FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Evaluierung von Coherent Hierarchical Culling Revisited mit variierten Parametern

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Ing. Jürgen Thann
Matrikelnummer 01025543

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Mag. Stefan Ohrhallinger, PhD

Wien, 30. Juni 2019

Jürgen Thann Michael Wimmer

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Evaluation of Coherent Hierarchical Culling Revisited with Varied Parameters

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Ing. Jürgen Thann
Registration Number 01025543

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Mag. Stefan Ohrhallinger, PhD

Vienna, 30th June, 2019

Jürgen Thann	Michael Wimmer

Erklärung zur Verfassung der Arbeit

Ing. Jürgen Thann

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 30. Juni 2019

Jürgen Thann

Danksagung

Hiermit möchte ich mich bei Mag. Stefan Ohrhallinger, PhD bedanken, welcher mich in das Thema des wissenschaftlichen Arbeitens eingeführt hat und mir bei dieser Arbeit mit Rat, Tat und Wegweisung immer zur Verfügung stand. Vielen Dank auch für die zur Verfügung gestellten Punktwolken. Diese haben in meiner Arbeit für eine breitere Perspektive gesorgt und eine bessere Bewertung der Ergebnisse meiner Tests ermöglicht.

Ebenfalls bedanken möchte ich mich bei Univ.Ass. Dipl.-Ing. Markus Schütz MSc für die zur Verfügung gestellte Punktwolke „Project Endeavor“ und die Entwicklung und Unterstützung bei der Verwendung von Potree. Diese Anwendung hat mich aus einer schwierigen Situation gerettet und die weitere Entwicklung meiner Implementierung deutlich vereinfacht.

Ich danke meiner Mutter dafür, dass sie mir dieses Studium sowohl durch ihre Erziehung, als auch durch ihre Hingabe und Unterstützung ermöglicht hat. Vor allem aber möchte ich mich bei meiner Frau für ihre immerwährende Geduld, Unterstützung und Motivation bedanken. Ohne sie wäre diese Arbeit wohl nie vollendet worden.

Acknowledgements

Hereby, I want to thank Mag. Stefan Ohrhallinger, PhD for introducing me to the topic of scientific working and for always supporting me with advice, action, and guidance. I also thank him very much for the provided point clouds. These point clouds supplied my thesis with a wider perspective and allowed me to better assess the results of my tests.

I also want to thank Univ.Ass. Dipl.-Ing. Markus Schütz MSc for the supplied point cloud "Project Endeavor" and the development of Potree, as well as his support in using it. This application helped me out of a difficult situation and eased my way in the further development of my implementation.

I thank my mother for allowing me these studies through her upbringing as well as her commitment and support. I especially want to thank my wife for her everlasting patience, support, and motivation. Without her, this thesis probably would have never been completed.

Kurzfassung

Die Größe von Punktwolken, welche aus verschiedenen Arten von 3D-Scan-Techniken gewonnen werden, steigt stetig. Um mit dieser Entwicklung mithalten zu können, muss die Geschwindigkeit des Renderings von Punktwolken entsprechend verbessert werden. Zusätzlich zu View Frustum Culling kann auch Occlusion Culling verwendet werden, um die Anzahl der zu ladenden Punkte zu reduzieren. Der Occlusion Culling Algorithmus, welchen ich evaluiere, nennt sich „CHC++: Coherent Hierarchical Culling Revisited“ [MBW08]. Für diese Arbeit habe ich einen parametrisierbaren Punktwolken-Renderer implementiert, welcher mir den Test mehrerer Parameter mit variierten Werten erlaubte. Obwohl nicht alle Parameter in der evaluierten Abhandlung definiert wurden, nimmt doch jeder Einfluss auf die Leistung des Algorithmus. Meine Evaluierung zeigt, dass der Algorithmus die Leistung der Darstellung nicht-synthetischer Szenen deutlich erhöht. Allerdings führt die Verwendung von Tighter Bounding Volumes zu einer geringen Verschlechterung der Leistung, anstatt diese zu erhöhen. Weiters führt die Verwendung einer SSD-Festplatte anstelle einer HDD-Festplatte nicht zu dem erwarteten Einfluss auf die Geschwindigkeit des Ladevorgangs. Abschließend stelle ich eine anwendungsfallbasierte Entscheidungsrichtlinie für die Wahl der Werte der Parameter zur Verfügung. Meine Implementierung, welche ich für diese Arbeit erstellt habe, steht unter der GNU Lesser General Public License, Version 3 zur Verfügung.

Abstract

Point clouds received through various types of 3D-scanning techniques increase in size constantly. To compensate for this fact, the performance of rendering point clouds has to be improved accordingly. In addition to view frustum culling, occlusion culling can be used to reduce the number of points that has to be loaded. The occlusion culling algorithm I evaluate is called "CHC++: Coherent Hierarchical Culling Revisited" [MBW08]. For this thesis, I implemented a parameterizable point cloud renderer that allowed me to test varied values for multiple parameters. While not all parameters are defined in the evaluated paper, all influence the algorithm's performance. My evaluation shows that the algorithm increases performance clearly for non-synthetic scenes. However, with the scenes in this evaluation, Tighter Bounding Volumes introduced a slight decrease in performance instead of improving it. Furthermore, using an SSD instead of an HDD did not yield the expected impact on the loading speed. Finally, I provide a general use case specific decision guideline for choosing values for the parameters. My implementation produced for this thesis is available under the GNU Lesser General Public License, Version 3.

Contents

Kurzfassung **xi**

Abstract **xiii**

Contents **xv**

1 Introduction **1**

2 Theory **3**
2.1 Octree 3
2.2 Potree 3
2.3 Occlusion Culling 4
2.4 Coherent Hierarchical Culling Revisited 6

3 Method **9**
3.1 Implementation 9
3.2 Point clouds 14
3.3 Metrics 15

4 Evaluation **17**
4.1 VFC vs. CHC++ 18
4.2 HDD vs. SSD 21
4.3 Node size 23
4.4 Occlusion check frequency for visible nodes 27
4.5 Tighter Bounding Volume depth 30

5 Conclusion **33**
5.1 Future work 34

List of Figures **35**

List of Tables **37**

List of Algorithms **39**

CHAPTER 1

Introduction

Creating 3D-scans of buildings and areas has become a widely used method to analyze and document antique sites, famous buildings or the progress on building sites. To create such scans there are multiple techniques available. The most frequently used techniques are using a laser scanner or extracting points from videos or pictures. While the first method shoots laser beams at a high frequency and captures their refraction/reflection, the second extracts points from multiple images by triangulating point positions based on the distance between two viewpoints in comparison to the distance the point has moved. The captured points are collected in a list, where their coordinates, as well as other attributes like the orientation and color of the surface, are stored. As digital storage becomes faster and cheaper, the detail provided by such scans increases as well.

As the size of point clouds increases, the performance of rendering software has to keep up. To manage this, observations from real-life were integrated into algorithms which prematurely exclude points that would be hidden from view anyway. To understand such algorithms we first have to consider the method used when rendering point clouds.

Since the captured points have no size per se, to render point clouds to the screen a technique called "splatting" is used. You can imagine splatting as if shooting the points in the point cloud with paintballs of the corresponding color, resulting in round color-splats centered on the given coordinates. The bigger the distance between the scanner and the scanned surface is, the bigger the resulting distance between the captured points will be. Because of this, the splat size has to be chosen large enough to avoid gaps in the surface, but small enough to keep edges of surfaces intact and visually appealing.

To reduce the number of points rendered we can now apply two steps. First, since we have a limited field-of-view, we can easily exclude points outside of it. This technique is called "view frustum culling". Secondly, when we render points as previously mentioned, we can go ahead and remove all points that are completely hidden behind other points.

This technique is called "occlusion culling" and is not as easily accomplished as view frustum culling.

Many algorithms have been implemented to tackle this task. The algorithm this evaluation focuses on is called "CHC++: Coherent Hierarchical Culling Revisited" [MBW08], which further improves an already sophisticated algorithm called Coherent Hierarchical Culling (CHC) [BWPP04]. This algorithm manages to efficiently detect and exclude occluded objects from the loading process. Thus, this technique reduces the RAM required and improves render performance.

CHC++ has some flexibility built into it in the form of parameters. Although Mattausch et al. recommended values for these parameters based on their used scenes, they also mentioned that the exact values for these parameters remain scene-dependent. The goal of this thesis is to evaluate the effect varied values for certain parameters have on the performance of rendering scenes with a vastly different amount of occlusions. Furthermore, I strive to provide a general guideline for a more goal-oriented parametrization.

CHAPTER 2

Theory

In this chapter I explain the concepts used in this bachelor's thesis, including the data structure, the basics of occlusion culling as well as the main subject of this thesis — Coherent Hierarchical Culling Revisited [MBW08].

2.1 Octree

An octree is a hierarchical data structure that stores its contained points based on their 3D position. A node can be either an inner-node that redirects points into their appropriate container or a leaf-node that stores the aforementioned points. Inner-nodes represent cubes whose child-nodes split each dimension (x, y, z) in half, thus resulting in 8 (2^3) child-nodes. After reaching a certain predefined depth a leaf-node will be created that stores all points redirected to it.

2.2 Potree

Potree [Sch16] is a point cloud renderer, specifically designed to handle large point clouds and render them to a web browser. In this thesis, I am specifically interested in the data structure that comes with it.

Schütz's Potree data structure is based on Scheiblauer's "modifiable nested octree" (MNO) [Sch14]. An MNO supports rendering a point cloud in different levels of detail by storing some points inside the inner-nodes. The original MNO described by Scheiblauer splits each inner-node into a 128x128x128 grid (128^3 sub-cubes). When filling the data structure each of these sub-cubes stores a single point to create a low-resolution representation for the current level and stores any additional points that fall into it in a back buffer. If we split the back buffer in half in every dimension we receive 8 sub-buffers (64^3 sub-cubes) that represent 8 possible new child-nodes. When a sub-buffer

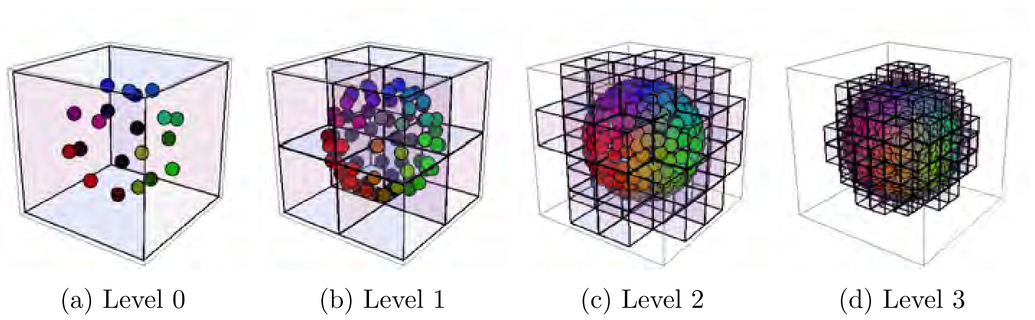

Figure 2.1: Example of multiple levels of the "modifiable nested octree" (MNO) [Sch14] and Potree data structure [Sch16] respectively. Image by [Sch16].

has collected points at least equal to a certain threshold the points in the back buffer are used to create new child-nodes. Otherwise, the node is considered a leaf-node. Points that are used for representation are only stored in the described grid and in no other node, thus the MNO stays free of unnecessary redundancy.

