Novel Parallelization Techniques for Computer Graphics Applications

Master of Science Thesis
University of Turku
Department of Future Technologies
Cadmatic
2021
Diego Sanz Villafruela

Supervisors:
Dr Mohammad-Hashem Haghbayan
Prof Juha Plosila

UNIVERSITY OF TURKU
Department of Future Technologies

Diego Sanz Villafruela: Novel Parallelization Techniques for Computer Graphics Applications

Master of Science Thesis, 75 p.
Cadmatic
March 2021

Increasingly complex and data-intensive algorithms in computer graphics applications require software engineers to find ways of improving performance and scalability to satisfy the requirements of customers and users. Parallelizing and tailoring each algorithm of each specific application is a time-consuming task and its implementation is domain-specific because it can not be reused outside the specific problem in which the algorithm is defined. Identifying reusable parallelization patterns that can be extrapolated and applied to other different algorithms is an essential task needed in order to provide consistent parallelization improvements and reduce the development time of evolving a sequential algorithm into a parallel one.

This thesis focuses on defining general and efficient parallelization techniques and approaches that can be followed in order to parallelize complex 3D graphic algorithms. These parallelization patterns can be easily applied in order to convert most kinds of sequential complex and data-intensive algorithms to parallel ones obtaining consistent optimization results.

The main idea in the thesis is to use multi-threading techniques to improve the parallelization and core utilization of 3D algorithms. Most of the 3D algorithms apply similar repetitive independent operations on a vast amount of 3D data. These application characteristics bring the opportunity of applying multi-thread parallelization techniques on such applications. The efficiency of the proposed idea is tested on two common computer graphics algorithms: hidden-line removal and collision detection. Both algorithms are data-intensive algorithms, whose conversions from a sequential to a multithread implementation introduce challenges, due to their complexities and the fact that elements in their data have different sizes and complexities, producing work-load imbalances and asymmetries between processing elements.

The results show that the proposed principles and patterns can be easily applied to both algorithms, transforming their sequential to multithread implementations, obtaining consistent optimization results proportional to the number of processing elements. From the work done in this thesis, it is concluded that the suggested parallelization warrants further study and development in order to extend its usage to heterogeneous platforms such as a Graphical Processing Unit (GPU). OpenCL is the most feasible framework to explore in the future due to its interoperability among different platforms.

Keywords: hidden-line removal, collision detection, parallelism, computer graphics, Cadmatic, multi-threading

Contents

List of Figures

Abbreviations

API	Application Programming Interface
BVH	Bounding Volume Hierarchy
CAD	Computer-Aided Design
CAM	Computer-Aided Manufacturing
CD	Collision Detection
CPU	central processing unit
CREW	Concurrent Read Exclusive Write
CUDA	Compute Unified Device Architecture
FPGAs	Field-Programmable Gate Array
GPU	Graphical Processing Unit
HLR	Hidden-Line Removal
OpenCL	Open Computing Language
PLPP	Pattern Language for Parallel Programming
POSIX	Portable Operating System Interface
PRAM	Parallel Random-Access Machine

1 Introduction

Increasingly complex and data-intensive algorithms require software engineers to find ways of improving performance and scalability to satisfy requirements of customers and users in accordance to Lehman's first law of software evolution [1], in which a software application must constantly change or become more useful. In addition to that, *Moore's law* is not longer applicable due to thermal limitations with respect the number of transistors that could be fit on a chip [2], stagnating the exponential progress in processor performance prognosticated by Moore. In order to make sustainable these increasing rates of performance, it is necessary to evolve sequential algorithms into parallel algorithms by taking advantage of hardware acceleration techniques, such as multi-core processors or other parallel platforms [3].

Evolving sequential to parallel code is not a simple task, as Massingill [4] explains: "Creating parallel software is difficult, time-consuming and error-prone". Moreover, parallel software is often specifically tailored for each specific algorithm, preventing its solution to be applied to other algorithms. Additionally, there are some other challenges [5] when evolving sequential to parallel because parallel code is not as intuitive as sequential code, and many programmers are not properly trained in this field. In addition, programmers may not fully understand the algorithm they are trying to parallelize, as well as the data dependencies that are present in the code. These issues could lead to incorrect results when evolving a sequential to parallel algorithm. Even if the parallelization of an algorithm is successfully developed, its solution is not reusable because it can not be utilized outside the specific problem in which an algorithm is defined.

Identifying reusable parallelization solutions for commonly occurring optimization problems is essential in order to provide general high-performance computation techniques. This thesis focuses on defining general and efficient parallelization techniques and approaches that can be followed in order to improve the performance of 3D design applications when evolving sequential to parallel algorithms.

Computer graphics is the field that describes any use of computers to create and manipulate images [6]. Computer graphics involves numerous diverse areas such as modeling, rendering and animation. It also includes other minor areas such as user interaction, virtual reality, visualization, image processing, 3D scanning, etc. It is also present in many important applications such as video games, cartoons, Computer-Aided Design (CAD) and Computer-Aided Manufacturing (CAM) applications, medical imaging, etc [6].

This thesis is focuses on CAD/CAM applications, such as **Cadmatic**. CAD/-CAM applications are used to design parts and products on the computer and then, using these virtual designs, to guide the manufacturing process [6]. The usage of computer-aided design and computer-aided manufacturing tools is widely extended in many engineering fields such as marine and plant industries.

There are some studies and literature [4] about methodologies for parallel programmers collected in a set of patterns, such as the "Pattern Language for Parallel Programming (PLPP)". However, most of the methodologies described in those studies do not consider many aspects, such as data asymmetry, that causes imbalances among different threads due to the fact that elements have different complexities and sizes requiring different execution times. For these kind of problems, those methodologies are not suitable for a specific environment such as Computer Graphics, in which many algorithms deal with data asymmetry and other secondary issues.

Two kinds of complex algorithms that are investigated in this thesis to prove the efficiency of the proposed optimization techniques are: Hidden-Line Removal (HLR) and Collision Detection (CD). Both algorithms CD and HLR are widely used in CAD software applications, such as Cadmatic. Cadmatic is a 3D design tool for marine and plant industries. Cadmatic provides the framework and environment in which these two algorithms are run. Cadmatic is the commissioner of this thesis and has taken part actively in development of this master's thesis.

Nowadays it is possible to apply several **parallelization techniques** as most of the computers have multiple processor cores [7]. There is usually a highly parallel central processing unit (CPU) that possesses several physical and virtual cores, in addition to a GPU which has many small processing elements.

A normal CPU is an electronic circuitry design to carry out general tasks by executing instructions in a sequential order. CPUs are suitable for general computing tasks such as logic-arithmetic tasks, memory management, working with memory addresses and register, and for interacting with peripherals. CPUs' manufactures affront technical issues with higher clock frequencies, for this reason CPUs' performance is mostly currently improved by adding extra cores.

A GPU can also be used as a hardware acceleration technique. A GPU is an electronic circuitry designed to perform rapid mathematical calculations, primarily for the purpose of rendering images. GPUs have also evolved from fixed function rendering devices into programmable parallel processors.

A GPU is a very complicated system with many characteristic, and it can easily outperform a common CPU in processing of float numbers [8]. However, a CPU offers many additional features that GPU cannot give [9], such as virtual memory, preemption, interruption, controllable switch context, and I/O interaction. An additional difference between CPUs and GPUs is the way data that is handled. In the case of GPU the focus is data independence.

Advances on GPUs brought the possibility of executing small programs called kernels, allowing programmers to take full advantage of most of GPUs' hardware resources. Among the most important GPU frameworks are Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL).

CUDA was created by NVIDIA to provide an easy way to use GPU features in non-graphic problems. CUDA is based on a SIMD architecture in which the same piece of code is executed into many processing elements. CUDA was rapidly discarded for this work because of its lack of heterogeneity with non-NVIDIA hardware architectures.

OpenCL is a general purpose parallel programming framework for heterogeneous architectures [7] such CPUs, GPUs, Field-Programmable Gate Array (FPGAs), etc. OpenCL provides portable and efficient code to access these different types of processing hardware elements.

This master's thesis takes as case of study complex 3D-graphic algorithms: hidden-line removal and collision detection, these algorithms have data dependencies and include third party software that can not be loaded directly into a GPU.

Multi-threading, using several processing elements of a CPU, is the approach that will be followed in this thesis in order to provide general approaches and techniques of improving non-parallel complex algorithms. Achieving parallelism, in an optimal way, avoiding waiting synchronization mechanism such as mutexes requires a deep understanding of concurrency and parallelism techniques, that will be covered in this thesis.

1.1 Aim and objectives of the study

The aim of this master's thesis is to develop a methodology that could be used to evolve sequential into parallel algorithms in a reusable way, in order that the applied techniques could be extrapolated to other algorithms obtaining consistent performance improvements. This methodology focuses on complex and data-intensive algorithms in the field of computer graphics algorithms.

The primary objective of the thesis was to assess the proposed methodology of evolving sequential into parallel algorithms. In order to test the proposed methodology two common computer graphics algorithms were taken as case of study: hidden-line removal and collision detection.

In order to achieve the set objective of the study, the following work was carried:

1. Defining a methodology to support the evolution of algorithms from a sequential to a parallel implementation.
2. Implementing a multi-thread version of hidden-line removal and collision detection.
3. Evaluating the performance of the new parallel implementations.
4. Assessing the proposed methodology.

The **structure of this thesis** is divided into the following sections.

- **Chapter 1** is the introduction.
- **Chapter 2** introduces the theoretical background. In this chapter, there is a detailed description of hidden-line removal and collision detection. Cadmatic is also introduced in this section as the environment in which these two algorithms are executed and analyzed.
- **Chapter 3** describes the methodology and parallelization techniques followed in order to evolve sequential algorithms into parallel ones. It also explains how these techniques could be extrapolated to other algorithms. Moreover, Cadmatic hidden-line removal and collision detection algorithms are transformed to their parallel implementations.
- **Chapter 4** describes the obtained performance results in the two study cases of hidden-line removal and collision detection.
- **Chapter 5** discusses the obtained results, making an interpretation. The validity of the proposed parallelization framework is also discussed.
- **Chapter 6** corresponds to the conclusions of this master's thesis, and future work that needs to be considered.

2 Theoretical Background

2.1 Cadmatic

Cadmatic is a 3D design tool [10] for the marine and plant industries. Cadmatic helps designers and engineers through the whole shipbuilding phases: initial design, preliminary design, basic design, and detailed design. In addition to that, Cadmatic is used in other later stages such as operation and maintenance, giving ship production information and support.

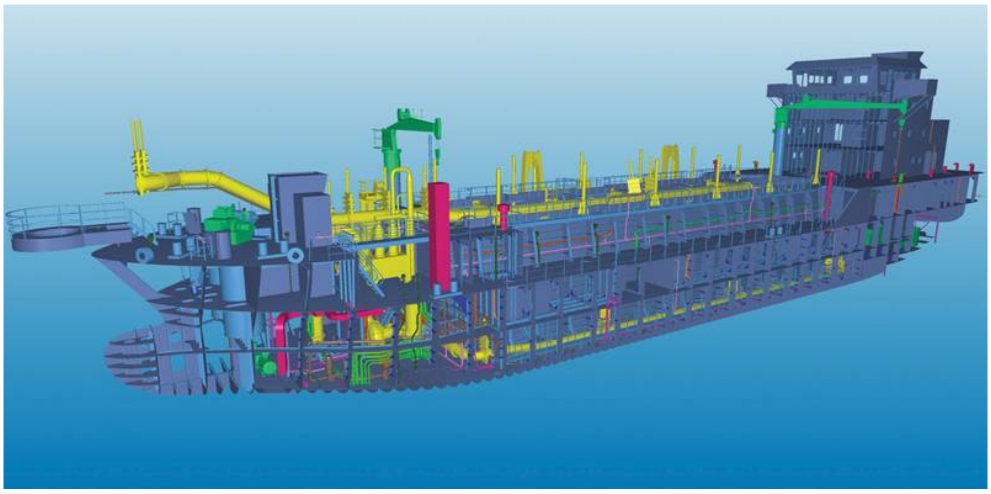


Figure 2.1: Ship designed using Cadmatic.

Cadmatic also possess a plant design software aimed to be used by engineers and designers to create 3D models of complete industrial projects. It provides outfitting functionalities such as piping, instrumentation diagrams and generation of isometric drawings.

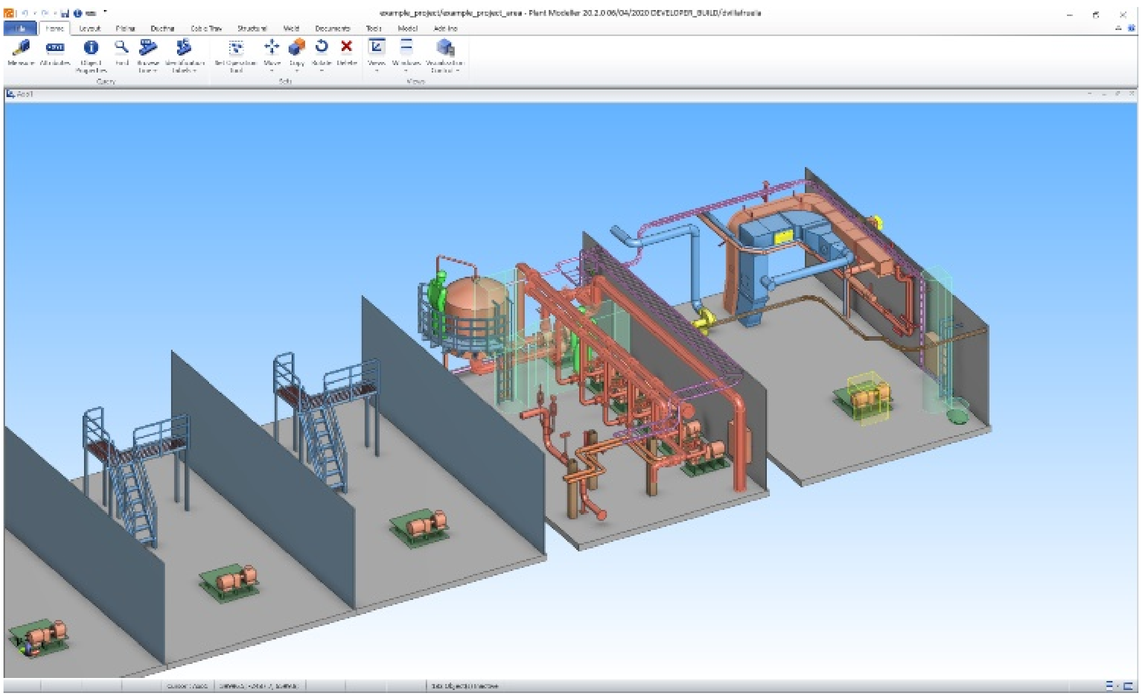


Figure 2.2: Plant industry designed using Cadmatic.

