To improve the performance of the isosurface extraction further, a priori knowledge about the voxel occupancy e.g. from the brick occupancy structure of Jakob Wagner [48] can be used in the reconstruction pipeline in order to improve the isosurface extraction schemes.

1 `https://www.khronos.org/opengl/wiki/Vertex_Rendering#Multi-Draw` (Accessed on 11/10/2020)

Acknowledgements

I would like to thank the Computer Vision in Engineering Group at the Bauhaus-University Weimar for making the Obelisk model available for the experiments in this thesis. I would also like to show my deep appreciation to my supervisors Adrian Kreskowski and Gareth Rendle who helped me finalize this thesis.

Bibliography

[1] Bernhard Preim and Dirk Bartz. Chapter 03 - acquisition of medical image data. In Bernhard Preim and Dirk Bartz, editors, *Visualization in Medicine*, The Morgan Kaufmann Series in Computer Graphics, pages 35 – 64. Morgan Kaufmann, Burlington, 2007.

[2] Bernd. Fröhlich, Stephen Barrass, Björn Zehner, John Plate, and Martin Göbel. Exploring geo-scientific data in virtual environments. pages 169 – 173, 11 1999.

[3] Brian Curless and Marc Levoy. A volumetric method for building complex models from range images. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, page 303–312, New York, NY, USA, 1996. Association for Computing Machinery.

[4] Dimitrios S. Alexiadis, Dimitrios Zarpalas, and Petros Daras. Real-time, realistic full-body 3d reconstruction and texture mapping from multiple kinects. In *IVMSP 2013*, pages 1–4, 2013.

[5] Hubert Nguyen. *GPU Gems 3*, chapter 30, pages 633 – 677. Addison-Wesley Professional, 2007.

[6] Nvidia turing gpu architecture. `https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf`. (Accessed on 10/15/2020).

[7] Michael Deering. Geometry compression. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 13–20, 1995.

[8] Li Jiankun and C.-C. Jay Kuo. Dual graph approach to 3d triangular mesh compression. volume 2, pages 891 – 894 vol.2, 11 1998.

[9] Jarek Rossignac. Edgebreaker: Compressing the incidence graph of triangle meshes. 07 1998.

[10] Hugues Hoppe. Optimization of mesh locality for transparent vertex caching. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 269–276, 1999.

[11] Tom Forsyth. Linear-speed vertex cache optimisation. `https://tomforsyth1000.github.io/papers/fast_vert_cache_opt.html`, September 2008. (Accessed on 09/17/2020).

[12] Francine Evans, Steven Skiena, and Amitabh Varshney. Efficiently generating triangle strips for fast rendering. 1996.

[13] A. James Stewart. Tunneling for triangle strips in continuous level–of–detail meshes. 07 2001.

[14] Oliver van Kaick, Murilo da Silva, and Helio Pedrini. Efficient generation of triangle strips from triangulated meshes. pages 475–482, 01 2004.

[15] Jihad El-Sana, Francine Evans, Aravind Kalaiah, Amitabh Varshney, Steven Skiena, and Elvir Azanli. Efficiently computing and updating triangle strips for real-time rendering. *Computer-Aided Design*, 32(13):753 – 772, 2000.

[16] Pablo Diaz-Gutierrez, Anusheel Bhushan, Meenakshisundaram Gopi, and Renato Pajarola. Single-strips for fast interactive rendering. *The Visual Computer*, 22:372–386, 06 2006.

[17] Petr Vaněček and Ivana Kolingerová. Comparison of triangle strips algorithms. *Computers Graphics*, 31(1):100 – 118, 2007.

[18] Song Ho Ahn. Opengl display list. `http://www.songho.ca/opengl/gl_displaylist.html#:~:text=Display%20list%20is%20one%20of,perform%20the%20actual%20data%20transfer.`, 2005. (Accessed on 09/22/2020).

[19] Bernhard Kerbl, Michael Kenzel, Elena Ivanchenko, Dieter Schmalstieg, and Markus Steinberger. Revisiting the vertex cache: Understanding and optimizing vertex processing on the modern gpu. *Proc. ACM Comput. Graph. Interact. Tech.*, 1(2), August 2018.

[20] William Lorensen and Harvey Cline. Marching cubes: A high resolution 3d surface construction algorithm. *ACM SIGGRAPH Computer Graphics*, 21:163–, 08 1987.

[21] Jane Wilhelms and Allen Van Gelder. Octrees for faster isosurface generation. *ACM Trans. Graph.*, 11(3):201–227, July 1992.

[22] Michael Giles and Robert Haimes. Advanced interactive visualization for cfd. *Computing Systems in Engineering*, 1(1):51 – 62, 1990.

[23] Yarden Livnat, Han-Wei Shen, and Christopher R. Johnson. A near optimal isosurface extraction algorithm using the span space. *IEEE Trans. Vis. Comput. Graph.*, 2:73–84, 1996.

[24] Jihad El-Sana. Out-of-core algorithms for scientific visualization and computer graphics. 06 2003.