Building an octree in that way allows for loading points level by level without loading the whole tree, increasing its detail with every step. This is especially useful for nodes that are far from the viewer since traversal can be stopped as soon as a deeper level only adds details in the sub-pixel space.

The main difference between Scheiblauer's MNO and Schütz's Potree data structure is the sub-sampling technique used. The former does not guarantee a minimum distance between points. So 8 points that are very close to each other could theoretically fall into 8 different sub-cubes and thereby reduce the quality of the low-detail representation by clustering points. The latter implements "Poisson-disk sub-sampling", which guarantees a minimum distance by calculating the distance between the already added neighboring points and the point to be added and compares it against a given threshold. Schütz suggests using the node size divided by 128 as the threshold to closely mimic the 128^3 grid used by Scheiblauer. In figure 2.1 you see an example of an MNO and Potree respectively.

2.3 Occlusion Culling

Since the scenes to be rendered become increasingly complex in modern use cases we have to develop new ways to increase performance constantly. For the most part, we manage this by reducing the number of objects rendered that have no influence on the end result. One early technique to exclude non-visible objects from rendering is called view frustum culling (VFC). In VFC, objects that are not in the view frustum (the visible field) are ignored. However, there are many other objects that are in the view frustum, but still have no influence on the end result, namely occluded objects — objects that are completely covered by other objects in front of them.

This is where occlusion culling comes into play. Occlusion culling detects objects that

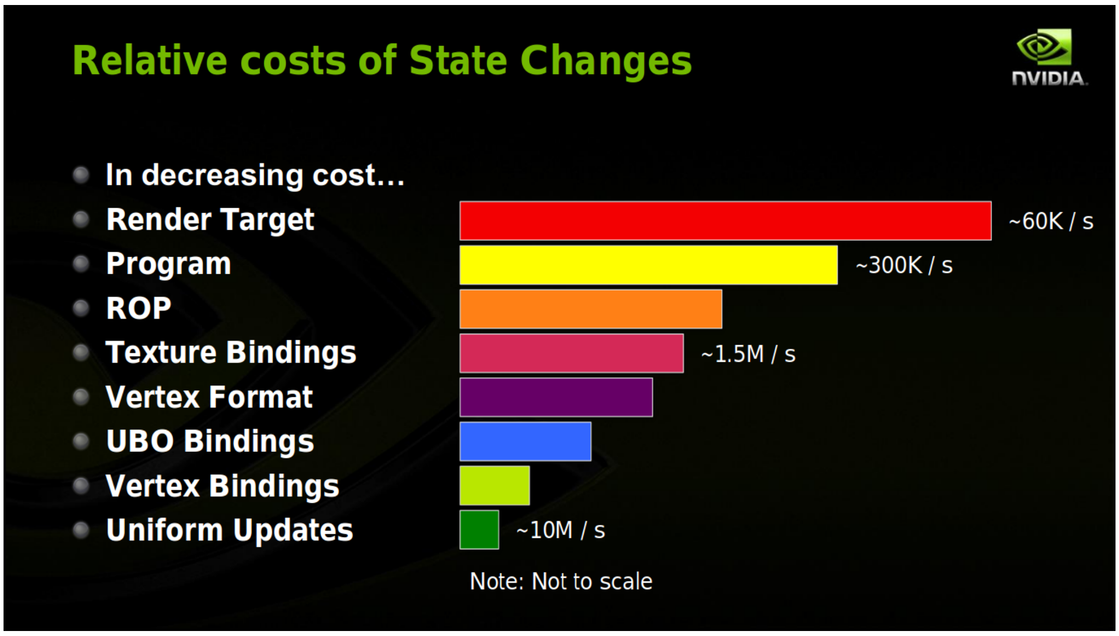

Figure 2.2: Types of state changes ordered by cost. CHC++ [MBW08] significantly reduces the number of state changes, which typically are changes of "Render Target" and "Program". Extracted from page 48 of [EM14].

have no impact on the image when they are rendered and prematurely excludes them. To improve the performance of this check, modern graphics-hardware implements occlusion queries. When using occlusion queries we first render all objects that are currently not occluded to initialize the z-buffer. Afterward, we change the hardware-state to switch from rendering to executing queries. In this state we render the object to test, but because of the query-state instead of rendering the object, the hardware counts the numbers of pixels that would be changed after the depth-test. If no pixel is altered the object is occluded. If we use a simpler bounding object, like an axis-aligned bounding box (AABB), we can further improve performance by using a less complex object for the occlusion query and possibly even avoid loading the whole object.

Depending on the implementation each occlusion query might require a change of the shader program. As we can see in figure 2.2, in 2014 graphics hardware was able to process approximately 300,000 program state changes per second. Even if today's hardware is considerably faster changing the program stays the second-most complex state change. If we neglect this fact, the overhead for changing the state on every node queried might skyrocket. Thus algorithms have to be developed to reduce state changes to clearly out-perform simply loading and rendering the object. The occlusion culling algorithm that this thesis is based upon is called "CHC++: Coherent Hierarchical Culling Revisited" [MBW08].

2.4 Coherent Hierarchical Culling Revisited

CHC++ by Mattausch et al. [MBW08] focuses on decreasing the number of state changes as well as the number of queries itself. Occlusion culling using CHC++ can be separated into the following steps:

- **Batch previously invisible nodes**
  While traversing the tree we collect all previously invisible nodes in the invisible-queue (i-queue). The visibility of previously invisible nodes has to be detected as fast as possible, since not rendering nodes that should be visible leaves gaps that significantly reduce visual quality. To avoid this, we check these previously invisible nodes every frame. To reduce expensive changes of the hardware state, occlusion queries are only executed after the number of nodes in the i-queue surpasses a certain threshold.

- **Batch previously visible nodes**
  While traversing the tree we also collect previously visible nodes in the visible-queue (v-queue). Since rendering a small number of occluded objects does not have a big influence on the performance, previously visible nodes are only checked again after a certain number of frames. That way the number of queries is further reduced. Also, to decrease CPU stalls, we can assume the node stays visible and simply render it immediately, only taking the result of the query into account on the next frame.

- **Execute multi-queries**
  We use the i-queue to create multi-queries. When multiple nodes are invisible for approximately the same number of frames, we assume that their visibility-state is strongly interdependent. This is called temporal coherence. Using this assumption a single multi-query can be executed to check the occlusion of those strongly interdependent nodes. This way the number of queries is significantly reduced. If one of the nodes checked by the multi-query becomes visible however, all the nodes contained in the multi-query have to be queried again independently.

- **Execute single-queries**
  After traversal and when all queries in the i-queue are processed we use the v-queue to keep the GPU busy. Since the results of occlusion queries are not immediately available we give the GPU time to finish the queries while continuing with the frame. When we return to traversing the tree next frame the results should already be available and we exclude all nodes that are now considered occluded from rendering.

One of the problems of using AABBs as bounding objects is the number of nodes falsely considered visible. This occurs because oftentimes only some of the octree's child-nodes contain data, but are still encased by the bounding volume. To tackle this problem Mattausch et al. implemented "tighter bounding volumes" - a technique where instead of using the AABB of the node to check, they iterate three levels into the node and combine the AABBs of all the child-nodes that contain points. This way the number of queries is even further reduced by possibly eliminating the occlusion queries necessary for all of the child nodes.

CHAPTER 3

Method

In this chapter, I present my implementation of CHC++ [MBW08] that is used for all evaluations in this thesis. Afterward, I present the test scenes used and which challenges each provides to CHC++. At last, I describe what comparisons I perform and which values are used to present the results.

3.1 Implementation

For this thesis, I implemented a point cloud renderer called "UPCR" (unlimited point cloud renderer). It is written in C++ using Qt, based on OpenGL 4.2, and does not require graphics hardware of any specific brand. UPCR has been developed and tested on Ubuntu 14.04. No guarantee can be given about its stability or operationality on other operating systems. To allow easy comparison between view frustum culling and CHC++ the rendering process only differs in the use of "tighter bounding volumes" (described in Section 2.4) and occlusion queries when using CHC++. As for the load order of nodes, the distance-based priority queue proposed in CHC [BWPP04], the groundwork for CHC++, is used in both modes (VFC and CHC++). See Figure 3.1 for a sample screenshot of UPCR.

The main features of UPCR are:

- loading point clouds using the Potree data structure
- rendering point clouds using square splats
- view frustum culling
- CHC++
- statistics collection and rendering

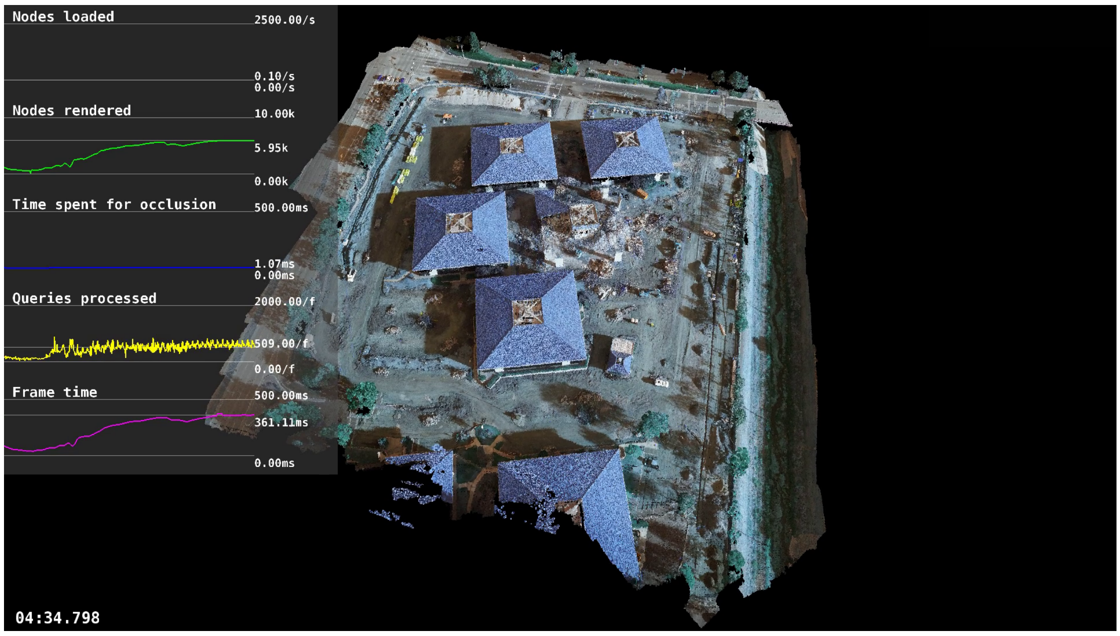

Figure 3.1: A screenshot of UPCR — my point cloud renderer implemented to collect the necessary data for the evaluations in this thesis. The loaded point cloud presents the Nvidia Endeavor Campus in Santa Clara, California during construction, see Section 3.2.2.

- recording flythrough paths
- playing flythroughs using Catmull-Rom splines
- taking a video of a flythrough using FFmpeg

The source code is published under LGPL license and can be found here. Accessing the source code requires an account from the Institute of Visual Computing & Human-Centered Technology at TU Wien. Please contact ohrhallinger@cg.tuwien.ac.at if you are interested in gaining access.