Performance and scalability are important factors for Cadmatic in order to satisfy growing needs and demands of users and customers. Most Cadmatic algorithms are memory efficient, specially optimized to reduce their memory usage. Cadmatic software was created in the 80s when memory was a scarce resource in many computers. Nowadays, memory is an abundant resource and it is possible to increase memory consumption in order to reduce the amount of computation by using memorization techniques or dynamic programming [11].

Cadmatic software is written mostly in C and C++. These programming languages generate machine code, providing less overhead than other programming languages such as Python or Java.

Most parts of Cadmatic software can only run on a single processing element. Cadmatic software was designed in the 80s when concurrency did not exist or was not important because most computers had a CPU with only a single processing element (core in a multi-core processor). Nowadays most computers are conformed by several different kinds or same kind but redundant processing elements [7]. It is not unusual to find several processing elements in a CPU and a graphical processing unit. This brings new possibilities to Cadmatic to boost performance of some complex and data-intensive algorithms, such as hidden line removal and collision detection, whose sequential execution times are taking considerable long times.

Hidden-line removal [12] is the method of computing which edges are visible by the faces of parts for a specified view. Collision detection [13] is the computational problem of detecting the intersection of two or more objects.

Using a GPU to accelerate those algorithms seems to be the right option, but GPU software integration on Cadmatic is not an easy task. It is worth of mentioning that Cadmatic uses OpenGL API to render 2D and 3D vector graphics. Using GPU programming frameworks such as OpenCL or CUDA could affect the rendering performance of Cadmatic software because of memory context switches between different kernels, being a real issue that needs to be studied. Moreover, as it was mentioned previously, hidden-line removal and collision detection algorithms have data dependencies and include third party software that can not be loaded directly into a GPU.

Multi-threading, using several processing elements of a CPU, is the approach followed in this thesis in order to define the methodology of evolving sequential to parallel algorithms. It is difficult to compare performance obtained by applying parallelism using multi-threading or a GPU, due to the fact that both are application specific. Even though a CPU has less parallelization capabilities than a GPU, a CPU could provide a better solution for coarse-grain granularities. In addition to that, multi-threading can be easily applied to an existing sequential algorithm without many modifications.

2.2 Multi-threading

Most modern CPUs have several processing units, exploiting parallelism across threads to improve performance [14]. A thread contains a set of instructions that are executed in a processing element in sequence. In modern operating systems, there is almost no limitation to the number of threads that can be created. However, the number of threads that can be concurrently executed at a specific moment is determined by the hardware. It is not possible to have more active threads than processing elements. In addition to that, there are other factors that determine the performance of multi-threading [15] such as the thread granularity, the workload distribution among threads, and the number and cost of context switches. **Thread granularity** is determined by the number of instructions per thread. Three are three basic kinds of granularities: fine-grain (tens of instructions), medium-grain (hundreds or more), and coarse-grain threads (thousands or more). The **workload distribution** [15] makes reference to how data is distributed among each thread. It is not a good performance solution to have threads that are doing most of the work whereas other threads are already finished or idle, being their processing elements free. This problem is present specially in algorithms in which there is big processing time asymmetries between different elements, e.g., fault simulation algorithms [16]. In addition to that, using threads can yield bad performance results when threads are interrupted, having to store their state for resuming their execution later on. These **context switches** are often caused by other threads or processes that are running concurrently [17]. However, when a thread needs to access a shared resource in an exclusive way, it needs to lock it, blocking other threads to access it. A blocked thread has to do a context switch, and it will resume its execution when the shared resource is again available. Reducing data dependencies is an essential task that needs to be carried out when parallelizing an algorithm in order to obtain satisfactory performance results.

The standard C++ multi-threading library is used in this thesis. This standard library contains most of the features that are available in other libraries such as Portable Operating System Interface (POSIX) Threads or p-threads libraries.

2.3 Hidden-line removal

Hidden-line removal is a computer graphics algorithm [18] used to determine which object parts (line segments) in a wire-frame view (includes lines and vertices) are not visible when occluded by other objects. Wire-frame views are important in order to create 2D drawings. Even though most of the design is done with 3D models, 2D drawings are required for manufacturing. Manufacturers produce designed components by taking these 2D drawings as reference.

In a three-dimensional computer graphics image display, it is often desirable [19] to remove lines which are obscured or hidden from a viewer by an object which is closer to the surface of the display screen. Hidden-line removal algorithm is specially important in engineering when working with drawings.

Figure 2.3 and 2.4 show a Cadmatic drawing illustrating the behavior of hidden-line removal:

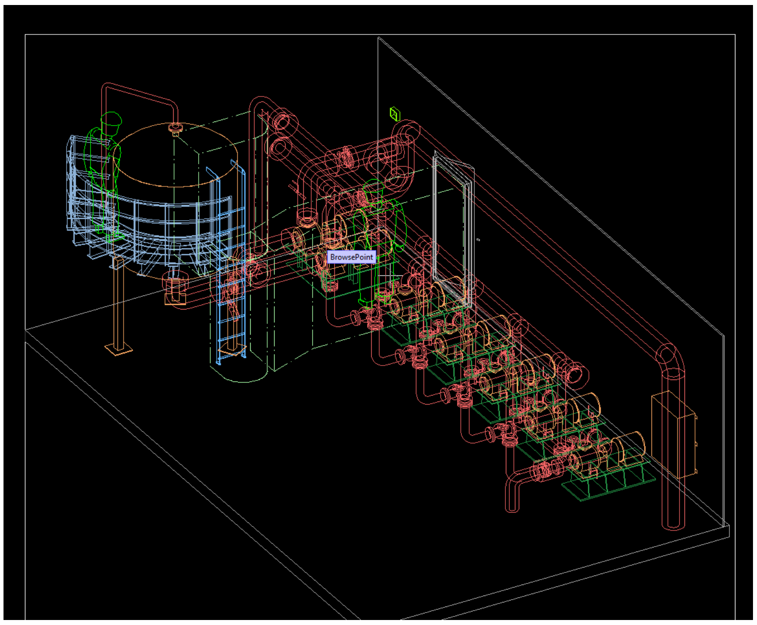


Figure 2.3: The first image represents a set of plant model objects contained in a view in which the hidden lines have not been removed.

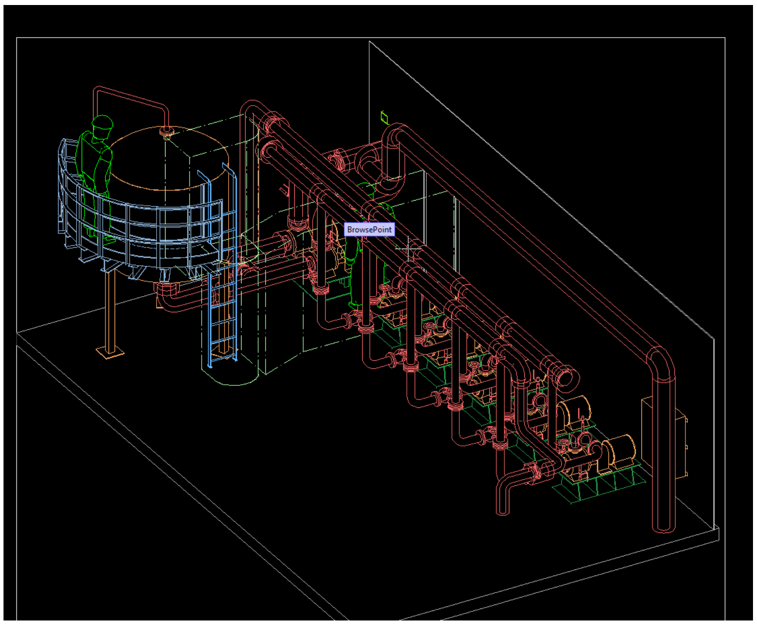


Figure 2.4: The second image represents the same view than the first image, after applying hidden-line removal algorithm.

2.3.1 Existing sequential algorithms

Hidden-line removal was an important research area in the 1970-1980s. In this decade, many hidden-line removal algorithms were developed. Every year a new algorithm was published improving the previous one. This iteration culminated with the best sequential algorithm published by Dévai in 1986 [20]. The first group of algorithms published before 1984 were based on the idea of dividing edges into line segments by the intersection points of their images taking $O(n^3)$ in the worst-case scenario.

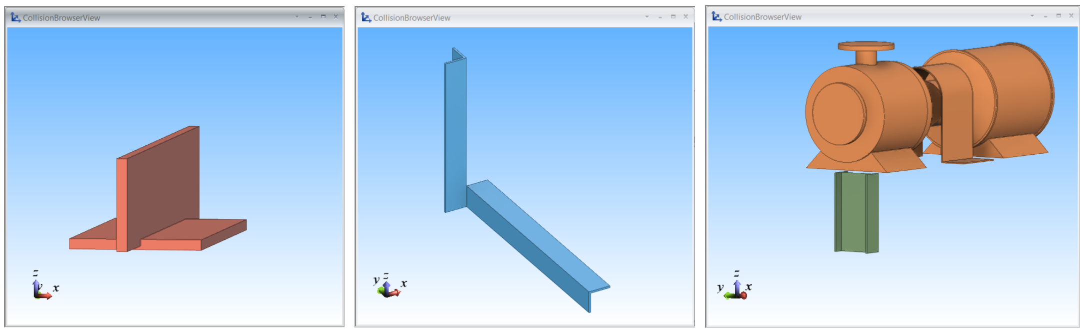
Appel was the first researcher to give a general and easy solution for the problem of hidden-line removal [21]. However, Appel's solution was very inefficient resulting in very long running times caused by applying a brute force approach, since all lines of a surface are quantized into points and each point is tested independently to check whether or not it is hidden by other parts of that previous surface [19]. Galiimberti and Montanari improved Appel's algorithm by computing only the set of faces which hide an edge point, discarding the rest of them [12]. Hornung, combined Appel's idea of contour points with the notion of quantitative invisibility, to compute by intersection, only the points in which there are changes in the visibility, testing edge intersection of all edges against only all contour edges [22].

2.3.2 Parallel algorithms

In the early 80s there were not much research done about parallel algorithms for hidden-line removal. In those times in which these sequential hidden-line removal algorithms were developed, parallelism was not a priority because most personal computers had only a single processing element. Sequential hidden-line removal algorithms previously mentioned, are difficult to parallelize due to their complexity and dependencies [23]. Dévai was the first one to propose a parallel algorithm for hidden-line removal whose time complexity is $O(log(n))$ when n^2 processors are used. Although, the algorithm was not optimal, he proved that the hidden-line removal problem can be solved in polylogarithmic time on a parallel computer with a polynomial number of processors. Reif and Sandeep Sen implemented a new algorithm for hidden-surface removal in 1988 modifying the algorithm to work for hidden-line removal [24]. Hidden-surface removal is an algorithm used to determine which surfaces should not be visible to the user. In the hidden-surface removal surfaces are the main point of concern, however in hidden-line removal, visibility of the edges is the focus of interest. The parallel version of the algorithm has a time complexity of $O(log^4(n+k))$ when using $O((n+k)/log(n))$ Concurrent Read Exclusive Write (CREW) Parallel Random-Access Machine (PRAM) processors, being n the input size, and being k the total number of the intersection points. In 2011, Dévai parallelized the already optimal sequential algorithm that he developed back in the 80s [20], being this parallel algorithm work-optimal, having a time-processor product of $O(n^2)$. Having a time complexity of $O(log(n))$ when using $n^2/log(n)$ CREW PRAM processors, these kind of processors can read concurrently the same memory address but only one can modify it at a specific time.

2.4 Collision detection in computer graphics

Collision detection is the problem of determining whether two given objects defined by their own geometry (meshes) intersect or not with each other. One of the main problems in 3D design is to detect collisions, contacts and clearance violations between objects inside a model e.g. cruise ship. A collision between two objects is produced when they intersect with each other. A contact is produced when both objects are touching each other but without intersecting. Clearance violations are produced when the distance between two objects is less than a specific user-defined value.



(a) collision (b) contact (c) clearance violation(10 mm)

Figure 2.5: Examples of a collision, contact and clearance violation between two different objects.

In computer graphics, collision detection has been an interesting field of research for many years [13]. It has been used in many diverse areas such as computer graphics, computer games, computer simulations, robotics and computational physics [25].

Big engineering projects, such as an ultra complex model of a cruise ship are typically composed of thousands or millions of 3D objects. Collision detection is becoming a bottleneck due to increasingly complex geometries and amount of object in which collisions needs to be determined [26]. Performance of collision detection could be improved by evolving the algorithm to a parallel version. It is important to mention that it is complicated to evaluate and compare collision detection algorithms [27]. They are sensitive to factors such as the complexity of the objects, the positions where they are situated to each other, the distance, etc.