[25] Jagannathan Lakshmipathy, Wieslaw Nowinski, and Eric Wernert. A novel approach to extract triangle strips for iso-surfaces in volumes. pages 239–245, 01 2004.

[26] Manuel Scholz, Jan Bender, and Carsten Dachsbacher. Real-time isosurface extraction with view-dependent level of detail and applications. *Computer Graphics Forum*, 34, 09 2014.

[27] Junjie Chen, Xiaogang Jin, and Zhigang Deng. Gpu-based polygonization and optimization for implicit surfaces. *Vis. Comput.*, 31(2):119–130, 2015.

[28] Baoquan Liu, Gordon Clapworthy, Feng Dong, and Enhua Wu. Parallel marching blocks: A practical isosurfacing algorithm for large data on many-core architectures. *Computer Graphics Forum*, 35:211–220, 06 2016.

[29] Pablo Diaz-Gutierrez, Meenakshisundaram Gopi, and Renato Pajarola. Hierarchyless simplification, stripification and compression of triangulated two-manifolds. *Comput. Graph. Forum*, 24:457–467, 09 2005.

[30] Martin Isenburg. Triangle strip compression. *Computer Graphics Forum*, 20, 2001.

[31] Francine Evans, Steven Skiena, and Amitabh Varshney. Optimizing triangle strips for fast rendering. *Proceedings of the IEEE Visualization Conference*, 08 2001.

[32] Dave Shreiner, Graham Sellers, John M. Kessenich, and Bill M. Licea-Kane. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*. Addison-Wesley Professional, 8th edition, 2013.

[33] Regina Estkowski, Joseph Mitchell, and Xinyu Xiang. Optimal decomposition of polygonal models into triangle strips. *Proceedings of the Annual Symposium on Computational Geometry*, 05 2002.

[34] Jirí Síma and Radim Lněnicka. Sequential triangle strip generator based on hopfield networks. *Neural computation*, 21:583–617, 03 2009.

[35] Arseny Kapoulkine. Mesh optimization library that makes meshes smaller and faster to render. `https://github.com/zeux/meshoptimizer`. (Accessed on 09/17/2020).

[36] Mirela Ben-Chen and Andy Lai Lin. Microsoft powerpoint - 02_basics_2.pptm. `http://graphics.stanford.edu/courses/cs468-10-fall/LectureSlides/02_Basics.pdf`, 2010/2011. (Accessed on 09/22/2020).

[37] Pedro Sander, Diego Nehab, and Joshua Barczak. Fast triangle reordering for vertex locality and reduced overdraw. *ACM Trans. Graph.*, 26:89, 07 2007.

[38] Klaus Engel, Markus Hadwiger, Joe Kniss, Christof Rezk-Salama, and Daniel Weiskopf. *Real-Time Volume Graphics*. Ak Peters Series. Taylor & Francis, 2006.

[39] Nvidia/cuda-samples: Samples for cuda developers which demonstrates features in cuda toolkit. `https://github.com/NVIDIA/cuda-samples`. (Accessed on 09/17/2020).

[40] Christopher Lux. The opengl timer query. In Patrick Cozzi and Christophe Riccio, editors, *OpenGL Insights*, pages 493–502. CRC Press, 2012.

[41] Erik Smistad. smistad/gpu-marching-cubes: A gpu implementation of the marching cubes algorithm for extracting surfaces from volumes using opencl and opengl. `https://github.com/smistad/GPU-Marching-Cubes`. (Accessed on 09/21/2020).

[42] Cyril Crassin. Opengl geometry shader marching cubes. `http://www.icare3d.org/codes-and-projects/codes/opengl_geometry_shader_marching_cubes.html`, January 2007. (Accessed on 09/21/2020).

[43] Paul Bourke. Polygonising a scalar field (marching cubes). `http://paulbourke.net/geometry/polygonise/`, May 1994. (Accessed on 09/21/2020).

[44] Jason Sanders and Edward Kandrot. *Cuda by Example: An Introduction to General-Purpose GPU Programming*, chapter 4.2, pages 38 – 57. 01 2011.

[45] Arseny Kapoulkine. Using the modulo (%) operator in cuda 65536 | beechwood.eu. `https://www.beechwood.eu/using-the-modulo-operator-in-cuda-65536/`, November 2014. (Accessed on 09/16/2020).

[46] Thrust. `https://developer.nvidia.com/thrust`. (Accessed on 09/17/2020).

[47] Nicholas Wilt. 10.13 texturing quick references | the cuda handbook: Texturing | informit. `https://www.informit.com/articles/article.aspx?p=1949761&seqNum=10`, April 2013. (Accessed on 09/16/2020).

[48] Jakob Wagner. Real-time 3d reconstruction from multiple rgbd sensor streams, 2016.

[49] Patric Ljung. Efficient methods for direct volume rendering of large data sets. 2006.

[50] Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter-Pike Sloan. Interactive ray tracing for isosurface rendering. pages 233–238, 11 1998.

[51] Manuel Scholz, Jan Bender, and Carsten Dachsbacher. Real-time isosurface extraction with view-dependent level of detail and applications. *Computer Graphics Forum*, 34, 09 2014.