3.1.1 Key-Bindings

3.1.2 Parametrization

UPCR can be parametrized in many ways to compare different scenarios without recompiling and is thus also usable for test automation using scripts. The supported parameters are:

- `--input <path> (-i <path>)`
  Defines the path of the Potree point cloud to load

Key(s)	Function
W/S	moves the camera in the (opposite) view direction
A/D	moves the camera left/right in relation to the view direction
Space/C	moves the camera up/down along the vertical axis
Arrow keys	turns the camera
R	toggles recording of a flythrough
O	toggles between occlusion culling (CHC++) and view frustum culling

Table 3.1: Key-bindings used in UPCR. Additionally, the camera can be turned by holding the left mouse button down and moving the mouse.

- `--diskSpeed <MB/sec>`
  With this parameter, the disk speed can be throttled. This is useful to simulate the performance of an average HDD while using an SSD, for example. This parameter can also be used to limit the impact data stored in RAM has on the results.

- `--diskSeekTime <ms>`
  This parameter defines the time a drive requires to address requested data. This number is especially useful for simulating an HDD while using an SSD since the mechanical process of reading data introduces a far greater seek time than the electronic process of an SDD.

- `--nooc`
  (No) (O)cclusion (C)ulling — disables CHC++ and only uses view frustum culling

- `--visibilityCheckFrequency <num> (-v <num>)`
  Usable in combination with CHC++. Defines the number of frames to wait before re-checking the visibility of a node that was previously checked and found to be visible.

- `--maxTightBVDepth <depth>`
  Defines the maximum depth to use to create "tighter bounding volumes" (described in Section 2.4). The default value is 3.

- `--flythrough <path> (-f <path>)`
  The flythrough path to load and play. Recorded flythroughs are located in the "flythrough" directory in the working directory.

- `--waitForLoadComplete (-w)`
  When the flythrough parameter is used in combination with this switch the flythrough will pause until all nodes in the view frustum are loaded. Afterward, flythrough playback continues until new non-loaded nodes are discovered.

- `--roc`
  (R)ender (O)cclusion (C)ulling — renders the bounding boxes that are used for occlusion queries.

- `--nosr`
  (No) (S)plat (R)endering — disables the rendering of splats. Mostly useful for examining occlusion queries (–roc).

- `--recordFileName <name> (-r <name>)`
  This parameter has multiple effects. It defines that this session shall be recorded, sets the name of the resulting video and also sets the name of the generated statistics file (instead of using a timestamp). The video can be found in the "videos" directory under the current working directory.

- `--collectStatistics (-s)`
  This switch defines whether or not statistics shall be collected. With this switch, the collected statistics will be written to a comma-separated values (CSV) file in the "statistics" directory in the current working directory.

- `--psStats`
  (P)er (S)econd (stats) - this switch defines that statistics shall not be collected per frame, but per second.

- `--collectUnrenderedNodesStats`
  Defines that the statistics collection shall contain statistics about unrendered nodes. Be aware that specifying this switch results in additional computational load, since additional occlusion queries have to be executed to collect the necessary data.

- `--statisticsToRender <CsvName>,<Label>,<MinValue>, <MaxValue>,<Divisor>,<Unit>`
  This parameter can be defined up to six times. It specifies which of the statistics written to the CSV statistics file shall also be rendered on-screen. The only required value is <CsvName> which defines the name of the statistics to render as found in the CSV file. When no min-/max-value is defined, they automatically adapt to the collected values. One scenario where using a divisor is useful is to convert statistics collected in megabytes to gigabytes. The divisor, in this case, would be 1024.

- `--statisticsPosition <Left/CenterLeft/CenterRight/Right>`
  Defines where the statistics shall be rendered.

3.1.3 Statistics rendering

Overall, UPCR collects 37 different values every frame/second. When comparing different configurations not all of these values will be of interest at the same time. Also, preparing the collected data for comparison for many different test scenarios can be a time-consuming task. To streamline this process I implemented statistics rendering into UPCR.

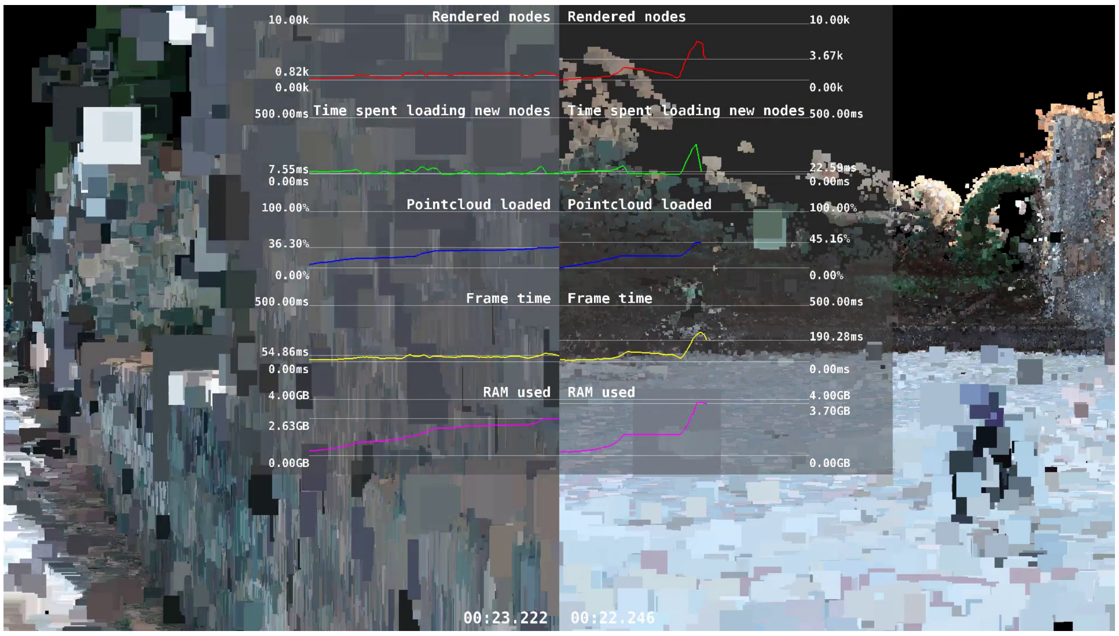

Figure 3.2: One possible way to compare data. For this sample I ran UPCR twice. On the left it runs with CHC++ [MBW08] and renders the statistics on the right. On the right it runs with VFC and renders the statistics on the left. I recorded both runs using the parameter `--recordFileName` and merged the resulting videos in a post-processing step using FFmpeg. The loaded point cloud is a scan of the Arènes de Lutèce, see section 3.2.3.

We can define up to six values which will be rendered to charts. I improved the readability of the charts by using a simple moving average (SMA), an unweighted average taken over a sliding window of a fixed size. The window size, in this case, is 10 frames or 1 second, depending on whether statistics are collected per frame or per second. To further improve usability these charts can be positioned left, right or to the left or right side of the center. Using the recording function of UPCR and the positioning of the charts right enables us to put two videos side-by-side in a post-processing step and achieve easy comparability using the horizontal line that indicates the current value in the chart. For this use case manually setting a minimum and maximum value is desirable to keep both sides of the video in relation. You find one sample result of this process in Figure 3.2.

Another possible use case is the comparison of run-time. UPCR uses a fixed, time-independent step size for playing flythroughs. When we check two configurations that yield a vastly different number of FPS a time-dependent step size would achieve the same run-time in both cases. Using a fixed step size on the other hand we can clearly see which configuration performs better on average. When we position recordings of such two executions side-by-side and keep the last frame of whichever video finishes earlier we can see a direct comparison of the run-time using the time-stamp rendered on-screen.

You can find example-usages for the described use cases in the shell-scripts here. For information on how to gain access see Section 3.1.

3.1.4 Flythrough

Flythroughs provide an easy way to repeat tests and use the same scenario for collecting different data in multiple executions. To collect the data for this thesis I used a single flythrough per test scenario, but replayed it for every comparison. Every replay rendered different statistics on-screen to allow easy side-by-side comparison.

To start recording a flythrough run UPCR using the desired point cloud and press R. To improve precision while recording camera speed is reduced to 25% of its original value. When recording a flythrough the position and view direction are recorded while moving the camera. Points are only recorded after the camera has moved a small distance. The reason behind this limitation is to limit the size of the resulting flythrough file and to create a more fluid flythrough. Keep in mind that creating flythroughs that pause in between is thus impossible with the current implementation.

When playing a flythrough, the previously collected positions are used to create a Catmull-Rom spline. A Catmull-Rom spline has the advantage that the resulting spline (or path) passes through all the points that build it, and no additional attributes are required. That way we can use the spline to interpolate a path that will make the flythrough pass exactly through all the points recorded. To further increase flexibility and quality of the flythrough the view direction is interpolated between points. This enables us to look sideways at an object while passing it and gives the flythrough a more natural look-and-feel.

3.2 Point clouds

Occlusion culling is specifically designed to increase the performance of rendering for scenes with a lot of occlusions. To evaluate CHC++ [MBW08], I used a set of point clouds that allow the algorithm to perform at its best, but also pose the biggest possible challenge to it.

3.2.1 Sphere - no occlusion - 31,401,475 points

This point cloud is hand-made by modeling it in Blender and converting the mesh to a point cloud using CloudCompare's point sampling function. It is made up of – as the title suggests – a sphere. The challenge this point cloud poses is that when the camera is moved into the sphere not a single node is occluded. This fact is used to examine whether CHC++ always performs better than simple view frustum culling.

3.2.2 Project Endeavor - few occlusions - 48,231,579 points

Created by Nvidia [Nvi16], provided by Markus Schütz

This point cloud is a drone scan of the building site of the Nvidia Endeavor Campus in Santa Clara, California. Since the scan was created from above by using a drone, many parts of the walls are missing. Also, the building site is relatively flat. Thus not a lot of objects are occluded, resulting in a good challenge to start seeing first improvements over simple view frustum culling.

3.2.3 Arènes de Lutèce - moderate occlusions - 60,534,942 points

Created by The Harvest4D Consortium [Con16], provided by Stefan Ohrhallinger

This point cloud is a ground-based scan of the Arènes de Lutèce in Paris, France. Its side passages offer great opportunities for occlusion while the open space in the center provides the possibility for challenging the CHC++ algorithm.

3.2.4 Office - lots of occlusions - 22,054,298 points

This is another hand-made point cloud using Blender. It consists of ten floors, each containing a hallway, nine rooms and a staircase to the other floors, resulting in 100 rooms. This scene represents the best-case scenario for CHC++.

3.3 Metrics

As mentioned in Chapter 3.1.3, UPCR captures 37 different values. Since not every value is significant for every comparison, I limit myself to a small set of the most relevant statistics. The most important attribute when comparing performance is how many milliseconds it takes to process a single frame — especially when comparing the benefits of using CHC++ [MBW08] over view frustum culling. A chart depicting this value is present in every comparison in Chapter 4, with one exception.

The only case where the milliseconds per frame have no importance is when comparing the performance of the loading process on an SDD vs. an HDD. In this case, the only value of real importance is how many points can be loaded per second. When comparing the performance of view frustum culling with the performance of CHC++, we can use the same value but expect a different outcome. We can use it to prove that occlusion culling reduces the number of points loaded and thus see a decline in the number of points loaded per second.