The naive and simplest approach to detect collisions between two sets of objects is to compare all primitives of each object against all primitives of the rest of the objects. It is important to mention that objects are composed by simple shapes called primitives. These common primitives can be 0-dimensional (points), 1-dimensional (lines and curves), 2-dimensional (polygons and triangles) and 3-dimensional (cube, cylinder, sphere, cone, pyramid and torus). Directly testing collisions between two objects by considering their geometries is often computationally expensive, due to the fact that objects can be composed of hundreds or even thousands of primitives (e.g. polygons). In order to achieve an efficient collision detection it is necessary to carry out some fast intersection tests between pairs of objects to discard collisions such as bounding volumes or other basic primitive tests.

2.4.1 Bounding volumes

Object bounding volumes offer a great opportunity to minimize the cost of calculating collisions between two objects. A bounding volume [28] is a single simple shape encapsulating one or more 3D objects whose geometries are more complex than the bounding volume. Testing collisions between these simple bounding volumes such as boxes or spheres is less computationally expensive than testing in detail their encapsulated complex objects. Bounding volumes rapidly allow to discard objects that do not overlap, avoiding carrying out more complex, detailed and computationally expensive collision tests.

When objects are overlapping and their bounding volumes are intersecting, this additional test results in an increase in computational time [13]. However, in most situations compared objects are not close enough to overlap, and consequently using bounding volumes usually results in a remarkable performance boost.

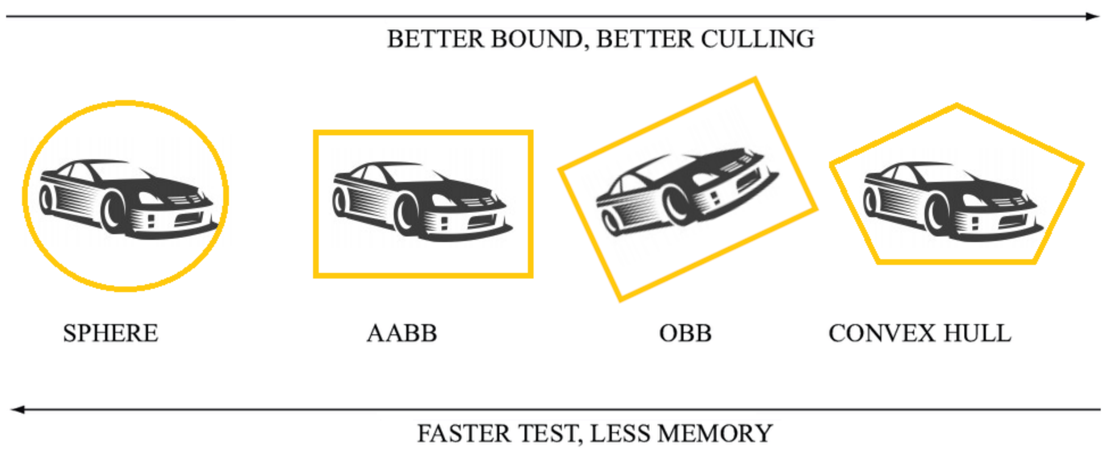


Figure 2.6: Most common types of bounding volumes

Bounding volumes are typically computed in a pre-processing step rather than at run time [28], not causing performance issues.

2.4.2 Basic primitive tests

In the case that two objects' bounding boxes are overlapping it is necessary to carry out more computationally expensive tests, in order to check that their geometries are intercepting.

2.4.2.1 Closest-point computations

The closest points between two objects is obtained and used in order to calculate the distance between both objects. If this minimum distance between two objects is less than the combined maximum movement of both objects, then a collision can be discarded [28]. A hierarchical representation can be created using these closest-point computations in order to prune parts of the hierarchy that will never come close enough to collide.

2.4.2.2 Testing Primitives

Checking collisions between two objects by testing theirs primitives is more specific than the computation of distance between both objects. These primitive tests will only indicate that both objects' primitives are intersecting without entering into details about where or the way that they are intersecting [28], and hence these intersection tests are considerably faster than other tests that provide extra information.

Separating-axis Test This collision test is based on the hyperplane separation theorem.

Theorem 1. *Given two convex sets A and B, either the two sets are intersecting or there exists a separating hyperplane P such that A is on one side of P and B is on the other.*

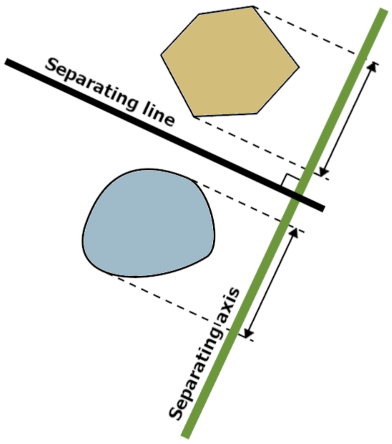


Figure 2.7: Illustration of the hyperplane separation theorem

Thus, two non-intersecting objects always have a gap between them, in which a plane that separates both objects can be placed.

Intersecting Lines, Rays, and (Directed) Segments

These kind of tests involve lines, rays or segments. They are cast from one object and it is checked if they hit the other object. Using different directions and senses it is possible to know if the object is colliding or not.

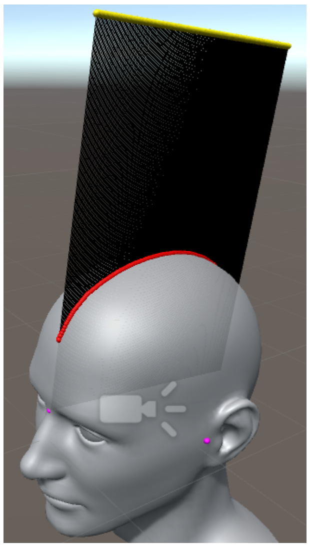


Figure 2.8: Ray casting example.

Additional Tests

There are some additional low level geometry tests:

- Testing Point in Polygon.
- Testing Point in Triangle.
- Testing Point in Polyhedron.
- Intersection of Two Planes.
- Intersection of Three Planes.

The previous image was created by the author of this thesis to illustrate how ray casting works in a visual way.

2.4.3 Bounding Volume Hierarchies

As it was explained previously, wrapping objects in bounding volumes and testing collisions was faster than testing geometries of both objects. Even though, it is considerable faster, the asymptotic time remains the same. By grouping the bounding volumes into a tree hierarchy known as Bounding Volume Hierarchy (BVH), the time complexity can be improved to logarithmic [28] considering the total number of tests performed.

In a bounding volume hierarchy [28], the leafs of the tree are formed by the original objects. Larger bounding volumes are created on top by grouping and enclosing previous small sets, in a recursive fashion. At the end, it results in a tree structure with a single bounding volume at the root node.

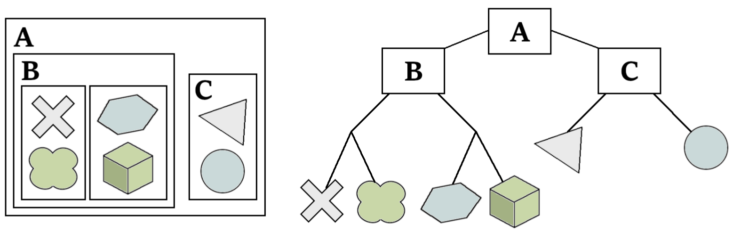


Figure 2.9: An example of a bounding volume hierarchy using rectangles as bounding volumes.

2.4.4 Spatial partitioning

Spatial partitioning techniques are used to divide an Euclidean space into two or more non-overlapping regions [28], testing if objects overlap a same region in the partitioned space. The number of pairwise tests decreases because two objects can only collide if they are in the same region.

In this section only Octrees/Quadtrees are explained. Other spatial partitioning techniques [28] such as uniforms and hierarchical grids,... are not considered.

2.4.4.1 Octrees and Quadtrees

An octree and a quadtrees are tree data structures with eight and four children respectively [28]. Octrees are used to partition a three-dimensional region by simultaneously dividing the cube in half along each of the x, y, and z axes. On the other hand, quadtrees subdivide a rectangle or square along the x and y axes. Child nodes are recursively subdivided in the same way. Typically, the criteria for stopping the recursion is when the tree reaches a maximum depth or the size of each subdivision (e.g cubes in the case of an octree) is sufficiently small or a user-defined threshold is reached.

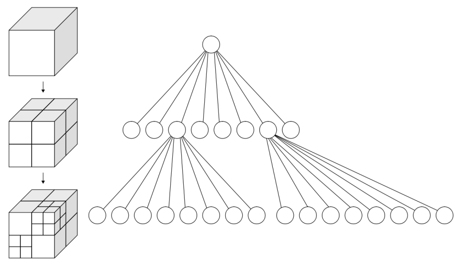


Figure 2.10: Recursive subdivision of a cube into octants and its octree.
Source: `https://en.wikipedia.org/wiki/Octree`

3 Methodology

To obtain the results required in the thesis, it is necessary to define the principles and techniques that are going to be used to evolve sequential code into parallel code in the context of complex and data-intensive algorithms. In addition to the environment and the cases of studies in which these "methodologies" will be tested.

For the purpose of this thesis, it was decided to use Cadmatic software as the environment in which the parallelization will be applied, and the Cadmatic hidden-line removal and collision detection as the two cases of study. Due to the fact that Cadmatic software is written in C/C++, most of the examples and code snippets that are written in this thesis are also written in C/C++. Pseudo-code is also used to illustrate some general ideas. The hardware acceleration technique used in this thesis is multi-threading. Running code in a GPU is not a reasonable option due to the software modules that are used and the existence of dependencies in the code.

The first part of this section consists of defining the "framework" containing the principles, techniques and ideas used in order to evolve sequential code into parallel code. This part contains two different patterns about how to make the code thread-safe and how to improve the performance of the existing multi-thread code.

After the parallelization methodologies are defined, it is time to apply them to two Cadmatic data-intensive algorithms (hidden-line removal and collision detection) to test how performance is improved. Both algorithms are frequently used by Cadmatic users. The sequential implementation of hidden-line removal and collision detection are explained and compared against their multi-thread versions. Parallelization issues such as asymmetries of their inputs and solutions are explained in detail.

The data generated by new multi-thread versions of hidden-line removal and collision detection is recorded in order to check that matches the older versions of both algorithms. At the same time, information about execution times is recorded in order to be able of comparing the performance results and speedup. These results could be used to evaluate the impact and effect of the suggested parallelization principles and techniques.

3.1 Parallelization techniques and patterns

This section describes common principles and techniques that can be applied to a wide range of data-intensive sequential algorithms in order to automatize their parallelization, obtaining common ranges of performance improvements.

These generic sequential-parallel conversion patterns provide consistent performance improvement results that can be easily applied to most kinds of sequential algorithms in order to obtain their multi-thread versions, yielding acceptable optimization results.

This section is divided into two parts:

1. **Thread-safe patterns:** how to make the code thread-safe, so it can be run concurrently without causing race conditions, deadlocks, livelocks ...
2. **Optimization patterns:** it describes patterns that try to improve the speedup of the parallelization of an algorithm.

3.1.1 Thread-safe patterns

This section describes steps and patterns used to make thread-safe code.

Identifying the critical sections: The first and one of the most important steps to follow in order to generate thread-safe code is to identify the number of critical sections and the places in the code where they are located. It is important to notice that complex algorithms such as hidden-line removal or collision detection can become a difficult task.

A critical section or critical region is a shared resource or part of a program that is accessed concurrently by several threads [29]. Critical sections need to be protected in order to avoid race conditions. Race conditions [30] occurs when two or more threads access shared memory without proper synchronization, causing the program to behave unexpectedly. Identifying these critical sections is an essential task. Otherwise, the parallelization of a sequential algorithm can not be obtained.

Spotting critical sections is not a trivial task, especially if an algorithm is using several software modules. For this reason, it is indispensable to pay attention if the *const keyword* is present when functions and methods are called in the algorithm. The *const keyword* guarantees that the contents of variables are not modified inside that function. It is worth spending time on adding *const keywords* to functions, methods and variables that actually do not modify that variable.

It is possible to rapidly find parts of the code that are thread-safe by paying attention to the *const keyword*. This is immensely useful when working with complex algorithms because these algorithms have many calls in the call stack.

Observing the diagram in figure 3.1, it is possible to rapidly discard parts of the code in which a variable is not modified, considering the *const keyword*.

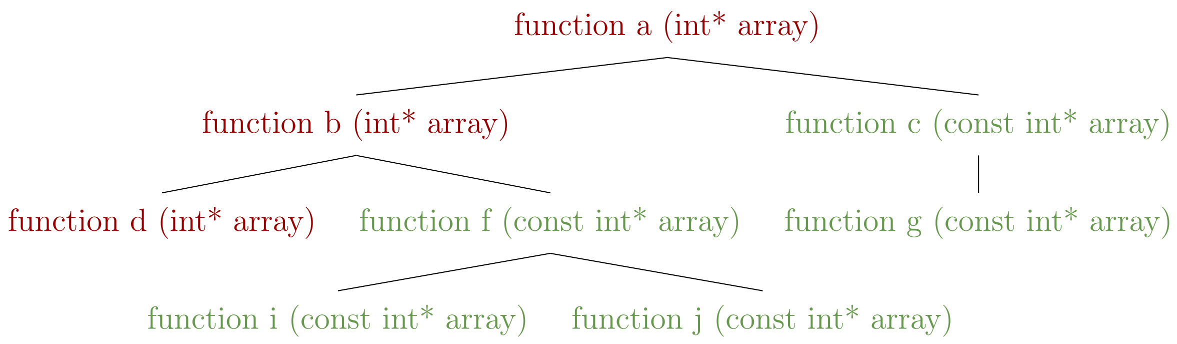

function a (int* array)
function b (int* array)
function c (const int* array)
function d (int* array)
function f (const int* array)
function g (const int* array)
function i (const int* array)
function j (const int* array)

Figure 3.1: Example of the importance of the *const keyword*

Looking at the tree generated from an algorithm call stack, it is possible to rapidly observe where the critical sections might be located. Notice that this method does not guarantee that global and static variables are not accessed in those functions.

Once all critical sections are identified, it is the time of getting rid of them or making them thread-safe.

Converting global variables to local variables: Replacing global and static variables (accessed by several threads) by local variables (accessed by each thread independently) eliminates many of the conflicts existing between threads.

This process should be relatively easy to do when the variables are not accessed directly by threads. However, there are situations in which each thread needs to access these variables independently from other threads. It is necessary to provide a context to each thread containing the variables that are going to be accessed during the lifespan of a thread.

Encapsulating thread-access variables into a context structure: Keeping track of all variables that are modified by a single thread is a complicated task. These variables are local-thread variables or shared-thread variables. These local-thread variables are only accessible by the current thread and other threads can not access those. The shared variables are variables commonly shared with other threads and they require access by mutual exclusion in order to avoid conflicts.

Algoritm 1 Encapsulating thread-access variables into a context structure

```
struct {
    struct {
        // only variables accessible by the current thread
        ...
        thread_local_var;
    } local_variables;
    struct {
        // accessible by all threads
        ...
        mutex; // used to get mutual exclusion
    } shared_variables;
} ThreadContext;

ThreadContext threadContext1;
ThreadContext threadContext2;

// Initializes contexts with the local and shared variables
ThreadContext set_local_values(threadContext1, threadContext2);
ThreadContext set_shared_values(threadContext1, threadContext2);

Thread thread1(threadContext1);
Thread thread1(threadContext2);
```

Pseudo-code 1 illustrates how variables used in an algorithm can be encapsulated in order to be used by several threads simultaneously.

Combining thread-results: Frequently, threads require accessing variables that are modified concurrently by several threads. In these kinds of situations, it is possible to avoid accessing the *result* critical section by using thread-local variables to store each thread partial result, combining the results afterward. These partial thread-results are stored in a common array accessible by each thread during its execution. Each thread uses its own thread ID as an index in that array.

Consider a problem of determining the minimum bounding box enclosing different objects. Each object requires to update the resulting bounding box if the current object's bounding box is not contained in it.

```
int get_minimum_bounding_box(const std::vector<Object> &objects, Mbb &result)
{
    if (objects.empty())
        return -1;

    result = objects[0].mbb;// minimum bounding box

    for (size_t i = 1; i < objects.size(); ++i)
        mbb_combine(result, objects[i].mbb);

    return 0;
}
```

The sequential code mentioned above can be transformed by creating a thread-local result for each thread. These partial results can be later combined obtaining the final result.

```
int get_minimum_bounding_box(const std::vector<Object> &objects, Mbb &result)
{
    if (objects.empty())
        return -1;

    const size_t processor_count = std::max(std::std::thread::hardware_concurrency(), 1);
    size_t num_threads = std::min(objects.size(), processor_count);

    ThreadPool threadPool(num_threads);
    std::vector<Mbb> threadResults(num_threads);

    // Each thread calculates its own partial result.
    threadPool->parallelFor([&objects, threadResults](std::size_t fromIndex, std::size_t toIndex,
      std::size_t threadIndex) {

        Mbb &thread_result = results[threadIndex];
        thread_result = objects[i].mbb;

        for (size_t i = fromIndex + 1; i < toIndex; ++i) {
            mbb_combine(thread_result, objects[i].mbb);
        }
    }, objects.size());


    // Combining the results obtained by each thread
    result = objects[0].mbb;
    for (const Mbb &thread_result : threadResults)
        mbb_combine(result, thread_result);

    return 0;
}
```

Mutex - Mutual exclusion: The easiest approach to achieve mutual exclusion of a critical section is to use a mutex. A mutex is a mutual exclusion object that synchronizes access to a critical region in a way that only one thread can be in the critical section. It is important to mention that when a thread locks a critical section, the rest of the threads will have to wait to access that resource. For that reason, minimizing the number of critical sections and the time spent on each critical section is essential to achieve suitable optimization results when transforming a sequential algorithm to a multi-thread version.

Algoritm 2 Example of a Mutex

```
std::mutex mtx;

mtx.lock();
// Access to the critical section
mtx.unlock();
```

Notice that C++ implement wrappers for owning a mutex for the duration of a scoped block. This code is the same as above:

Algoritm 3 Example of a Mutex

```
std::mutex mtx;
{

   std::lock_guard<std::mutex> lock(mtx); // Locks
   // Access to the critical section

} // unlocks when object is destroyed
```

Pre-computation: There are situations in which critical sections are instantiated only once during the execution of an algorithm. For instance, some geometry calculations need to be computed only once during the whole execution of an algorithm. The instantiation of these critical sections is done lazily, on-demand, at any point during its execution. In some situations, they even might not be instantiated at all if they are not needed. Consider the following example in which elements firstly need to be computed and their results stored in an array data structure:

```
double get_result(int index) {
    if (!Cache) { /* critical section */
        Cache = (double *) malloc(sizeof(double) * CACHE_MAX_SIZE)
        for(size_t i = 0; i < CACHE_MAX_SIZE; ++i) {
            mem[i] = compute_result(i);
        }
    }

    double value = cache[index];
    double result = do_stuff(value)     /* some operations */

    return result;
}
```

If several threads instantiate the *cache* data structure at the same time it would produce race conditions. In order to avoid race conditions, a pre-computation technique could be followed, avoiding using mutexes. In this technique, elements inside the critical sections are pre-computed before starting the execution of the algorithm.

```
init_cache(Cache); /* early initialization avoids race  conditions */

threadPool->parallelFor([&Cache](std::size_t fromIndex, std::size_t toIndex, std::size_t
     threadIndex) {
    for (size_t i = fromIndex; i < toIndex; ++i) {
        element = elements[i];

        get_result(element); /* original function in the critical section is instantiated if it
     is null */
    }
}, CACHE_MAX_SIZE);
```

It is important to mention that race conditions can not appear afterward because instantiations and initializations were already carried out at the beginning.

Avoid dynamic memory reallocation: Inserting elements into a dynamic data structure is not a thread-safe operation due to the fact a dynamic data structure might reallocate memory in order to shrink or grow. This is problematic in the case that several threads are modifying the same dynamic data structure.

The easiest solution to this problem is to allocate the necessary memory beforehand, so threads can modify elements into that dynamic structure knowing that it is not going to grow or shrink.

```
size_t n_elements = 50;
std::vector<int> elements(n_elements);

std::atomic_int atomic_index(-1);
threadPool->parallelFor([&elements, &n_elements, &atomic_index](std::size_t threadIndex) {
    int local_index;
    while ((local_index = ++atomic_index) < n_elements) {
        elements[local_index] = threadIndex;
        /* ... */
    }
}, n_elements);
```

Recursive Mutexes: A recursive mutex [31] is a particular type of mutual exclusion object that allows the same thread to lock the same mutex several times without producing a deadlock. The same thread also needs to unlock the mutex the same number of times that it was locked. Otherwise, no other thread can obtain the mutex.

There are some situations in which using a recursive mutex is significantly useful.

Consider a situation in which there is an algorithm that is using several functions that need to be executed on mutual exclusion and at the same time they have conditional references or calls to each other:

Algoritm 4 Example of a Recursive Mutex

```
function1(ThreadContext &context)
{
   context.mtx.lock();
   // Access to the critical section
   context.mtx.unlock();
}

function2(ThreadContext &context)
{
   context.mtx.lock();
   // critical section
   ...
   function1(contex); // refers to function 1
   ...
   // end - critical section
   context.mtx.unlock();
}
```

Using a recursive mutex solves the problem. Typically, a recursive mutex has a counter that is incremented every time the mutex is locked. The counter is decremented every time the mutex is released. The thread that acquires the mutex must be the same one that releases it. No deadlock is produced.

3.1.2 Optimization patterns

This section describes techniques that could be applied in order to obtain better optimization results when converting a sequential algorithm to a parallel one.

Thread pool: A thread pool is a common and reusable solution for achieving concurrency of execution [32]. A thread pool consists of a fixed number of threads, usually the same number as the number of cores, waiting for jobs to be assigned to them for being concurrently executed. Threads are light-weight processes that are able of executing different jobs in parallel. The main difference between a thread and a process is that a thread shares a common address space with other threads whereas a process has its own address space.

Creating threads dynamically is a slow process [29] that requires allocating memory within the process (address space) in order to allocate stacks for these newly created threads. In addition to that, the creation of a thread introduces latency to the execution of a task because of the extra time required to create a thread. By creating a thread pool, whose threads are reused constantly for different tasks, performance is improved and all latency issues associated with the creation of threads are eliminated.

The number of threads in a pool of threads is an important factor to take into consideration. In the case of using a fixed amount of threads, it is convenient to set it to the number of CPU cores. Using the standard thread library of C++, it is possible to obtain the maximum number of concurrent threads supported, typically the number of cores by using ***std::thread::hardware_ concurrency***.

It is convenient to check that the number of cores is not zero when using *std::thread::hardware_ concurrency*. Otherwise, some data structures would not be initialized correctly.

A thread pool can dynamically alter the number of threads during the course of a program based on the number of waiting jobs.

Having an excessive number of threads increases the resource usage of an application, and not having enough introduces some latency issues. If the thread pool creates too many threads, and they are active, then it can create performance issues because of context switches. A context switch stores the state of a thread, so it can be later restored and resume execution. Setting priorities to different threads helps to manage this situation.

If threads are inactive, destroying them helps to avoid wasting computer resources. However, destroying too many threads requires more time later when creating them again. On the other hand, if there are not enough threads, it increases long wait times and latency to tasks that are waiting to be executed.

In this case of study of this thesis, only thread pools with a fixed amount of threads matching the number of cores are used.

Workload Distribution between Threads: Matching data distribution to workload distribution is an important process in order to improve the performance of distributed-memory multiprocessors [14]. Every thread taking part in the parallelization of an algorithm should be assigned a workload similar to other threads. Otherwise, some threads will finish their tasks much sooner than others, causing performance imbalances.

Asymmetries of the work distribution between different threads can produce inefficient optimization results, even worse than the non-parallel solution. For this reason, achieving an equal workload between threads is an essential task.

Equal Indices-Range Distribution of Data This approach is used to assign equal workload between different threads by splitting the number of elements into ranges of indices and assign each unique range to a thread.

Each thread can use its own range of indices to access elements from the data structure in which elements are stored. Each element is only accessed by one thread using its given range of indices, it is impossible to have race conditions between threads.

The code below represents how a pool of threads uses this approach in order to split the elements of an array among different threads:

```cpp
threadPool->parallelFor([&elements](std::size_t fromIndex, std::size_t toIndex, std::size_t
    threadIndex) {
    for (size_t i = fromIndex; i < toIndex; ++i) {
        auto element = elements[i];

        execute(element);
    }
}, n_elements);
```

Notice that the *parallelFor* method receives a lambda function as an argument. This lambda function is given to the threads of the thread pool.

Even though each thread computes the same number of elements, the execution time required by each thread is different because of data asymmetry between different elements.

```
Thread 6 -> Elapsed time:  0.234790s

Thread 4 -> Elapsed time:  0.735534s

Thread 5 -> Elapsed time:  1.449923s

Thread 3 -> Elapsed time:  2.011944s

Thread 2 -> Elapsed time:  4.654341s

Thread 1 -> Elapsed time:  7.526702s

Thread 0 -> Elapsed time: 19.264020s

Total Elapsed time: 22.632818s
```

It is important to notice that this approach does not perform well when there is an asymmetry in the time required to process different elements.

The reason why Thread 0 requires more time than the other threads is because this thread computed the most complex element. This is a recurrent problem in many complex computer-intensive algorithms in which each element has different complexity.

In order to get rid of this problem, it is necessary to carry out a more competitive approach.

Competitive Work-Load Distribution of Data: In a complex data-intensive algorithm, each element has different complexities and sizes. One element can be easily computed whereas another one can take several times longer.

In a competitive work-load distribution each thread is competing against each other for the next index in the array of the element to compute:

```cpp
std::atomic_int atomic_index(-1);
threadPool->parallelFor([&elements, &n_elements, &atomic_index](std::size_t threadIndex) {
    int local_index;
    while ((local_index = ++atomic_index) < n_elements) {
        auto element = elements[local_index];

        execute(element);
    }
}, n_elements);
```

Notice that an atomic variable is used to store the last index of the element that was assigned to a thread that won the competition because this thread was available and it was the first to increment the atomic variables. Accordingly to C++ language documentation, atomic types are types that encapsulate a value whose access is guaranteed to not cause data races and can be used to synchronize memory accesses among different threads. It is important to mention that using atomic variables instead of mutexes accomplishes better performance results and less overload.