Another parameter that influences the speed at which a point cloud can be loaded originates from the data structure itself. When converting a point cloud to the Potree data structure we can define the node size — the maximum number of points in a single leaf-node. Increasing this parameter increases the size of a single node-file, but also reduces the overhead when accessing many files in a short period of time.

CHC++ also has some parameters which can be optimized depending on the rendered scene. One big influence on performance is the number of occlusion queries executed. The importance of this is reflected in CHC++ in the form of a parameter that allows us to specify how often visible nodes will be rechecked. To better understand the parameter's influence on the rendering process we compare the number of occlusion queries required per frame when altering this parameter to certain values.

One feature of CHC++ that – based on the scene – may have beneficial or counterproductive effects is called "tighter bounding volumes", which is described in Section 2.4. In short, it produces tighter bounding volumes by iterating a certain depth into the node and using its child node's bounding volumes. The depth at which the bounding volumes are collected can be altered using another parameter of CHC++. To analyze the cost-benefit ratio at different values we compare the time required for occlusion culling each frame.

CHAPTER 4

Evaluation

To evaluate CHC++ [MBW08] and other aspects of point cloud rendering, I created two types of flythroughs. When comparing the overall performance, I used flythroughs that first travel through the scene along a walking-style path, flying close to the ground and through corridors. To allow frame-by-frame comparison, I use a fixed step-size. Thus, independent of parameters, all the flythroughs for one scene end after exactly the same number of frames. Those dynamic flythroughs end with an overview of the complete scene by flying up high while the view is directed towards the scene. The only flythrough that does not end in this way is for scene "Sphere" (described in Section 3.2.1). This scene was specifically created to test CHC++ against a scene that has no occlusions. Showing the whole sphere from the outside would render the use of this scene void since the nodes on the backside of the sphere would be occluded.

There is another form of flythroughs I am using when comparing the performance of the initial loading process of a scene. This form of flythroughs remains static in a single point. Unlike dynamic flythroughs that end after the flythrough is finished, static views end after all nodes in the view frustum are loaded. This is accomplished by using the `--waitForLoadComplete` switch of UPCR (see Section 3.1.2). Thus, depending on the parameters, the same flythrough for the same scene may end sooner or later, based on the parameters used.

To more clearly present the results, I prepared the collected data by removing outliers. I remove outliers by sliding a window containing 50 values across the data to adjust, calculate the standard deviation and mean in this window and remove values 3 times the standard deviation above or below the mean. Afterward, the sliding window is moved by 12 values. Additionally, I apply a moving average with a window size of 1% of the overall frame count. These preprocessing steps reduce detail for the sake of readability of the charts.

In the following sections, I will describe the results of my evaluation using the aforementioned dynamic flythroughs and static views. First I will compare the performance when using view frustum culling as opposed to CHC++ while using dynamic flythroughs. In the next sections, I compare the performance of initializing a scene using an HDD versus an SSD and using varied node sizes for the Potree data structure [Sch16] with static views. Finally, in the last sections, I compare varying values for two parameters of CHC++, namely the frequency at which non-occluded nodes are rechecked for occlusion and the depth used for creating Tighter Bounding Volumes.

4.1 VFC vs. CHC++

As expected, occlusion culling using CHC++ improves performance in most scenes. In Figure 4.1 we see some peaks in VFC as well as CHC++ that result from turns in the flythrough for scene "Project Endeavor" (see Section 3.2.2). As the flythrough closes in on the border of the scene, those peaks dissipate in both scenarios, since VFC – which is also used in the CHC++ algorithm – removes most of the nodes already. Still, for this challenging scene, we already see an improvement in render time of up to 81 milliseconds (27.1%) per frame on peaks. As the dynamic flythrough ends with an overview of the scene, almost no occlusion is present. At the end of the flythrough, where the whole scene is shown from above, we see that the advantage of using CHC++ is almost eliminated, but still performs at least as well as VFC alone.

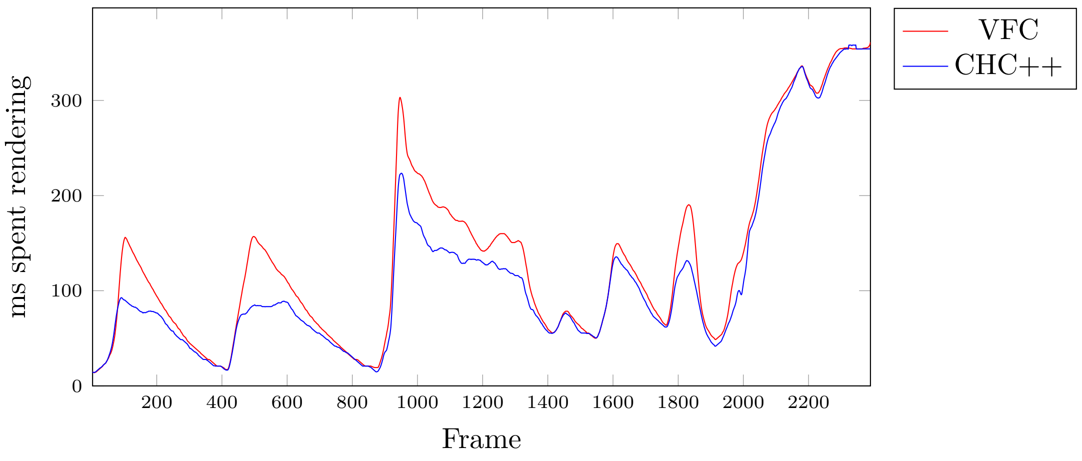

Figure 4.1: Render time per frame when using VFC as opposed to CHC++ for a flythrough through scene "Project Endeavor"

When we examine Figure 4.2 for the average-case scene "Arènes de Lutèce" (see Section 3.2.3), we start to see an intense improvement in performance. While initializing the point cloud, CHC++ already clearly surpasses VFC. This is accomplished by not loading occluded nodes that hide inside the side passages. Especially U-turns towards previously unloaded data cause a severe spike in render time while using only VFC. On the first U-turn alone, CHC++ reduces render time by up to 227 milliseconds (79.7%).

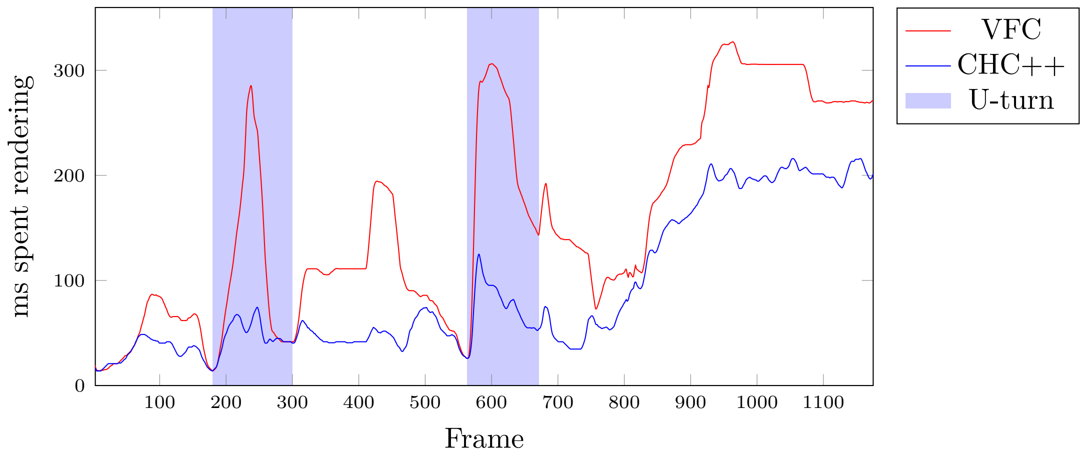

Figure 4.2: Render time per frame when using VFC as opposed to CHC++ for a flythrough through scene "Arènes de Lutèce". U-turns best represent the performance increase introduced by CHC++.

The scene "Office" (see Section 3.2.4) is where CHC++ really starts to shine. This dynamic flythrough travels through all the rooms on the first floor, continues with the second floor to visit one more room and finally follows the staircase upwards to the tenth floor as if climbing the stairs. Afterward, the camera rises through the ceiling to show the whole building from the outside. As we can see in Figure 4.3, the performance when visiting the rooms already increases clearly when using CHC++. But the best example for the advantages of CHC++ over VFC we encounter when looking at the part of the flythrough where the camera moves up the staircase. In this case, CHC++ avoids loading all the rooms and thus achieves to keep a constant frame time. VFC, on the other hand, has to load and render all the rooms in the view frustum on every U-turn and suffers from grave performance loss. The maximum reduction in render time while the camera moves through the staircase amounts to 55 milliseconds (79.4%). On rising to show the whole building we can even observe a performance gain of 98 milliseconds (77.9%) for CHC++.

To examine the performance of CHC++ in comparison to VFC when no occlusion is present, I used the scene "Sphere" (as described in Section 3.2.1). Figure 4.4 shows that CHC++ almost performs as well as VFC, even if not a single occluded node is available. On first sight, it appears that CHC++ performs better when initializing the point cloud for the flythrough. The reason for this observed performance improvement lies in my implementation of the CHC++ algorithm. In the CHC++ algorithm, occlusion queries are only created for leaf nodes, since the assumed data structure only stores points in its leaf nodes. The Potree data structure, on the other hand, has points stored in all nodes. To incorporate this difference in data structure into my implementation of CHC++,

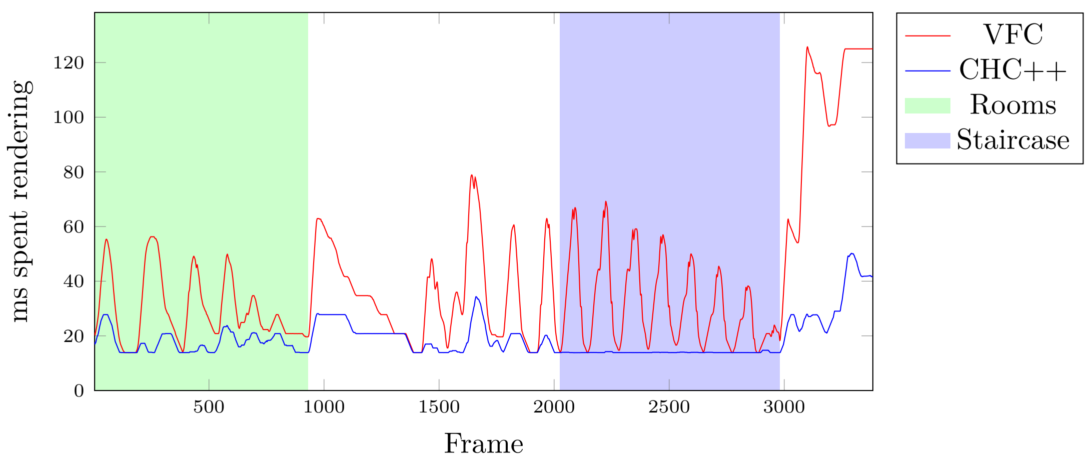

Figure 4.3: Render time per frame when using VFC as opposed to CHC++ for a flythrough through scene "Office". While loading the rooms shows a clear increase in performance, traveling up the staircase even achieves an almost constant render time when using CHC++.

I avoided waiting for the result of the occlusion queries for previously invisible nodes. Instead, I consider the results in the next frame – resulting in the initial loading delay we see in Figure 4.4.