As it can be observed in the following figure, the obtained load-work distribution is almost symmetric:

```
Thread 3 -> Elapsed time:  6.779115s

Thread 2 -> Elapsed time:  6.779133s

Thread 5 -> Elapsed time:  6.779178s

Thread 1 -> Elapsed time:  6.779193s

Thread 0 -> Elapsed time:  6.779231s

Thread 4 -> Elapsed time:  6.870100s

Thread 6 -> Elapsed time:  8.937288s

Total Elapsed time: 9.223558s
```

Avoiding busy-waiting or time-wait statements: In busy-waiting, busy-looping or spinning a thread is constantly checking that a specified condition is true. Consider the typical producer-consumer problem in which busy-waiting is applied:

```

#define MAX_ITEMS 1000

static void producer(ThreadData &commonData)
{
    int local_index;
    while ((local_index = ++commonData.atomic_input_index) < commonData.inputs.size()) {
        if (commonData.item_indices.size() >= MAX_ITEMS) {
            wait(10); // waits 10 ms if number of items are more than max
        }

        const Input &input = commonData.inputs[local_index];
        commonData.items[local_index] = create_item(input); // Produces an item

        commonData.queue_indeces_mtx.lock();
        commonData.item_indices.push(item_index);
        commonData.queue_indeces_mtx.unlock();
    }
}

static void consumer(ThreadData &commonData)
{
    size_t total_process_data_num = inputs.size();

    while(commonData.atomic_processed_data_num < total_process_data_num) {
        while (item_indices.empty()) {
            // Busy waiting: until there is a new created item
        }

        // Gets produced item
        commonData.queue_indeces_mtx.lock();
        int item_index = commonData.item_indices.front();
        commonData.item_indices.pop();
        commonData.queue_indeces_mtx.unlock();

        // Consumes an item
        Item &item = commonData.items[item_index]
        consume_item(item);
        ++commonData.atomic_processed_data_num;
    }
}
```

By using *condition variables*, it is possible to block a thread (or several threads) until another thread modifies a condition variable, notifying the blocked threads to resume their execution. In this case, two condition variables can be used to block producers in order to stop producing when a maximum value is reached. Readers can be blocked if there are no items to consume.

```
#define MAX_ITEMS 1000

static void producer(ThreadData &commonData)
{
    int local_index;
    while ((local_index = commonData.atomic_input_index++) < commonData.inputs.size()) {
        std::unique_lock<std::mutex> queue_lock(commonData.queue_indeces_mtx);
        {
        // Block producers if they have reach a threshold
        commonData.consumed_item_condition.wait(queue_lock, [&commonData] {
                return commonData.item_indices.size() < MAX_ITEMS
                        || commonData.completed_consumer_num == commonData.consumer_num;
        });}  // automatically releases lock

        const Input &input = commonData.inputs[local_index];
        commonData.items[local_index] = create_item(input); // Produces an item

        commonData.queue_indeces_mtx.lock();
        commonData.item_indices.push(item_index);
        commonData.queue_indeces_mtx.unlock();

        commonData.added_item_condition.notify_once(); // Notifies that one item was created
    }

    ++commonData.completed_producer_num;
    commonData.added_item_condition.notify_all(); // Notifies all consumers
}

static void consumer(ThreadData &commonData)
{
    size_t total_process_data_num = inputs.size();

    while(commonData.atomic_processed_data_num < total_process_data_num) {

        std::unique_lock<std::mutex> queue_lock(commonData.queue_indeces_mtx);
        {
        // Wait if there are no items to consume
        commonData.added_item_condition.wait(queue_lock, [&commonData, &producer_num] {
            return !commonData.item_indices.empty()
                    || commonData.completed_producer_num == producer_num;
        });} // automatically releases lock

        if (commonData.processedInputFilesIndices.empty())
            return; /* There are no producers in execution or items to consume */

        // Gets produced item
        commonData.queue_indeces_mtx.lock();
        int item_index = commonData.item_indices.front();
        commonData.item_indices.pop();
        commonData.queue_indeces_mtx.unlock();

        // Consumes an item
        Item &item = commonData.items[item_index]
        consume_item(item);
        ++commonData.atomic_processed_data_num;
        commonData.consumed_item_condition.notify_once();
    }
    ++commonData.completed_consumer_num;
    commonData.consumed_item_condition.notify_all();
}
```

Reducing lock conflicts: When mutual exclusion is needed, it is important to **use atomic variables instead of mutexes** due to the fact that atomic variables yield better results than mutexes. In cases in which it is necessary to use mutexes to achieve mutual exclusion, it is important to **minimize the time a critical section is blocked** in order to let other threads access that resource.

3.2 Hidden-line removal

Cadmatic hidden-line removal algorithm was developed in the 1980s to remove occluded lines from wireframe views (only includes lines and vertices) and 2D drawings. It was developed considering several approaches and ideas of the published scientific papers of that decade. Although the Cadmatic algorithm for hidden-line removal is based on those publications, it was implemented and tailored following the most convenient and beneficial ways for Cadmatic.

Cadmatic applies hidden-line removal to wireframe views in order to remove occluded lines. These wireframe views are visual representations of 3D objects using only "wires" (lines and vertices). Cadmatic users can create custom views by specifying view properties such as viewpoint, view direction, up vector, and view limits. In addition to that, the user can select specific objects or parts of the model (e.g. decks in a ship) to visualize them in these wire-frame views. After that, these wire-frame views can be used to create user-defined 2D drawings. 2D drawings are required for the manufacturing process of designed 3D components.

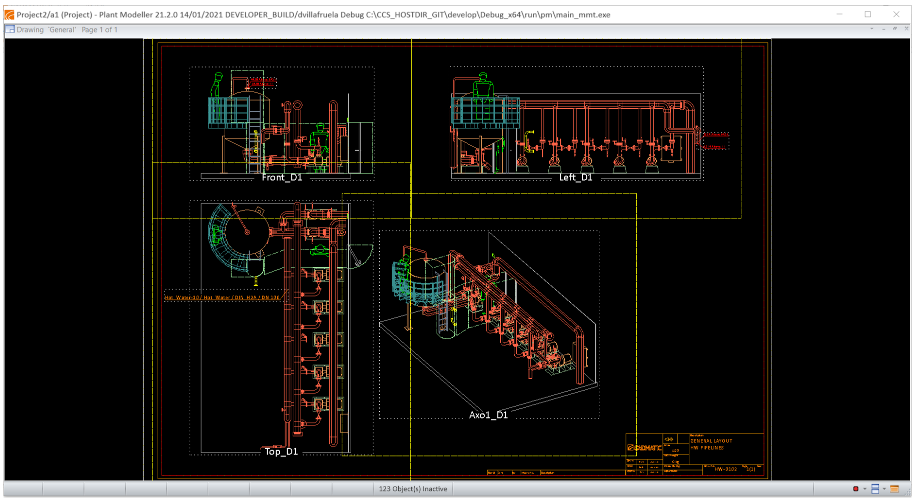

Project2/a1 (Project) - Plant Modeller 21.2.0 14/01/2021 DEVELOPER_BUILD/dvillafruela Debug C:\CCS_HOSTDIR_GIT\develop\Debug_x64\run\pm\main_mmt.exe
Front_D1
Left_D1
Top_D1
Axo1_D1
123 Object(s) Inactive

Figure 3.2: Example of a drawing in Cadmatic with four wire-frame views.

Cadmatic hidden-line removal is an **edge intersection algorithm** that determines edge visibility changes by finding intersection points between different lines (polylines or edges), or between a line and an object plane. A **polyline** is a single object composed of a connected sequence of line segments.

Cadmatic hidden-line removal algorithm has two main parts: **local hidden-line removal** and **global hidden-line removal**. Both local and global hidden-line removal work in a similar way by determining intersection points between lines, and between lines and planes. From these intersection points visibility changes are computed.

The main difference between local and global hidden-line removal resides in the objects that are considered to calculate visibility changes. Local hidden-line removal computes visibility changes on each object independently. On the other hand, global hidden-line removal computes visibility changes considering how other objects occlude the current object. Notice that Cadmatic uses some specific tolerance or epsilon values needed for computing visibility changes in both local and global hidden-line removal algorithms.

3.2.1 Bucketing acceleration

This technique was already implemented by Cadmatic several years ago in order to reduce the time required to compute both local and global hidden-line removal for complex objects that contain numerous numbers of faces and edges. Using bucketing acceleration allows to avoid comparing all planes of over objects against unnecessary planes of their corresponding under objects when calculating visibility changes on edges between two planes. Over objects are located on top of under objects and they partially or totally occlude them. It exclusively compares a plane against those planes whose buckets are intersecting that reference plane, reducing the number of planes that need to be compared.

For each complex object, a grid of buckets is created, being this grid a matrix of buckets. The width and height of this grid are determined by the bounding box of that complex object. Each bucket in this grid has similar dimensions and it corresponds to a specific part of that object.

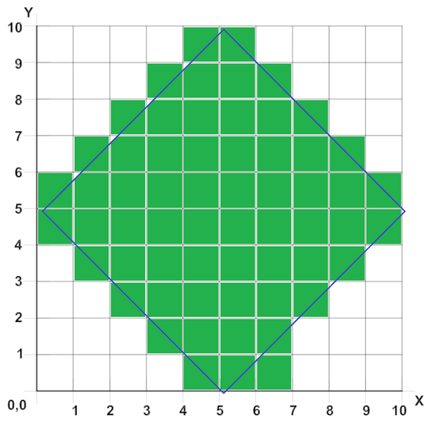


Figure 3.3: Bucket acceleration.

Each bucket in this grid contains intersecting planes of all objects. Therefore, it is only necessary to test a plane against those planes contained in the intersecting buckets of that object, discarding visibility tests against non-intersecting planes.

It is important to mention that using bucketing optimization provides better execution times than normal hidden-line removal without bucketing. However, it is theoretically possible to obtain worse execution times in the case of having drawings in which most of the objects are overlapping, testing the same redundant visibility tests.

For this reason, it is important to properly tailor the number of buckets used in the application. A higher number of buckets yields better results when most objects are scattered. A lower number of buckets yields better results when objects are closer or intersecting.

A dynamic bucketing approach that determines the number of buckets on execution would probably yield better results than a static bucketing approach. Cadmatic could benefit from this dynamic approach, whose current bucketing approach for complex objects is statically determined.

Even thought bucketing acceleration was a great improvement in the performance of the Cadmatic hidden-line removal, it was decided not to disclose its implementation due to the fact that the work done in this thesis did not modify the original Cadmatic implementation.

3.2.2 Local hidden-line removal

Local hidden-line removal is applied independently to each object in the wire-frame view. Typically an object is composed of several planes (faces), and those planes are composed at the same time of edges. In local hidden-line removal, only planes and edges of an object are considered.

The input of the local hidden-line removal algorithm is a set of model objects. Consider a cube, formed by six faces of the same size. Figure 3.4 below represents a cube whose occluded edges have not been yet removed:

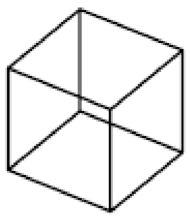


Figure 3.4: Cube whose occluded-lines are still visible. Local hidden-line removal has not been yet applied to this cube and it is possible to observe all occluded edges.

Local hidden-line removal algorithm requires iterating independently through all objects in a view, loading each object information into memory. An object has several faces and edges assigned to it. It is required to reset the visibility of those lines (edges and polylines) at the beginning of the algorithm.

Once an object is loaded and its visibility is reset, it is time to iterate through all faces of that object, calculating how each face hides edges of the rest of the faces of that specific object. The same process is repeated with polylines.

In the specific case of checking the visibility between a plane and any kind of line, it is possible that a plane totally occludes or excludes a line, marking the line as invisible or invisible respectively and its calculation trivial. However, there are situations in which a plane occludes a line but only partially. Comparing an edge of the plane of an object against another line segment (edge or polyline), or an object polyline with the rest of polylines, is done by using a **line-segment intersection algorithm** approach. In this approach, all the intersecting points between these two line segments are located, storing the visibility information of the generated line parts.

From these generated line parts, **a graph** can be generated containing these interception points (nodes) and visibility values (edges). Although the algorithm only stores the distances between points and not the Cartesian coordinates (x, y), it is still possible to obtain them afterward by using the "equation of a straight line": $y = mx + n$.

It is important to remark that Cadmatic local hidden-line removal algorithm does not use any graph to store the intersecting points, it only stores distances from points and visibility information.

When the local visibility of an object is determined, it is necessary to create line segments containing this information. Hidden parts of edges are permanently invisible, being only necessary to create new line segments for visible parts using the coordinates of the interception points and the visibility information.

After applying local hidden-line removal, occluded edges are completely removed.

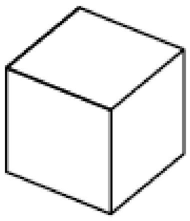


Figure 3.5: Cube whose hidden-lines are not visible

In most cases, local hidden-line removal is only computed once, and after that, the visibility results are stored so it is not necessary to compute them again.

3.2.2.1 Multi-threading local hidden-line removal

Multi-threading local hidden-line removal by applying data parallelism, splitting the workload among different threads symmetrically, is not a trivial task. In order to be able to parallelize local hidden-line removal, it is necessary to compute visibility changes on different objects on parallel without race conditions or conflicts among threads.

Removing static and global variables In local hidden-line removal, there are numerous global variables that are accessed by many methods all around the code in order to compute local hidden-line removal. Each object in a wire-frame view, also mentioned in this thesis as **segment** is loaded from disk and saved into these global variables. The loaded information of a *segment* is stored in different kinds of variables:

1. Variables that define the number of entries allocated for arrays holding data of segment that is being built (active segment).
2. Variables that point to allocated arrays holding data of segment that is being built (active segment).
3. Variables that define the current minimum bounding box of a segment that is being built (active segment).
4. Variables holding control data related to the active segment.

5. Variables used as buffers to store temporal results, avoiding reallocating memory.

The existence of static and global variables represents a big problem, due to the fact that the whole Application Programming Interface (API) for hidden-line removal is not thread-safe because most of its methods are accessing global variables. When the original code was implemented, this did not suppose a problem because most computers only had a single CPU, and were only able to run a thread or process at a specific time.

In order to make the code thread-safe, it was necessary to encapsulate these global variables referring to *segments* inside a **struct**, creating a **segment context** that contains all the information referring to a single active segment that is in the process of being computed. This way, it is possible to determine visibility changes between segments on parallel, by each thread assigning a different segment context, being each thread able to compute a different segment independently without conflicting with other threads.