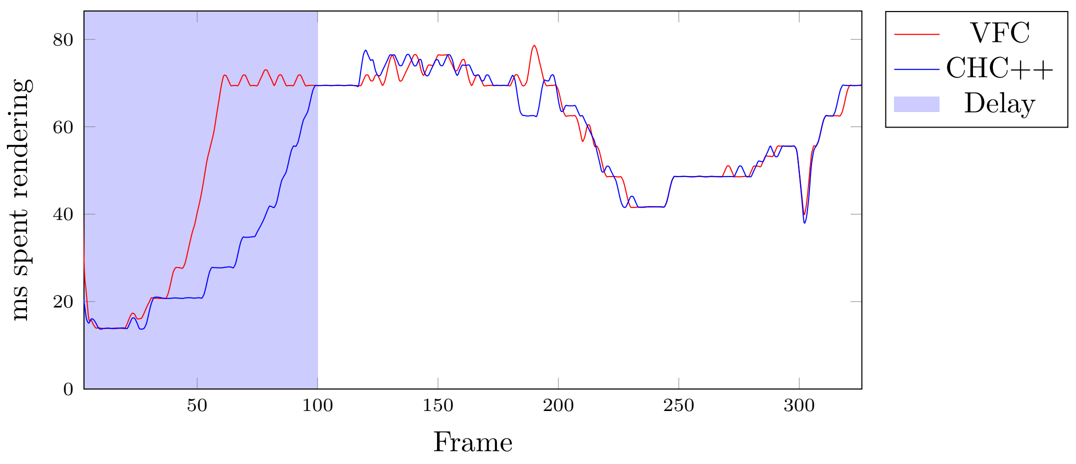

Figure 4.4: Render time per frame when using VFC as opposed to CHC++ for a flythrough through scene "Sphere". The highlighted delay is introduced by my implementation of CHC++, which delays the result of querying invisible nodes until next frame.

In Table 4.1 I provide an overview of the average render time per frame when using view frustum culling as opposed to CHC++. The scenes are ordered by the challenge they pose to the CHC++ algorithm, as described in Section 3.2. As we can see in the table the performance of CHC++ improves the more occlusions the scene offers.

The results for the scene "Sphere" have to be interpreted carefully, since the initial delay originating from my implementation leads to a longer period with a decreased render time for CHC++. When executing the flythrough for scene "Sphere" with the `--waitForLoadComplete` parameter we receive a different result. This can be seen in Table 4.2. With this additional parameter, we can observe a small decrease in performance when loading the point cloud with CHC++.

	Load method	
Scene	**VFC**	**CHC++**
Sphere	56.2 ms ±0.0%	51.2 ms −8.9%
Project Endeavor	134.8 ms ±0.0%	116.5 ms −13.6%
Arènes de Lutèce	158.8 ms ±0.0%	91.1 ms −42.6%
Office	40.6 ms ±0.0%	18.9 ms −53.4%
	Average render time	

Table 4.1: Comparison of the average render time per scene using view frustum culling versus CHC++.

	Load method	
Scene	**VFC**	**CHC++**
Sphere	66 ms ±0.0%	67.8 ms +2.7%
Project Endeavor	139.2 ms ±0.0%	116.6 ms −16.2%
Arènes de Lutèce	157.7 ms ±0.0%	86.7 ms −45.0%
Office	42 ms ±0.0%	19 ms −54.8%
	Average render time	

Table 4.2: Comparison of the average render time per scene using view frustum culling versus CHC++ with the `--waitForLoadComplete` parameter described in Section 3.1.2.

4.2 HDD vs. SSD

When switching from a hard disk drive to a solid-state disk we expect a sharp increase in performance when considering the loading speed. Against any expectation, my evaluation shows that the benefits of such a switch are only minimal. As we see in Figure 4.5 with a node size of 20,000 loading the scene "Arènes de Lutèce" using an SSD only takes 3 frames (2.4%) less than using an HDD.

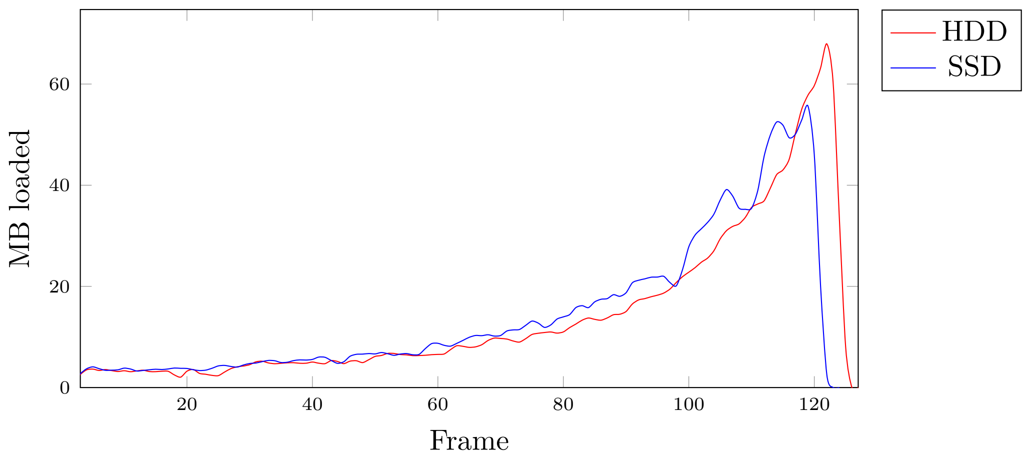

Figure 4.5: Data loaded per frame during initialization of scene "Arènes de Lutèce" when using an HDD as opposed to an SSD with a node size of 20,000

To further analyze this observation I use the same scene with the same static view, but increase the node size of the Potree data structure to 200,000. This way larger chunks of data can be loaded at once, and my specific implementation has less impact on the result. Increasing the node size increases the average file size from an average of 50 kilobytes to 500 kilobytes per file. The first noticeable fact seen in Figure 4.6 is that scene initialization requires 57 frames (44.9%) less when increasing the node size from 20,000 to 200,000. Nevertheless, using an SSD over an HDD still only decreases the number of frames required by 4 (5.7%). To gain intense improvements from using an SSD a very large node size has to be used. But as we will see in Section 4.3, larger node sizes have a downside to them.

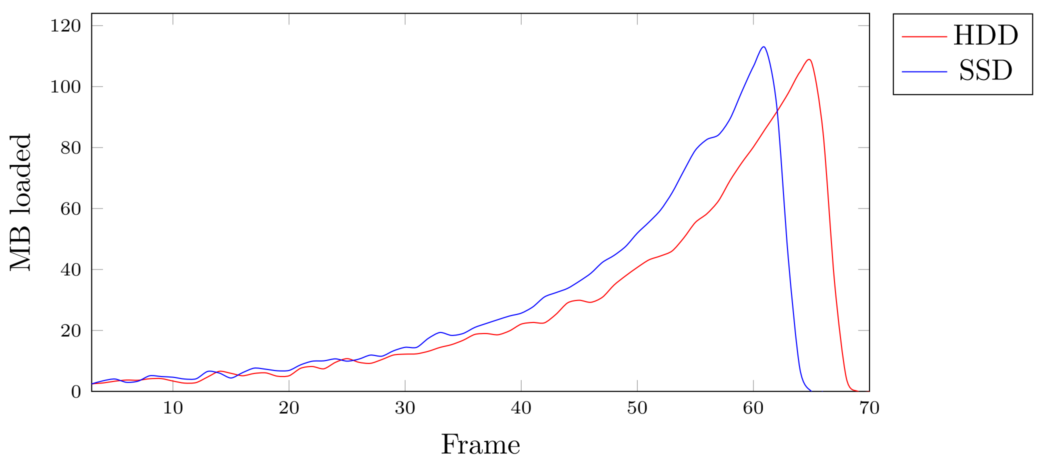

Figure 4.6: Data loaded per frame during initialization of scene "Arènes de Lutèce" when using an HDD as opposed to an SSD with a node size of 200,000

4.3 Node size

The chosen node size for the Potree data structure has a big influence on many aspects of my implementation. Varying the node size changes the time the point cloud requires to load from the drive and the time nodes upload to the GPU, for example. When we choose a large node size, the resulting data structure will not be as deep as when using a small node size. Thus, different node sizes also influence the number of nodes that can be occluded.

In my tests, I monitored the time it took to render the last frame. I expected this result to show the final, stabilized render time. But the values I received changed massively between test runs. To receive a stable result, I changed my implementation to render 30 frames after the loading process completes. This stabilization phase enabled me to acquire constant end results and enhances comparability immensely.

As we can see in Figure 4.7 for scene "Arènes de Lutèce", the higher we set the node size the higher the average frame time will be. This observation also stays true throughout a dynamic flythrough. This downside comes with a benefit though. As previously described in Section 4.2 increasing the node size also increases the speed at which the nodes in the view frustum are loaded. When initializing the scene with a node size of 80,000 as opposed to 5,000, we achieve a decrease of 122 frames (58.1%) required for loading the point cloud.

The same phenomenon can also be observed for the loading speed when looking at Figure 4.8. We see more megabytes loaded per frame the higher the node size, resulting from the overhead of accessing many different files in a short time.

In Figure 4.9 we see that render time is equal for all tested node sizes initially for scene "Office". As more occluded data inside the office building gets loaded by larger node

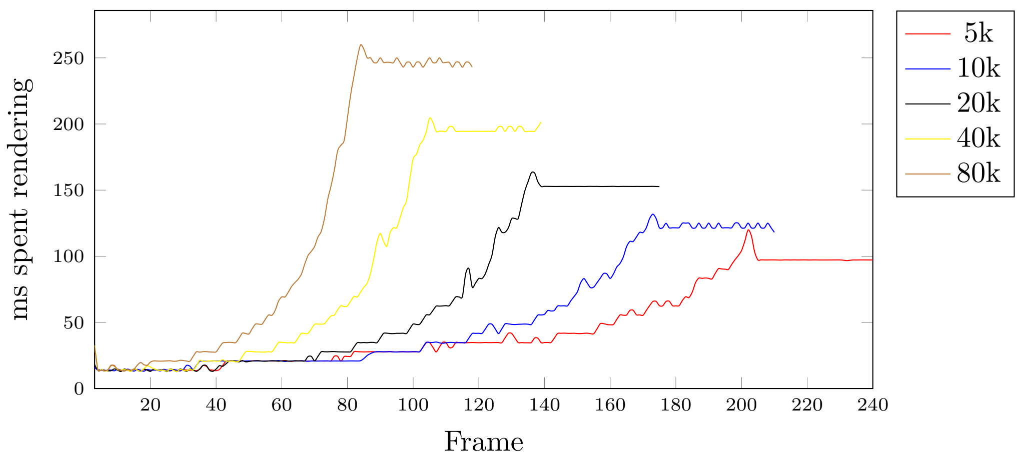

Figure 4.7: Render time per frame during initialization of scene "Arènes de Lutèce" when comparing varied node sizes

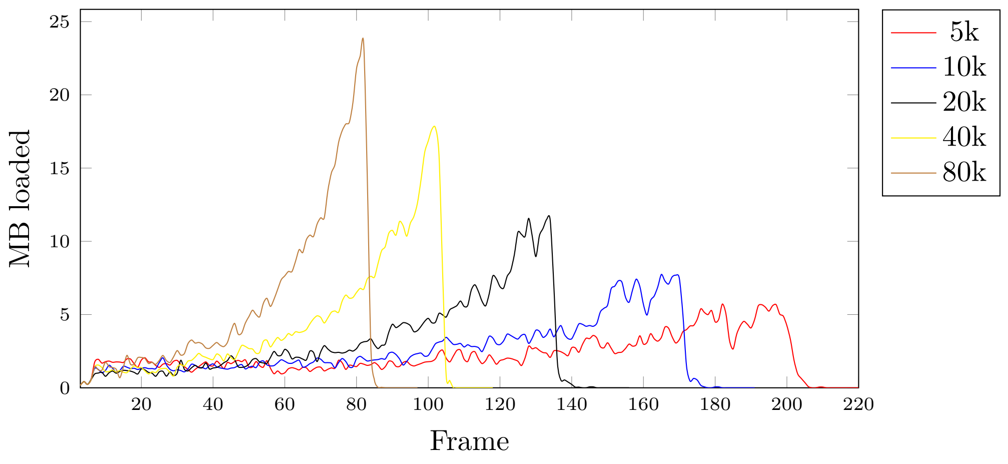

Figure 4.8: Data loaded per frame during initialization of scene "Arènes de Lutèce" when comparing varied node sizes

sizes, smaller node sizes achieve to keep a relatively low render time by allowing efficient occlusion culling of interior nodes. After loading all necessary data, we end up with a frame time 27.8 milliseconds (44.5%) faster when using a node size of 5,000 as opposed to 80,000. As we can see, choosing the node size is a matter of preference. If the aim is to achieve less fluctuating frame rates, we should choose a small node size for stability. If a more static approach is desired, where loading data fast is preferred, we should choose a large node size.

In Table 4.3 I present an overview of the observed decrease in the load speed. Please note that I excluded the stabilization frames to be able to present more accurate percentage-wise differences. As we can see, increasing the node size constantly decreases the number

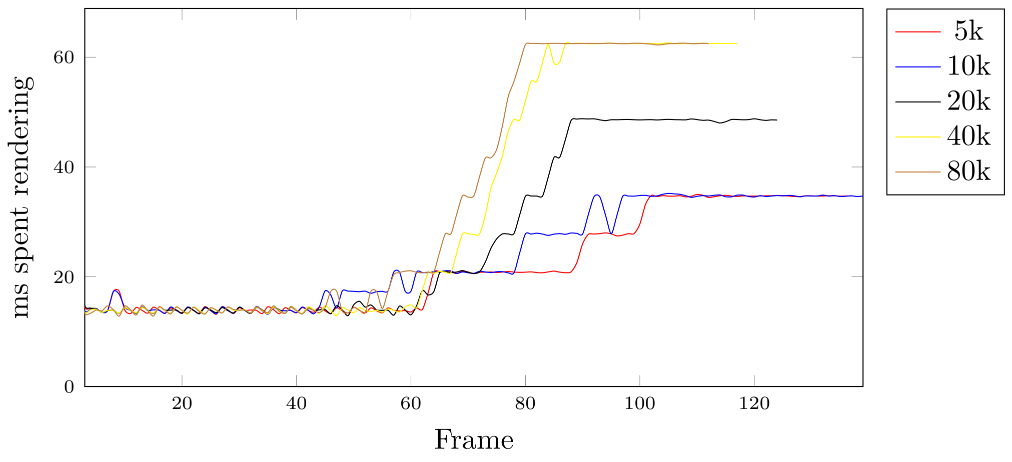

Figure 4.9: Render time per frame during initialization of scene "Office" when comparing varied node sizes

of frames required for initialization for every scene. I present the increase in render time when increasing the node size in Table 4.4, which shows the increase in render time when increasing the node size. The reason for this we see in Table 4.5. As the node size increases, more unnecessary points get loaded. These unnecessary points are either located outside the view frustum (for scene "Sphere") or would else be occluded. In combination, these tables show the trade-off between the speed of the loading and the rendering process.

There are some inconsistencies in these tables. Scene "Office" delivers unchanged render times when changing the node size from 5,000 to 10,000 and from 40,000 to 80,000. The reason is the small number of unnecessarily loaded points added by this increase. Regarding the scene "Sphere", my tests delivered varied results. The number of frames did not constantly reduce with an increase in node size for every test. Nonetheless, the increase in render time was always constant for an increase in node size. Having no occlusions seems to destabilize the loading speed, making the results less viable. I still include them for completeness.

	Node size				
Scene	**5,000**	**10,000**	**20,000**	**40,000**	**80,000**
Sphere	127	90	94	63	65
	±0.0%	−29.1%	−26.0%	−50.4%	−48.8%
Project Endeavor	253	184	182	121	92
	±0.0%	−27.3%	−28.1%	−52.2%	−63.6%
Arènes de Lutèce	210	180	145	109	88
	±0.0%	−14.3%	−31.0%	−48.1%	−58.1%
Office	107	109	94	87	82
	±0.0%	+1.9%	−12.1%	−18.7%	−23.4%
	Frames required to load scene				

Table 4.3: Comparison of the number of frames required to initialize each scene with varied node sizes

	Node size				
Scene	**5,000**	**10,000**	**20,000**	**40,000**	**80,000**
Sphere	90.3 ms	125.1 ms	131.9 ms	145.8 ms	159.6 ms
	±0.0%	+38.5%	+46.1%	+61.5%	+76.7%
Project Endeavor	194.6 ms	284.7 ms	361.8 ms	375.1 ms	423.6 ms
	±0.0%	+46.3%	+85.9%	+92.8%	+117.7%
Arènes de Lutèce	97.2 ms	118.1 ms	152.7 ms	201.4 ms	243 ms
	±0.0%	+21.5%	+57.1%	+107.2%	+150.0%
Office	34.7 ms	34.7 ms	48.6 ms	62.5 ms	62.5 ms
	±0.0%	±0.0%	+40.1%	+80.1%	+80.1%
	Last render time				

Table 4.4: Comparison of the render time after each scene completed loading with varied node sizes

	Node size				
Scene	**5,000**	**10,000**	**20,000**	**40,000**	**80,000**
Sphere	8,259,400	12,779,400	13,112,000	15,071,800	16,192,500
	±0.0%	+54.7%	+58.8%	+82.5%	+96.0%
Project Endeavor	19,096,900	28,963,200	37,510,300	39,032,700	44,711,500
	±0.0%	+51.7%	+96.4%	+104.4%	+134.1%
Arènes de Lutèce	8,782,480	11,821,000	15,088,600	20,075,700	25,618,600
	±0.0%	+34.6%	+71.8%	+128.6%	+191.7%
Office	2,244,990	2,625,920	4,288,610	5,347,400	5,582,550
	±0.0%	+17.0%	+91.0%	+138.2%	+148.7%
	Rendered points				

Table 4.5: Comparison of the number of rendered points after each scene completed loading with varied node sizes

4.4 Occlusion check frequency for visible nodes

To reduce the number of occlusion queries, CHC++ does not query all nodes in each frame as described in Section 2.4. Nodes that were considered non-occluded recently are only rechecked after a variable number of frames. Reducing the occlusion check frequency can potentially increase performance where occlusion queries are expensive. In this section, I analyze the trade-off between rendering nodes that are already occluded and increasing the number of occlusion queries per node.

As we can see in Figure 4.10 for the scene "Project Endeavor", my implementation manages to render at best performance when querying visible nodes every 5 frames throughout most parts of the flythrough. This is due to the dynamic walking-style path chosen for the flythrough. Since many nodes are visible for a small number of frames, querying visible nodes less often results in performance loss due to unnecessarily rendered nodes.

On frame 875 the flythrough reaches the edge of the scene and takes a U-turn. My implementation of CHC++ stops counting the number of frames left until a new occlusion query is required when a node leaves the view frustum. As the node re-enters the view frustum the frame count is continued where it left off. In this flythrough, the part of the scene that the camera turns toward was previously visited from the other side of the scene. As the camera turns these nodes are now still considered visible and will immediately be rendered, although they are mostly occluded at this time. While rechecking often manages to resolve this situation quickly, checking less often continues to render these unnecessary nodes longer. This results in up to 832 nodes (+27.5%) that get rendered additionally when rechecking visible nodes every 80 frames as opposed to rechecking every 5 frames. The frame time, in this case, is increased by 42 milliseconds (+23.4%) when rendering those additional nodes.

At the end of the flythrough, when the camera pans out to show the whole scene, the view is static enough for nodes to stay visible for 80 frames or more. Thus, we start to see an improvement in render time when using a higher occlusion recheck delay.

Figure 4.11 clearly shows the expected result when examining the difference in the number of occlusion queries between the different recheck delays. We can observe that the number of occlusion queries increases drastically when doubling the occlusion check frequency.

When rendering the scene "Office" – which represents the best-case scenario for CHC++ as described in Section 3.2.4 – every used occlusion recheck delay results in almost the same performance as seen in Figure 4.12. The reason can be seen in Figure 4.13. This scene contains a lot of occluded data yielding just a small number of nodes whose visibility has to be rechecked. Additionally, the dynamic nature of the flythrough only rarely leads to nodes that are visible for an extended period of time. Except for some peaks, big differences only start to show at the end, when the whole scene is in the view frustum. On rendering the whole scene a maximum of 304 nodes (51.2%) are unnecessarily rendered when using an occlusion check frequency of 80, resulting in an increase of 7.4 milliseconds

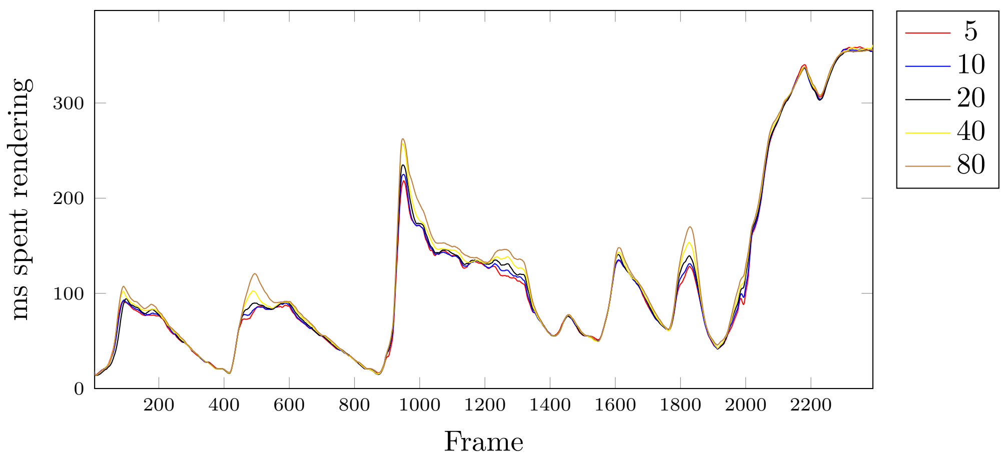

Figure 4.10: Render time per frame using varied occlusion check frequencies for a flythrough through scene "Project Endeavor"

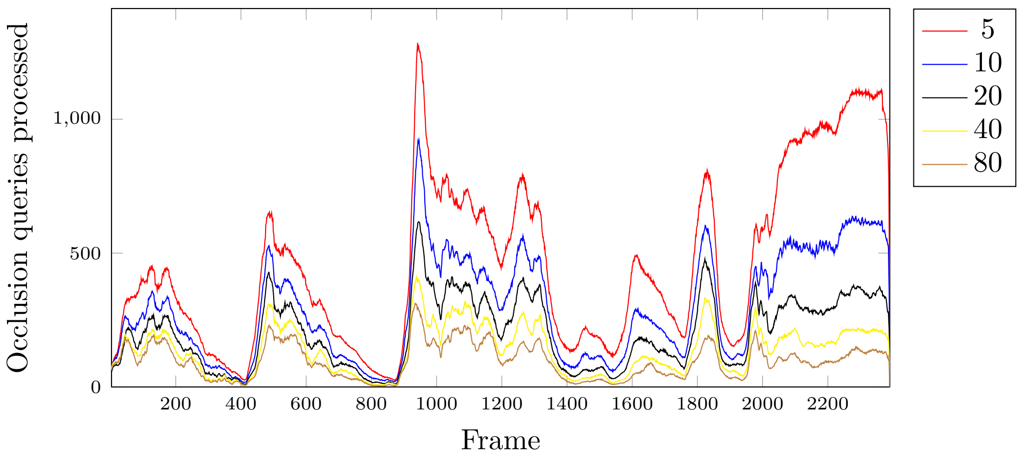

Figure 4.11: Number of occlusion queries per frame using varied occlusion recheck delays for a flythrough through scene "Project Endeavor"

(+16.6%) in render time.