After encapsulating these previously refereed global variables into a struct called `HGR_sgm_ctx`, there is some additional information relative to a segment, such as the status of the index of the 2D object (segment) in its wireframe view, and extra synchronization information used by each thread in order to protect the view data structure.

```
/* Active segment */
struct HGR_A_Sgm {
    HGR_INT sgm_ix = −1;  // active segment index
    HGR_sgm_ctx sgm_ctx; // encapsulates information relative to a segment
    HGR_INT status = 1;

    std::recursive_mutex* view_mutex = NULL; // protects the view
    bool reset_view = true; // false if several active segments are used simultaneously.
};
```

Once all global variables related to a *segment* are encapsulated, it is necessary to provide that segment context to each method that requires it, replacing old static and global references with this `HGR_A_Sgm` context from these methods.

In many of the functions used by the hidden-line removal algorithm exist accesses to common variables such as the active view and files, that need to be executed in mutual exclusion in order to avoid race conditions. `view_mutex` allows each thread to block the rest of the threads in order to protect the common wireframe view to all segments. It is important to remember that it is indispensable to reduce the time a resource is blocked, in order to avoid making other threads wait unnecessarily.

Recursive mutexes Accessing mutex-protected methods multiple times by a single thread produces a deadlock. This is a problem in hidden-line removal because some protected methods have conditional calls to other protected methods that also require mutual exclusion. The safest approach to deal with this issue is to use a recursive mutex. This way, it is guaranteed the algorithm will not suffer deadlocks.

Granularity When following a data parallelism strategy it is essential to define the thread granularity. It is possible to make each thread work with more complex data or with simpler data. A thread processing complex data is coarser than another thread that processes simpler data. In most cases, data granularities can be easily detected by looking at the block of code where there are loops that iterate through different elements.

At the time of analyzing the part of the code to parallelize using multi-threading in local hidden-line removal, there were several granularity levels to take into consideration.

1. ***Segment* granularity**: 2D objects in a wire-frame view.
2. **Reference plane granularity**: A segment can have several faces. These faces are used to see how they occlude other faces and polylines of that object.
3. **Line and plane visibility granularity**: Planes and polylines of objects whose visibility is computed by considering a *reference plane*.

Here is the pseudo-code of the for loops that could be parallelized:

Algoritm 5 Loops in Local Hidden-Line Removal - Pseudo-code

```
1: for each $segment \in Segments$ do
2:     for each $plane_{reference} \in segment.planes$ do
3:         for each $plane_{test} \in segment.planes$ do
4:             $visibility\_of\_plane\_edges(segment, plane_{reference}, plane_{test});$
5:         end for
6:         for each $line \in segment.lines$ do
7:             $visibility\_of\_line(segment, plane_{reference}, line);$
8:         end for
9:     end for
10: end for
```

Parallelizing local hidden-line removal by splittings segments among several threads seems to be the most intuitive approach, due to the fact that each segment can be computed independently from the rest because local hidden-line removal only individually considers faces and edges of a single segment to compute visibility changes.

The reference plane granularity and the line-plane granularity were rapidly discarded due to numerous simultaneous accesses to the same segment variables, making their parallelization inefficient because of the large number of blocks of code that required mutual exclusion. In addition, it is not possible to use a line-plane granularity due to the existence of a feedback loop in which each plane's visibility depends on the visibility of the previous one.

As a general rule of thumb, it is crucial to remark that the fewer critical sections and mutexes an algorithm has, the better its parallelization (speedup) is going to be. Additionally, the balance of the workload among threads is an important factor to consider. It is not desirable to have some processing elements idle whereas others are carrying out all work.

Applying parallelism Once the granularity level is specified, it is time to divide the data among threads.

This code below represents a simplification of the original algorithm.

```
void remove_local_hlines()
{
    const int store_to_file = 1;
    int rm_local_hlines = 1;

    HGR_e_shdl  *segment_header = Hgr_Shdl[I_view]; // a view has segments assigned to it.
    for (HGR_INT i=0; i < H_view->s_shdl; i++, p_s++) {
        if (segment_header->st_size <= 0) continue; // empty slot
        if (!(segment_header->hl_size & HGR_MASK_RM_LOCALHL)) continue;

        hgr_reopen_sgm(I_view, i);  // Loads segment information

        // Performs local hidden-line removal
        hgr_close_segment(store_to_file, rm_local_hlines);
    }
}
```

As it can be observed, the loop iterates through all segments in the view, checking their header information in order to know if they are in used or if hidden-line removal has been already applied. If not, they are loaded from the file of the view and local hidden-line removal is performed afterward.

Efficient thread-competitive approach The following code represents an efficient thread parallelization of the code mentioned above. It has a lambda function `hidden_line_removal_task` that makes threads efficiently compete against each other in order to compute local hidden-line removal on the next available *segment*. This competitive approach yields a symmetric workload among threads.

Each thread increments an atomic variable `atomic_segment` that represents the index of the segment that is about to be computed. This incremented atomic index `++atomic_segment` is stored locally, in order to avoid losing the value when other threads increment it. Remember that operations `variable++` and `++variable` are not the same, this last one does not create an extra copy.

```
void remove_local_hlines(parallel::WorkExecutor *threadPool)
{
    const size_t sgm_no = H_view->s_shdl;
    const int store_to_file = 1, m_local_hlines = 1;

    std::atomic_int atomic_segment(-1);
    HGR_A_Sgm *active_sgms = NULL; // each thread has its own HGR_A_Sgm

    auto hidden_line_removal_task = [&active_sgms, &sgm_no, &use_visibility_culling, &
     store_to_file, &atomic_segment](size_t thread_ix) {
        const HGR_e_shdl *segment_header = Hgr_Shdl[I_view];

        HGR_A_Sgm& active_sgm = active_sgms[thread_ix];
        int segment;
        while ((segment = ++atomic_segment) < sgm_no) {
            if (segment_header[segment].st_size <= 0) continue; // empty slot
            if (!(segment_header[segment].hl_size & HGR_MASK_RM_LOCALHL)) continue;

            hgr_reopen_sgm(I_view, segment, active_sgm); // Loads segment information

            hgr_close_segment(store_to_file, rm_local_hlines, active_sgm); // Local HLR
        }
    };
    ...
}
```

Notice that before calling the lambda function, it is necessary to set the active segments and grid of buckets (bucketing acceleration) that are going to be used inside the lambda function. Each thread will access those variables by using their unique thread index, which is given as a parameter by the lambda function.

The lambda function described above `hidden_line_removal_task` can be used either by a single thread or several threads. When using a single thread it is impossible to have race conditions, so the lambda function can be called directly. On the other hand, when there are several threads it is necessary to copy, prepare and split data among each thread. It is necessary to create an active segment for each thread, so threads can work in parallel without conflicting with each other. In addition to that, it is necessary to to combine the partial thread results at the end of the execution.

```
    ...
    if (threadPool == NULL) {  // A single thread
        active_sgms = &Hgr_A_sgm;

        hidden_line_removal_task(0); // Thread index is 0
    }
    else { // Several threads
        size_t n_active_sgms = std::min(threadPool->getNumThreads(), sgm_no);

        // Creation and initialization of active segments
        std::recursive_mutex view_mutex;
        std::vector<HGR_A_Sgm> active_sgms_vector(n_active_sgms);
        prepare_active_sgms(active_sgms_vector, view_mutex);

        active_sgms = active_sgms_vector.data();
        threadPool->parallelFor(hidden_line_removal_task, n_active_sgms);

        ... // Combining of all thread results
    }

    ...
} // End of move_local_hlines
```

Notice in the code how the parameters of the lambda function are configured depending if we are using a single thread or a multi-thread version. In the single thread version of local hidden-line removal, the `active_sgms` variable points to a single variable, whereas if a pool of threads is used `active_sgms` variable points to the row pointer of an *std::vector*.

Conflicts in local hidden-line removal In the case of most global and static variables, the concurrency issues were solved by encapsulating all global and static segment-referring variables in a segment context and creating and assigning each segment context to each thread. So each thread can work with its own variables without conflicting with other threads. However, there are cases inside the `hgr_reopen_sgm` and `hgr_close_segment` in which there are critical sections that need to be protected in order to avoid race conditions.

Inside the `hgr_reopen_sgm` function there are numerous functions that need to be executed in mutual exclusion, such as:

1. `Hgr_lda_sgm`: It loads a segment from disk to the current view.
2. `hgr_delete_sgm`: It removes a segment from the current view.
3. `hgr_create_sgm`: It creates a segment and adds it to the view.

In addition to that, it is necessary to protect the view structure when applying local-hidden-line removal to a single segment (`hgr_close_segment`) because results are ultimately stored in the view data structure for being used afterward for global hidden-line removal. It is important to mention that only small parts of code in those functions are executed in mutual exclusion in order to avoid blocking other threads for longer than necessary.

Showing all the code of local hidden-line removal is not possible due to its extension and Cadmatic copyrights. The pseudo-code of the algorithm was explained previously at Algorithm 5.

Mutexes are wrapped in the class `parallel::MutexWrapper` that overrides most of their methods so when the non-multithread version of local hidden-line removal is executed the wrapped mutex is not locked.

```
{
    parallel::MutexWrapper<std::recursive_mutex> view_mutex(active_sgm.view_mutex, true);

    Hgr_set_sgbox(Hgr_A_view, active_sgm.sgm_ix, a_sgm, active_sgm.view_mutex);

    // Set up min/max bounding box of the view.
    a_view->x_max = std::max(static_cast<HGR_FLT>(sgm_ctx.max_x), a_view->x_max);
    a_view->x_min = std::min(static_cast<HGR_FLT>(sgm_ctx.min_x), a_view->x_min);
    a_view->y_max = std::max(static_cast<HGR_FLT>(sgm_ctx.max_y), a_view->y_max);
    a_view->y_min = std::min(static_cast<HGR_FLT>(sgm_ctx.min_y), a_view->y_min);
    a_view->z_max = std::max(static_cast<HGR_FLT>(sgm_ctx.max_z), a_view->z_max);
    a_view->z_min = std::min(static_cast<HGR_FLT>(sgm_ctx.min_z), a_view->z_min);
}
```

Functions such as *Hgr_hide_data*, use a stack memory allocation in order to avoid doing heap allocations at run time. Even though it yields good results in sequential algorithms, it entangles and complicates the parallelization of the algorithm:

Algoritm 6 Memory stack allocation issues - Pseudo-code

```
1: $temp\_memory \leftarrow global\_temp\_memory[HGR\_NUM\_ITEMS]$;
2: if old_num_items + new_num_items >= HGR_NUM_ITEMS then
3:     $temp\_memory = malloc((old\_num\_items + new\_num\_items) * sizeof(*temp\_memory))$;
4: end if
```

As it was previously mentioned, the problem is easily fixed by removing the global variable, and providing a stack memory allocation to each thread, providing this memory variable as an argument to the function, so they can work independently from each other.

3.2.3 Global hidden-line removal

Once local hidden-line removal is carried out, obtaining line segments from locally visible parts, it is time to apply global hidden-line removal to compute how each object affects the visibility of other objects. Global hidden-line removal is different from local hidden-line removal, in the local hidden-line removal only visibility changes in one object at each time were considered, other objects were not considered. In global hidden-line removal, other objects can cause visibility changes to the current object in the case it is occluded by others.

The first step to carry out in global hidden-line removal is to load the visibility information obtained in the previous step (local hidden-line removal). After that, objects are sorted by their depth (z-index). Objects closer to the screen are processed first to discard visibility comparison against other objects that are already occluded. Sorting by depth should cover lines as soon as possible, avoiding computing hidden lines that would be later covered by a less distant object.

Global hidden-line removal compares all *over objects* (closer to the screen) against all possible *under objects*, objects that can be occluded and which have more depth than the *over object*. It computes how *over objects* occlude *under objects*. Under objects that are not occluded by *over objects* do not change their visibility.

In order to understand the meaning of *over objects* (*segments*) and *under objects* (*segments*), a deck of cards can be used as an analogy. In a deck of cards, some cards are over some other cards and under other cards.

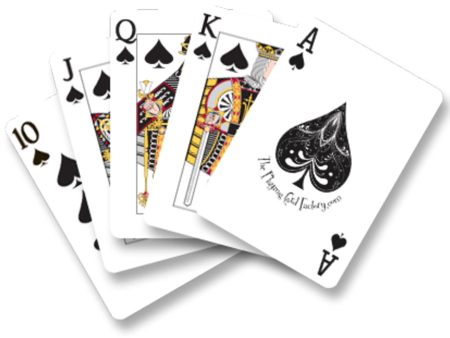


Figure 3.6: Cards representing over and under segments.

As it can be observed in the image, the Q and J of spades are over the 10 of spades. If two threads are storing simultaneously visibility results to the 10 of spades, it produces race conditions.

As in local hidden-line removal algorithm, global hidden-line removal is a **line-segment intersectional algorithm**, finding intersection points between line segments but this time visibility changes are calculated comparing edges and polylines of over segments against edges and polylines of under objects, modifying always the visibility of the under objects. The pseudo-code of global hidden-line removal is illustrated in the algorithm 7.

Local visibility results from the previous step (local hidden-line removal) are updated using these new visibility changes, obtaining a wireframe representation of all objects in that particular view without occluded lines.

3.2.3.1 Multi-threading global hidden-line removal

Multi-threading global hidden-line removal is easier to achieve than multi-threading local hidden-line removal because there are fewer critical sections. However, its parallelization is still challenging due to the numerous global and static variables that need to be removed and the asymmetry of the data itself that causes work imbalances among threads.

This section describes all strategies followed in order to optimize global hidden-line removal and how it was made thread-safe.