In Table 4.6 I present an overview of the change in average render time when altering the occlusion recheck delay. Scene "Sphere" has been excluded due to the missing pan-out in the fly-through and the resulting incomparability. As we increase the parameter's value, we can see a constant performance decrease. The reason for the high percentage-wise difference between the scenes "Project Endeavor" and "Arènes de Lutèce" lies in the different ratio between the dynamic part of the flythrough and the pan-out. The flythrough for scene "Project Endeavor" has a ratio of 83% flythrough to 17% pan-out as opposed to 42% flythrough to 58% pan-out for scene "Arènes de Lutèce". The ratio between the relative lengths of the pan-out is thus 3.4. If we apply this ratio to the

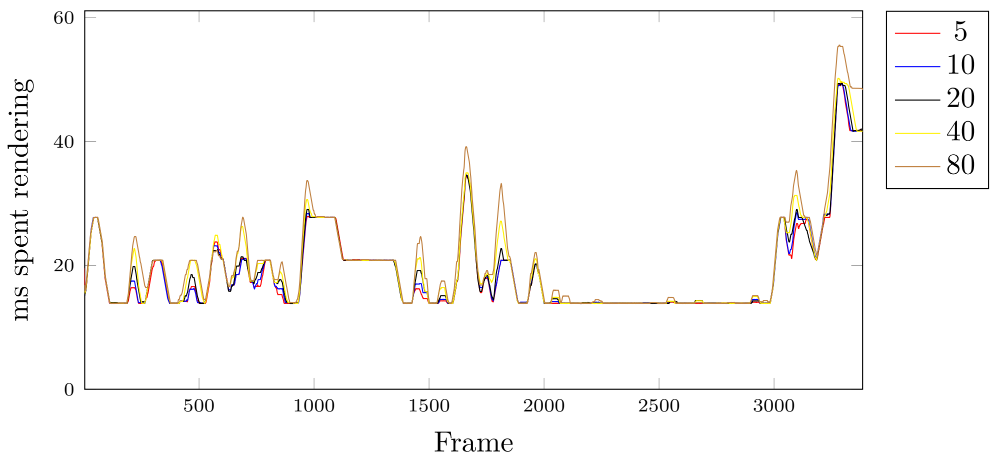

Figure 4.12: Render time per frame using varied occlusion check frequencies for a flythrough through scene "Office"

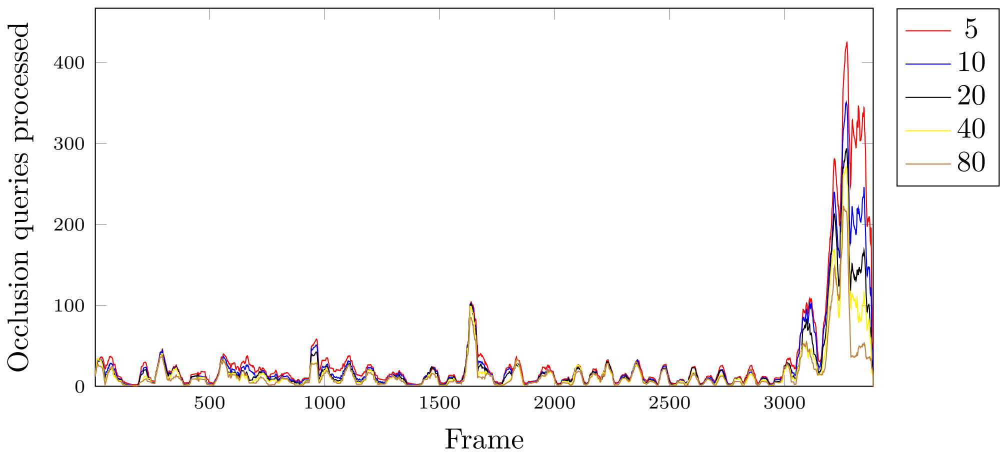

Figure 4.13: Number of occlusion queries per frame using varied occlusion check frequencies for a flythrough through scene "Office"

percentage-wise decrease in performance in scene "Project Endeavor", we receive a value that is very close to the percentages of scene "Arènes de Lutèce".

	Occlusion recheck delay				
Scene	**5**	**10**	**20**	**40**	**80**
Project Endeavor	116.1 ms ±0.0%	116.9 ms +0.7%	118.3 ms +1.9%	121.1 ms +4.3%	124.1 ms +6.9%
Arènes de Lutèce	88.7 ms ±0.0%	91.4 ms +3.0%	95.7 ms +7.9%	102 ms +15.0%	109.2 ms +23.1%
Office	18.7 ms ±0.0%	18.9 ms +1.1%	19 ms +1.6%	19.5 ms +4.3%	20.3 ms +8.6%
	Average render time				

Table 4.6: Comparison of the average render time per scene with varied occlusion recheck delays. Scene "Sphere" is excluded, as described in Section 4.4.

4.5 Tighter Bounding Volume depth

"Tighter Bounding Volumes" [MBW08] is a concept introduced by CHC++. It reduces the empty space included in bounding volumes as described in Section 2.4. Tightening the bounding volume increases the probability that nodes will be occluded since the content of a node is better represented by its bounding volume. Let us consider a person is standing behind a low wall where the person is completely occluded. An algorithm that uses the default bounding box might still consider the person as non-occluded since its bounding box may reach above the bounding box of the wall. But if we instead use a more tightly fitted bounding box for the person as well as the wall, we receive a more precise approximation of the object's size. Thus, using Tighter Bounding Volumes, we reduce the number of false negatives when checking for occlusion and end up with fewer nodes to render. On the other hand, iterating a certain depth into every bounding box introduces additional computational overhead. In this section, I evaluate the benefits of using more accurate Tighter Bounding Volumes versus using the unchanged bounding box.

As we can see in Figure 4.14, changing the depth when creating Tighter Bounding Volumes only produces a negligible difference in render time for the scene "Arènes de Lutèce". The same is true for every scene in my evaluation. When examining the effects of changing the Tighter Bounding Volume depth on a smaller scale in Figure 4.15 – by only taking the time spent for occlusion culling into consideration – we can see that using a Tighter Bounding Volume depth of 4 increases occlusion time by a maximum of 2.6 milliseconds (+47.4%) as opposed to using the original bounding box.

To further analyze the difference between a depth of 0 and a depth of 4, I calculated the difference between average rendering times. The result is an increase of 2.8 milliseconds (+3.2%) in render time per frame when using a Tighter Bounding Volume depth of 4.

Scene "Office" presents a very similar picture when we look at Figure 4.16. Applying the same calculations as mentioned before to this flythrough results in an increase in render time of 70 nanoseconds (+0.4%) per frame when using a Tighter Bounding Volume depth

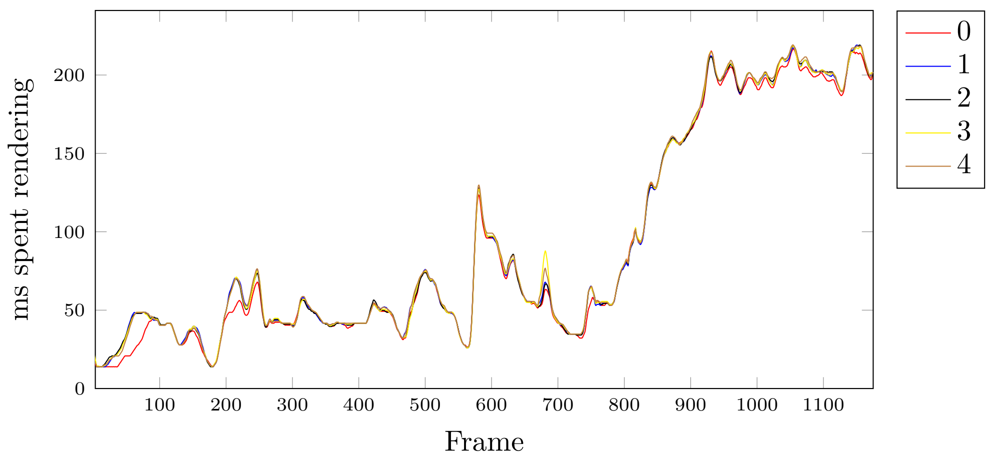

Figure 4.14: Render time per frame using varied Tighter Bounding Volume depths for a flythrough through scene "Arènes de Lutèce"

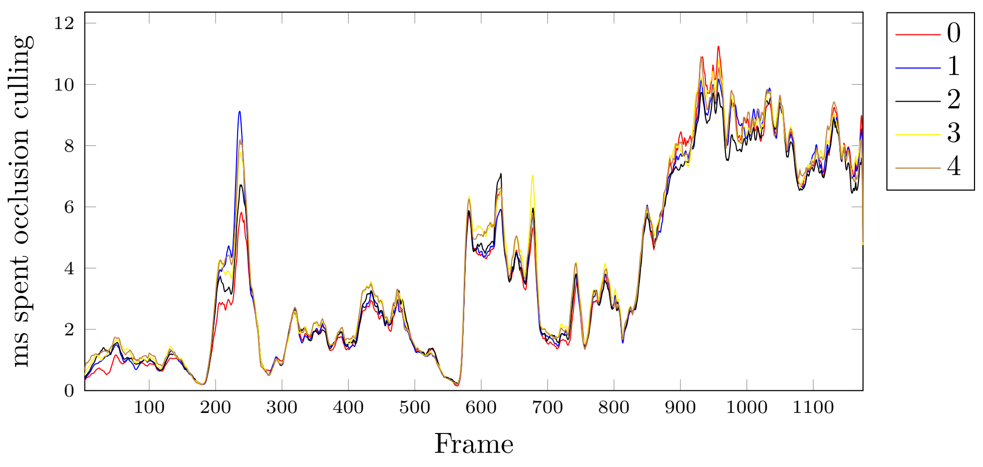

Figure 4.15: Occlusion culling time per frame using varied Tighter Bounding Volume depths for a flythrough through scene "Arènes de Lutèce"

of 4 as opposed to the unaltered bounding box. Although the average performance loss is only marginal, I was unable to achieve the expected decrease in render time – at least with the scenes I used in this evaluation.

In Table 4.7 we see that render time does not constantly increase when increasing the Tighter Bounding Volumes depth. What remains consistent is that every scene in this evaluation performed marginally better when not using Tighter Bounding Volumes. This fact is most prominent for the scene "Arènes de Lutèce". Like in Section 4.5, scene "Sphere" has been excluded from this table due to the pan-out missing from the fly-through and the resulting incomparability.

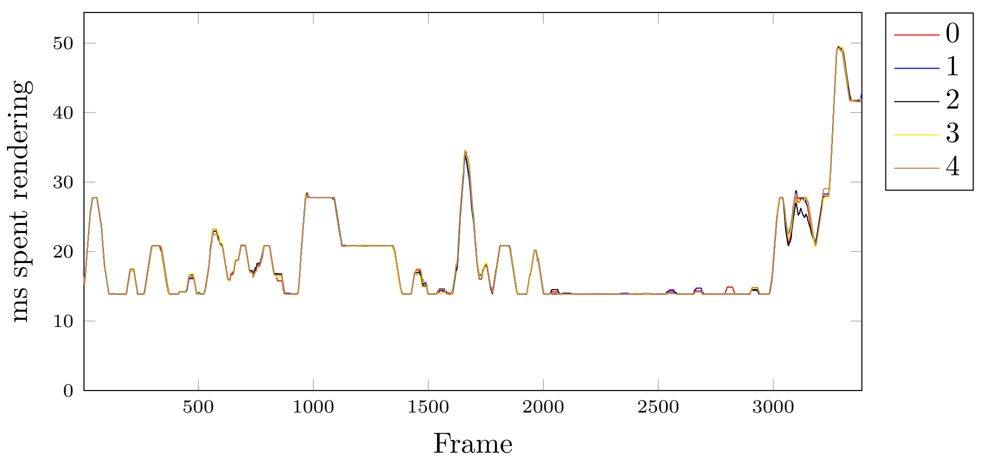

Figure 4.16: Render time per frame using varied Tighter Bounding Volume depths for a flythrough through scene "Office"

	Tighter Bounding Volume depth				
Scene	**0**	**1**	**2**	**3**	**4**
Project Endeavor	115.428 ms ±0.0%	117.224 ms +1.6%	116.671 ms +1.1%	117.081 ms +1.4%	117.465 ms +1.8%
Arènes de Lutèce	89.467 ms ±0.0%	91.663 ms +2.5%	91.616 ms +2.4%	91.956 ms +2.8%	92.321 ms +3.2%
Office	18.802 ms ±0.0%	18.813 ms +0.1%	18.805 ms +0.0%	18.822 ms +0.1%	18.872 ms +0.4%
	Average render time				

Table 4.7: Comparison of the average render time per scene with varied Tighter Bounding Volume depths. Scene "Sphere" is excluded, as described in Section 4.4.

CHAPTER 5

Conclusion

As shown in my evaluations, CHC++ [MBW08] heavily improves the rendering performance for real-life scenes like laser and drone scans. Most of the parameters tested resulted in a decision of preference or use case. While large node sizes for the Potree data structure [Sch16] manage to load scenes quickly, they have an over-proportional negative impact on render time required per frame, making it more useful for a static rendering style, where the number of frames per second is of little importance. Choosing a small node size, on the other hand, reverses these effects and allows for smoother user experience when navigating through a scene with the drawback of longer waiting times for details to load. This more dynamic approach can be further improved by combining it with a faster occlusion check frequency for visible nodes.

Due to the small node-file sizes I received from the evaluated values, using an SSD over an HDD did not achieve a significant improvement in loading speed. Although the node size could be further increased, the improvements introduced by CHC++ would suffer from the resulting shallow tree, rendering large node sizes unpractical for occlusion culling.

Surprisingly, Tighter Bounding Volumes introduced a small decrease in performance, independent of the depth used. This increase in render time might result from the scenes or the flythroughs used in this evaluation.

It shows that, when using modern-day graphics hardware, the values for the node size and the occlusion check frequency should be chosen as low as possible when a smooth user-experience is desired. Using small parameter values reduces the influence the rendered scene has on the result. If a static representation of the scene is desired, where the camera is not moved or rotated frequently, faster loading times can be achieved by increasing the node size at the cost of reduced rendering performance. In this case, scene choice has a big influence on rendering speed. The exact influence on rendering speed depends on many factors, like the number of nodes barely contained in the view frustum and nodes that would be occluded if there was another level of depth. The exact impact on performance is thus very situation-specific and can not be estimated beforehand. But since a static rendering style does not rely on fast rendering, the exact influence of scene choice is of little importance.

5.1 Future work

No specific recommendations can be made for the evaluated parameters since they are very dependent on the rendered scene, the user's preference and the exact interaction with the scene. To reduce the effort of hand-picking parameter values and to improve performance, an algorithm could be implemented that dynamically adjusts parameters like the occlusion check frequency and node size based on live measurements during execution.

For example, we could measure the number of frames a node is visible on average. Based on this value, we can choose the occlusion check frequency at run-time to minimize the number of unnecessary occlusion queries. To set the node size, we could collect the average number of occluded nodes. As the number of occluded nodes decreases, the node size can be increased dynamically. The exact threshold for the number of occluded nodes depends on the trade-off between using a larger node size with fewer occlusion queries and using a smaller node size with more occlusion queries. A static value for this threshold could be established by analyzing different scenes and scenarios. An algorithm that changes parameters during executing should also check for sudden changes in its measurements, to quickly adapt to a change in user behavior.

To provide the possibility to alter the node size at run-time, another algorithm has to be designed. Since hard drives benefit from larger file sizes, a larger node size would be desirable. To preserve the benefits introduced by occlusion culling, the node size used during execution has to be independent of the node size used to create the data structure. The impact on performance introduced by processing nodes in this way has to be kept at a minimum, but at least lower than the time saved when loading using a larger node size and rendering at flexible node sizes.

List of Figures

2.1 Example of multiple levels of the "modifiable nested octree" (MNO) [Sch14] and Potree data structure [Sch16] respectively. Image by [Sch16]. 4

2.2 Types of state changes ordered by cost. CHC++ [MBW08] significantly reduces the number of state changes, which typically are changes of "Render Target" and "Program". Extracted from page 48 of [EM14]. 5

3.1 A screenshot of UPCR — my point cloud renderer implemented to collect the necessary data for the evaluations in this thesis. The loaded point cloud presents the Nvidia Endeavor Campus in Santa Clara, California during construction, see Section 3.2.2. 10

3.2 One possible way to compare data. For this sample I ran UPCR twice. On the left it runs with CHC++ [MBW08] and renders the statistics on the right. On the right it runs with VFC and renders the statistics on the left. I recorded both runs using the parameter `--recordFileName` and merged the resulting videos in a post-processing step using FFmpeg. The loaded point cloud is a scan of the Arènes de Lutèce, see section 3.2.3. 13

4.1 Render time per frame when using VFC as opposed to CHC++ for a flythrough through scene "Project Endeavor" 18

4.2 Render time per frame when using VFC as opposed to CHC++ for a flythrough through scene "Arènes de Lutèce". U-turns best represent the performance increase introduced by CHC++. 19

4.3 Render time per frame when using VFC as opposed to CHC++ for a flythrough through scene "Office". While loading the rooms shows a clear increase in performance, traveling up the staircase even achieves an almost constant render time when using CHC++. 20

4.4 Render time per frame when using VFC as opposed to CHC++ for a flythrough through scene "Sphere". The highlighted delay is introduced by my implementation of CHC++, which delays the result of querying invisible nodes until next frame. 20

4.5 Data loaded per frame during initialization of scene "Arènes de Lutèce" when using an HDD as opposed to an SSD with a node size of 20,000 22

4.6 Data loaded per frame during initialization of scene "Arènes de Lutèce" when using an HDD as opposed to an SSD with a node size of 200,000 23
4.7 Render time per frame during initialization of scene "Arènes de Lutèce" when comparing varied node sizes 24
4.8 Data loaded per frame during initialization of scene "Arènes de Lutèce" when comparing varied node sizes 24
4.9 Render time per frame during initialization of scene "Office" when comparing varied node sizes 25
4.10 Render time per frame using varied occlusion check frequencies for a flythrough through scene "Project Endeavor" 28
4.11 Number of occlusion queries per frame using varied occlusion recheck delays for a flythrough through scene "Project Endeavor" 28
4.12 Render time per frame using varied occlusion check frequencies for a flythrough through scene "Office" 29
4.13 Number of occlusion queries per frame using varied occlusion check frequencies for a flythrough through scene "Office" 29
4.14 Render time per frame using varied Tighter Bounding Volume depths for a flythrough through scene "Arènes de Lutèce" 31
4.15 Occlusion culling time per frame using varied Tighter Bounding Volume depths for a flythrough through scene "Arènes de Lutèce" 31
4.16 Render time per frame using varied Tighter Bounding Volume depths for a flythrough through scene "Office" 32

List of Tables

3.1 Key-bindings used in UPCR. Additionally, the camera can be turned by holding the left mouse button down and moving the mouse. 11

4.1 Comparison of the average render time per scene using view frustum culling versus CHC++. 21
4.2 Comparison of the average render time per scene using view frustum culling versus CHC++ with the `--waitForLoadComplete` parameter described in Section 3.1.2. 21
4.3 Comparison of the number of frames required to initialize each scene with varied node sizes 26
4.4 Comparison of the render time after each scene completed loading with varied node sizes 26
4.5 Comparison of the number of rendered points after each scene completed loading with varied node sizes 26
4.6 Comparison of the average render time per scene with varied occlusion recheck delays. Scene "Sphere" is excluded, as described in Section 4.4. 30
4.7 Comparison of the average render time per scene with varied Tighter Bounding Volume depths. Scene "Sphere" is excluded, as described in Section 4.4. . 32

List of Algorithms

Bibliography

[BWPP04] Jiří Bittner, Michael Wimmer, Harald Piringer, and Werner Purgathofer. Coherent hierarchical culling: Hardware occlusion queries made useful. In *Computer Graphics Forum*, volume 23, pages 615–624. Wiley Online Library, 2004.

[Con16] The Harvest4D Consortium. Creators of the point cloud "arènes de lutèce". `https://harvest4d.org/?page_id=1367`, 2016. [accessed July 2, 2019].

[EM14] Cass Everitt and John McDonald. Beyond porting. `https://www.slideshare.net/CassEveritt/beyond-porting`, 2014. [accessed May 5, 2019].

[MBW08] Oliver Mattausch, Jiří Bittner, and Michael Wimmer. Chc++: Coherent hierarchical culling revisited. In *Computer Graphics Forum*, volume 27, pages 221–230. Wiley Online Library, 2008.

[Nvi16] Nvidia. Creators of the point cloud "project endeavor". `https://www.nvidia.com/`, 2016.

[Sch14] Claus Scheiblauer. *Interactions with gigantic point clouds.* PhD thesis, Technische Universität Wien, 2014.

[Sch16] Markus Schuetz. *Potree: Rendering Large Point Clouds in Web Browsers.* PhD thesis, Technische Universität Wien, 2016.