Granularity As it was mention previously, the first step is to analyze all the different granularity levels in the global hidden-line removal in which loops or recursive functions are located.

In the case of global hidden-line removal, it has a similar loop structure (granularity levels) as local hidden-line removal. However, it has one extra granularity level due to the fact that global hidden-line removal additionally considers other segments at the time of computing visibility changes. The different granularity levels of global hidden-line removal are:

1. **Over Segment granularity**: 2D objects that are located on top of other 2D objects.
2. **Planes of over segments granularity**: Each *over segment* can have faces (planes).
3. **Under Segment granularity**: 2D objects that are located under other 2D objects.
4. **Planes and polylines of under segments granularity**: Each *under segment* can have faces (planes) and polylines.

This pseudo-code represents a very simplified structure of the different granularity levels of Cadmatic global hidden-line removal:

Algoritm 7 Loops in Global Hidden-Line Removal - Pseudo-code

```
for each $segment_{over} \in Segments$ do
    $Segments_{under} \leftarrow$ get_possible_under_segments($segment_{over}$)
    for each $plane_{over} \in segment_{over}$ do
        for each $segment_{under} \in Segments_{under}$ do
            for each $plane_{under} \in segment_{under}$ do
                visibility_of_plane_edges($segment_{under}, plane_{over}, plane_{under}$)
            end for
            visibility_of_lines($plane_{over}, segment_{under}$);
        end for
    end for
end for
```

Using a plane granularity $Segments_{under}$ or $plane_{under}$ for parallelization is not possible due to the fact that these for loops are feedback loops in which the current result dependents on the previous iteration.

Even though the $segment_{under}$ granularity is a possible good candidate, it was decided to parallelize the first loop of $segment_{over}$, choosing the **over segment granularity**, due to the fact that in each iteration, it is necessary to calculate all $Segments_{under}$ of each $segment_{over}$. In addition to that, asymmetry in complex hierarchical data causes bigger workload imbalances when working with more fine-grained elements than with coarse-grained ones [14], when other elements can not be processed until the current one is complete by all threads. Imagine a situation in which each thread computes different parts of an object, they would have to wait until the last thread ends before starting computing parts of other objects. However, if an object is chosen as granularity, the waiting times become lower.

Applying parallelism Once the granularity level is specified, it is time to split the data among threads. Notice that global hidden-line removal uses the visibility information saved into a view during local hidden-line removal. When removing hidden-lines from a **new view**, both local and global hidden-line removal are applied. However, when removing hidden-lines from an **existing view** in which hidden-line removal was already applied successfully, only global hidden-line removal is executed due to the fact that the local hidden-line removal algorithm already recorded the visibility information into the view itself. In the case that a view is modified by adding objects to it, local hidden-line removal algorithm needs to be executed again.

The first step done in global hidden-line removal is to load visibility information of all objects existing in a view. This contains the output produced by local hidden-line removal. Secondly, after loading this visibility information, it is necessary to compute the visibility changes of global hidden-line removal. Thirdly, if there were no errors, this information is stored in the view.

Algoritm 8 Main Parts in Global Hidden-Line Removal

```
 1: if load_segments($view$) != OK then
 2:     return error
 3: end if
 4: if remove_hidden_lines($view$, $threadPool$) != OK then
 5:     return error
 6: end if
 7: if store_segments($view$) != OK then
 8:     return error
 9: end if
10: return success
```

The first and third function respectively `load_segments` and `store_segments` can not be parallelized due to the fact that they are reading and saving content into the same view. However, the `remove_hidden_lines` can be easily parallelized.

Inside this global hidden-line removal function `remove_hidden_lines`, the first thing the algorithm does is to sort the segments by depth using their z-index:

```
HGR_INT sgm_list = ( HGR_INT * ) sys_malloc( size );

/* sort segment indices by z_max of referenced segment */
qsort((void*)sgm_list, sgm_no, sizeof(*sgm_list), [] (const void* c, const void* d) {
    int cc = *((int*)c);
    int dd = *((int*)d);

    if (Sgm_hed[cc]->z_max >= Sgm_hed[dd]->z_max) return(-1);

    return(1);
});
```

Sorting segments by their depth is an important step in order to speed up the process of computing visibility changes. A segment on top of many will rapidly discard visibility changes of its under segments.

After this, the segment header information and data information are initialized. In addition, the visibility information is reset.

```
for (size_t i = 0; i < H_view->s_shdl; i++) {
    if (Sgm_hed[i] == NULL)
        continue;

    sgm_H_hed = Sgm_hed[i]; // segment header info
    sgm_H_h = &Sgm_h[i]; // segment data

    initialize_segment_info(sgm_H_hed, sgm_H_h);

    if (global_hlr_was_applied_before) {
        reset_visivility(sgm_H_hed, sgm_H_h);
    }
}
```

Time spent at these sorting and initialization steps is marginal, and their parallelization does not improve performance.

Before starting explaining the parallelization of global hidden-line removal, it is important to describe all the most important parts of the previous implementation of global hidden-line removal:

Algoritm 9 Loops in Global Hidden-Line Removal - Pseudo-code

```
1: free_mem[HGR_FREE_MEM];
2: $pln\_h\_alloc, dpln\_h \leftarrow init(pln\_h\_alloc, dpln)$;
3:
4: for $segment_{index} = 0; segment_{index} < segment\_num; ++segment_{index}$ do
5:    $sgm\_over \leftarrow sgm\_list[segment_{index}]$;
6:    $sgm\_hed\_over \leftarrow Sgm\_hed[sgm\_over]$;
7:
8:    if $segment_{over}.inten ==$ transparent then
9:       continue;
10:   end if
11:
12:   $sgm\_und \leftarrow \varnothing$;
13:   $sgm \leftarrow 0$;
14:   for $under_{index} = 0; under_{index} < segment\_num; ++under_{index}$ do
15:      sgm_hed_under = Sgm_hed[sgm_list[$under_{index}$]];
16:      if not is_under_segment($sgm\_hed\_over$, $sgm\_hed\_under$) then
17:         continue;
18:      end if
19:      $sgm\_und[sgm++] \leftarrow sgm\_list[under_{index}]$;
20:   end for
21:
22:   if sgm == 0 then
23:      continue;
24:   end if
25:
26:   sgm_h_over = Sgm_h[sgm_over];
27:   $sum\_of\_p\_counts \leftarrow 0$;
28:   for $plane_{index} = 0; plane_{index} < segment_{over}.num_planes; ++plane_{index}$ do
29:      if $init\_plane\_data\_global(sgm\_hed\_over, sgm\_h\_over, pln\_h,$
         $pln\_no, sum\_of\_p\_counts, pln, pln\_h\_alloc, dpln\_h) < 0$ then
30:         continue;
31:      end if
32:      $determine\_under\_segment\_visibilities(sgm, sgm\_und, pln\_no, pln\_h,$
         $free\_mem)$;
33:   end for
34: end for
```

Once all segment information and other variables have been allocated and set it is time to start global hidden line removal. The first step is to determine under segments of each over segment. After that, for each plane of each over segment, it is necessary to calculate how this $plane_{over}$ hides each $segment_{under}$ by calling $determine\_under\_segment\_visibilities$.

Once it was decided that the parallelization would be applied to the *over segments*, it is time to represent the basic structure of the multithread version of global hidden-line removal. The code that is concurrently executed by several threads can be encapsulated in a lambda function `global_hidden_line_removal_task`. Using this approach, `global_hidden_line_removal_task` can be easily executed by a pool of threads or by a single thread (calling the lambda function by the main thread).

```
int *thread_rts;
std::atomic_int atomic_segment(-1);
std::atomic_bool stop_computing_hidden_lines = false;
std::vector<std::atomic_int> under_sgm_conflicts(threadPool ? sgm_num : 0);

auto global_hidden_line_removal_task = [&] (size_t thread_ix) {
    ...
}
```

In order to make the lambda function compatible with a single or several threads, it is necessary to provide several arguments to *the capture clause* of the lambda function. These arguments are:

- `thread_rts`: an array, in which each thread will write on a specific position using its thread index, the result of its execution.
- `atomic_segment`: an atomic counter that is atomically incremented by each thread to get access to the next segment to compute.
- `stop_computing_hidden_lines`: if one thread reports an error, the rest of the threads will stop their execution.
- `under_sgm_conflicts`: vector of atomic indices needed to protect common accesses to same under segment by several threads.

Those arguments of the capture clause need to be properly configured, depending on the number of threads (single or several threads).

```
int rt = 0;
if (threadPool == NULL) { // Single thread
    thread_rts = &rt;
    global_hidden_line_removal_task(0); // Global hidden line removal
}
else { // Pool of Threads
    size_t max_nro_threads = std::min(threadPool->getNumThreads(), sgm_no);
    std::vector<int> thread_rts_vector(max_nro_threads, 0);

    thread_rts = thread_rts_vector.data();
    threadPool->parallelFor(global_hidden_line_removal_task, max_nro_threads);  // Global hidden
     line removal

    /* checks threads finished their jobs correctly. */
    for (const int& thread_rt : thread_rts_vector) {
        if (thread_rt < 0) {
            ccs_fclose(fp);
            rt = thread_rt;
            break;
        }
    }
}
```

In the case that there are several threads it is necessary to combine the partial results into a single one.

The basic structure of the lambda function `global_hidden_line_removal_task` is the following one:

```
auto global_hidden_line_removal_task = [&] (size_t thread_ix) {
    // Memory allocations
    HGR_INT* sgm_und = (HGR_INT *)sys_malloc(H_view->s_shdl * sizeof(*sgm_list));

    #define PLN_H_ALLOC 30
    HGR_c_pln_str dpln_h[PLN_H_ALLOC];
    HGR_c_pln_str* pln_h = dpln_h;
    HGR_INT pln_h_alloc = PLN_H_ALLOC;

    HGR_DBL free_mem[(HGR_FREE_MEM) / sizeof(HGR_DBL)];
    HGR_d_point_str pln_h_vec_mem[(4096) / sizeof(HGR_d_point_str)];
    ...
    // scan through segments in order of z_max, start close to viewer
    HGR_INT segment_to_compute;
    while((segment_to_compute = ++atomic_segment) < sgm_no && !stop_computing_hidden_lines) {
        ...
        sgm_over = sgm_list[segment_to_compute];  /* sorted indexes */
        sgm_hed_over = Sgm_hed[sgm_over];

        sgm_und = get_under_segments(...);
        ...
        // loop through all "under" segments
        for (HGR_INT pln = 0, pln_no = 0, sum_of_p_counts = 0; pln < sgm; pln++) {
            // Initialize plane data for the processed plane of the hiding segment
            if (init_plane_data_global(sgm_hed_over, sgm_h_over, pln_h, pln_no, sum_of_p_counts,
    pln, pln_h_alloc, dpln_h) < 0)
                continue;
            // Compute how the setup plane hides edges and lines in under segments.
            determine_under_segment_visibilities(sgm, sgm_und, pln_no, pln_h, (HGR_CHR*) free_mem
    , &under_sgm_conflicts);
        }
        ...
    }
    ...
}
```

In the multithread version of global hidden-line removal, there are situations in which different threads are computing visibility changes of different over segments applied to same under segments. Even though this situation does not happen often, it can cause conflicts among threads, in the case that several threads try to edit the same common under segment at the same time. It is important to notice at the time of parallelizing over segments that there can exist conflicts between under segments. This is the reason why read and write operations applied to under segments need to be protected in order to avoid read–write or write–write conflicts.

The time an under segment is locked is very limited and it only happens a couple of times per execution. For these reasons, it is suitable to use busy waiting by checking a flag atomic variable that informs if an under segment is currently being processed.

```cpp
int determine_under_segment_visibilities(..., std::vector<std::atomic_int> *under_sgm_conflicts)
{
    // loop through all "under" segments
    for (int j = 0; j < sgm; j++) {
        bool check_for_conflitcs = under_sgm_conflicts!=NULL && sgm_under<under_sgm_conflicts->
     size();

        // Several over segments can compute concurrently the same under segment.
        if (check_for_conflitcs) {
            std::atomic_int& conflicts_nro = under_sgm_conflicts->at(sgm_under);
            while (++conflicts_nro > 1) {
                while (conflicts_nro > 1) {
                    std::this_thread::yield();
                }
            }
        }
        ...
        if (check_for_conflitcs) (*under_sgm_conflicts)[sgm_under] = 0;
    }
}
```

In order to avoid mutexes it was decided to use **busy waiting** and an array of atomic integers `std::vector<std::atomic_int> under_sgm_conflicts` that is incremented by each thread before accessing an under segment.

The reason why **busy waiting** is used instead of other approaches is due to the fact that the time required by a thread to access and modify an under segment is small, and threads are usually processing different under segments making it unlikely to have two threads computing the same under segment simultaneously.

Other modifications and optimizations applied to global hidden-line removal are not shown in this thesis due to Cadmatic copyrights.

3.3 Collision detection

The Cadmatic implementation of the collision detection algorithm to detect collisions between two different sets of static objects was developed in the 90s following the scientific literature of the time. Cadmatic collision detection implementation was tailored specifically for the needs of Cadmatic.

Cadmatic collision detection is implemented following a sphere partitioning approach. This algorithm implementation is detailed in the following steps:

1. Select two sets of static objects, in which the user wants to check if there are collisions.

2. Prepare the pairs of objects in which collisions, contacts or clearances can appear. Every object has a bounding box, that wraps the object in a 3D space. These object bounding boxes can be used to discard collisions between objects rapidly.

3. Once all pairs of objects in which possible collisions can occur are determined, it is time to check collisions between object A and object B in each pair of objects. It is important to notice that every object is formed by many simple shapes (primitives) being necessary to check collisions between all primitives of the first object against all the primitives of the second object.

4. For each primitive of each object:
   (a) Bounding boxes of both primitives are compared to check possible collisions, contacts and clearances.
   (b) A sphere test is run for every primitive pair in which a collision/contact/clearance is possible to occur:
      i. The sphere test is based on the idea of space division, in which the intersection bounding box of both objects is set as the original test space and it is recursively subdivided into two halves, dividing recursively the intersected bounding box of these two primitives by the biggest axis of the test space.

The sphere test provides very good times when the objects are intersecting or the distance between them is not very close. However, when the objects are almost touching each other, the sphere test gets stuck in recursive space subdivision calls.

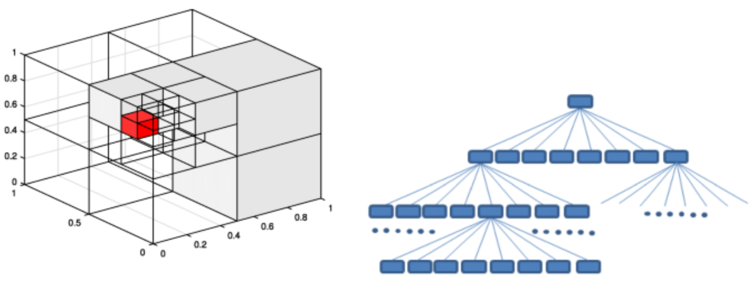


Figure 3.7: Space subdivision by collision detection.
Source: `https://en.wikipedia.org/wiki/Octree`

When spatially two large objects are close to each other but are not colliding and the box test does not recognize that, then the recursion needs to do a lot of work to figure out that there are no collisions.

3.3.1 Optimization

This section applies the optimization techniques described previously in this thesis in order to parallelize and improve the performance of collision detection.

The Cadmatic collision detection algorithm is divided into two main parts:

1. The first part is to determine which objects require detailed collision detection processing, based on bounding box tests.
2. The second part checks collisions, contacts and clearance violations in detail by using several collision tests such as a sphere partitioning approach.

Both parts are worth parallelizing, even though most performance improvement is obtained by parallelizing the second step.

3.3.1.1 Determining detailed collision object pairs

The first part of the algorithm takes two sets of objects testing each bounding box against each other and then checks the pairs of objects to check collisions are not excluded in the test. This function also caches detailed geometry data for every object that enters into detailed collision detection. As one object may be tested against multiple other objects, geometry creation and conversion routines need to be executed only once due to caching.

Below is the old implementation of this first part of the algorithm:

```
static int prepare_data_for_detailed_collision_detection(const int nr_models, const int nr_A, const int nr_B, const int *set_A, const int *set_B, const ModelFlags *model_flags, const coarse_test_info *coarse_info, const GMD_mbb *primitive_bounding_boxes, const SearchControlParameters *parameters, const char *geometry_data_path, coll_test_t *ctc, std::vector<CachedPrimitiveInfo> &primitive_cache, std::vector<TestPair> &pairs_for_testing)
{
    std::vector<bool> allowed_contacts(nr_models, false); // true for cavity planes tests
    std::vector<bool> compute_GMD_fast_info(nr_models, false);  // true for detailed CD
    float tolerance = std::max(1.0f, parameters->getClearance());

    for (int i = 0; i < nr_A; i++) {
        int mod_A = set_A[i];
        for (int j = 0; j < nr_B; j++) {
            int mod_B = set_B[j];

            // Don't test the model against itself
            if (mod_A == mod_B)
                continue;

            // if the two models are present in both sets tests only (A, B) but not (B, A)
            if (!(context->isInSetB(mod_A) && context->isInSetA(mod_B) && (mod_A > mod_B)) {
                if (coarse_info[mod_A].unit_group >= 0 && coarse_info[mod_A].unit_group == coarse_info[mod_B].unit_group)
                    continue;

                // Perform a fast box test against boxes of the whole object
                if (!boxtest_with_bboxes_no_hit(&coarse_info[mod_A].mbb, &coarse_info[mod_B].mbb, tolerance)) {

                    if (need_to_test_pair(ctc, mod_A, mod_B)) {
                        // Check whether an allowed contact exists between this pair
                        const MMT_allowed_contact_t *p_allowed = find_allowed_contact_between_objects(ctc, mod_A, mod_B);
                        if (p_allowed != NULL)
                            allowed_contacts[mod_A] = allowed_contacts[mod_B] = true;

                        pairs_for_testing.emplace_back(mod_A, mod_B, p_allowed);

                        compute_GMD_fast_info[mod_A] = compute_GMD_fast_info[mod_B] = true;
                    }
  \item Object pair granularity: each thread computes collision tests between two objects of two different sets.
            }
        }
    }

    /* Compute and cache GMD_fast_info for every object that requires detailed collision detection computations. Also compute vertex data for objects that enter into cavity plane tests, if the objects do not already have bsv info computed. */
    ...
    return 0;
}
```

Multi-thread version Object pair granularity is the only granularity that can be used in this first part of the algorithm. In this granularity, each thread computes if two objects from two different sets need to be tested in detail or not. It was decided to use an *equal indices-range distribution of data* to split the data among threads due to the fact that it takes a similar time to carry out fast bounding box tests on different pairs of objects.

The first optimization approach that was followed was to transform the two loops in the function into a single one. The first loop goes through the objects contained in the first set, and the second loop iterates through the elements of the second set. This loop transformation is done by applying an *index to matrix indices conversion*. After this conversion, each thread obtains a unique `mod_A` and `mod_B` from sets A and B respectively. Both values are used to store the results at `allowed_contacts` and `compute_GMD_fast_info_data` results, these two results need to be atomic because they are written by several threads simultaneously.

```
prepare_data_for_detailed_collision_detection(...)
{
    std::vector<std::atomic_bool> allowed_contacts(nr_models);
    std::vector<std::atomic_bool> compute_GMD_fast_info_data(nr_models);

    for (size_t index = 0; index < nr_models; ++index) {
        allowed_contacts[index] = false;
        compute_GMD_fast_info_data[index] = false;
    }
    ..

    auto work = [&](size_t from, size_t to, size_t thread_ix) {
        size_t i, j;
        for (size_t index = from; index < to; index++) {
            i = index / nr_B;
            j = index % nr_B;
            int mod_A = set_A[i];
            int mod_B = set_B[j];

            ...
    }
    if (threadPool == NULL) {
        work(0, nr_A * nr_B, 0);
    }
    else {
        threadPool->parallelFor(work, nr_A * nr_B);
    }
}
```

There are some situations in which the loops can not be removed. It is necessary to choose which loop should be parallelized in order to obtain the best performance.

If the first loop is parallelized it might produce performance issues due to the fact that the other loop can have more elements than the first loop or vice versa. Therefore, it is necessary to conditionally parallelize the iteration through the biggest set of elements in order to obtain the best performance.

```
prepare_data_for_detailed_collision_detection(...)
{
    std::vector<std::atomic_bool> allowed_contacts(nr_models);
    std::vector<std::atomic_bool> compute_GMD_fast_info_data(nr_models);

    for (size_t index = 0; index < nr_models; ++index) {
        allowed_contacts[index] = false;
        compute_GMD_fast_info_data[index] = false;
    }

    std::atomic_int atomic_i(-1);
    auto work = [&](size_t thread_ix) {

        bool invert_A_B = (nr_B > nr_A); /* parallelizes the set with more number of elements. */
        int local_nr_1 = (invert_A_B) ? nr_B : nr_A;
        int local_nr_2 = (invert_A_B) ? nr_A : nr_B;

        int i, j;

        int mod_A = -1, mod_B = -1;
        while ((i = ++atomic_i) < local_nr_1) {

            if (invert_A_B) {
                mod_B = set_B[i];
            }
            else {
                mod_A = set_A[i];
            }

            for (j = 0; j < local_nr_2; j++) {
                if (invert_A_B) {
                    mod_A = set_A[j];
                }
                else {
                    mod_B = set_B[j];
                }
                 ...
            }
        }
    }

    if (threadPool == NULL) {
        work(0);
    }
    else {
        threadPool->parallelFor(work, std::max(nr_A, nr_B));
    }
    ...
}
```

The new version of the algorithm uses more memory than the old existing version due to the fact that each thread needs to have its own stack buffer to store its variables and results. In addition to that, once all threads have calculated all their results it is necessary to combine them:

```cpp
/**
 * Moves all test pairs elements to the first vector of multiple_pairs_for_testing.
 *
 * @param multiple_pairs_for_testing[in]     vector of test pairs elements
 */
static void combine_pair_for_testing(std::vector<std::vector<TestPair>> &
    multiple_pairs_for_testing)
{
    if (multiple_pairs_for_testing.size() < 2)
        return;

    size_t nr_pairs = 0;
    for (size_t i = 0; i < multiple_pairs_for_testing.size(); ++i)
        nr_pairs += multiple_pairs_for_testing[i].size();

    std::vector<TestPair>& pairs_for_testing = multiple_pairs_for_testing[0];

    if (nr_pairs - pairs_for_testing.size() > 0) {
        pairs_for_testing.reserve(nr_pairs);

        for (size_t index = 1; index < multiple_pairs_for_testing.size(); ++index)
            for (const TestPair& testPair : multiple_pairs_for_testing[index])
                pairs_for_testing.push_back(testPair);
    }
}

coll_det2(...)
{
    std::vector<std::vector<TestPair>> multiple_pairs_for_testing(num_threads);

    r = prepare_data_for_detailed_collision_detection(nr_models, nr_A, nr_B, set_A, set_B, &
    model_flags, coarse_info, primitive_bounding_boxes, &parameters,
        geometry_data_path, &ctc, primitive_cache, multiple_pairs_f or_testing.data(), threadPool
    );

    combine_pair_for_testing(multiple_pairs_for_testing);
    const std::vector<TestPair> &pairs_for_testing = multiple_pairs_for_testing[0];
}
```

3.3.1.2 Performing detailed collision detection

After parallelizing the determination of collision object pairs, it is necessary to parallelize the detailed collision detection tests for each object pair that survived from `prepare_data_for_detailed_collision_detection` function.

Detailed collision detection uses the full object geometry, and it is only computed for those objects whose primitive bounding boxes intersect or are inside clearance violation distance.

Here is the original implementation of the detailed collision detection:

```
CollHitPoint results;  // Hit coordinates and distance
for (auto &test_pair : pairs_for_testing) {
    int mod_A = test_pair.m_modA;
    int mod_B = test_pair.m_modB;

    // Prepare to deal with an allowed contact between this pair.
    const MMT_allowed_contact_t *p_allowed = test_pair.m_pAllowed;
    if (p_allowed)
        ctc.nr_sent_with_allowed_contact++;

    auto result = collide2(mod_A, mod_B, p_allowed, &parameters, primitive_cache, &results);

    report(result, &coarse_info[mod_A], &coarse_info[mod_B], &results, o_fp); // output file

    if (toCuber_fp) // log file
        report_pairs_sent_to_cuber(&coarse_info[mod_A], &coarse_info[mod_B], p_allowed != NULL,
     convertToMacro(result), toCuber_fp);
}
```

For each `test_pair` two indices are obtained representing two objects in both sets. In the function `collide2` drives the spatial partitioning subdivision for one pair of objects returning a collision result (NO_HIT, CLEARANCE_VIOLATION, CONTACT, HIT). These collision results are logged and stored in two separate files. Implementation of `collide2` is hidden due to Cadmatic copyrights.

Multi-thread version In the case of this second part of the Cadmatic detection algorithm, there are two granularities that can be used to parallelize the algorithm.

1. Object pair granularity: each thread computes collision tests between two objects of two different sets.
2. Primitive pair granularity: each thread computes collision tests between two primitives of two different objects.

It was decided to use the object pair granularity because of its simplicity and fewer number of critical sections. A competitive work-load distribution was used to split the data among threads due to the fact that computation times for detailed collision tests between two objects are extremely asymmetric.

The first step that is necessary to do in order to parallelize this function is removing the common accesses to critical regions. Threads write collision information into the output file and the log files. The easiest option is to write the collision results on the file only once all results have been calculated. Another option that has not been implemented would be to use many producers (threads) that generate the collision results and a single producer that stores the results in the file. In this case, the time spent saving the collision results on the file is negligible compared to the time required to determine collisions between two sets of objects.

In addition to this, it is also necessary to use atomic variables in order to avoid race conditions when several threads are incriminating the same counter several times e.g. `nr_sent_with_allowed_contact_atomic`.

In addition to that, the function `collide2` is not thread-safe because it uses geometry functions in which the geometry information of primitives such as edges, normals and points of faces, and contours can be computed at run time in the case this information is missing.

```c
GMD_dir *gmd_bsv_face_normal(GMD_bsv_info *exp_bsv, int face_inx)
{
    GMD_edge_dir_and_length *gmd_bsv_edge_info(GMD_bsv_info *exp_bsv)
{
    if (exp_bsv->edges == NULL) {
        exp_bsv->edges = sys_malloc(exp_bsv->bsv->nedg * sizeof(GMD_edge_dir_and_length));

        for (int i = 0; i < exp_bsv->bsv->nedg; i++) {
            GMD_bspnt *p_one = &exp_bsv->pnt_tbl[exp_bsv->edg_tbl[i].pnt1];
            GMD_bspnt *p_two = &exp_bsv->pnt_tbl[exp_bsv->edg_tbl[i].pnt2];
            D3_pnt *dir = &exp_bsv->edges[i].dir;

            dir->x = p_two->p_x - p_one->p_x;
            dir->y = p_two->p_y - p_one->p_y;
            dir->z = p_two->p_z - p_one->p_z;

            exp_bsv->edges[i].len2 = dir->x * dir->x + dir->y * dir->y + dir->z * dir->z;
        }
    }

    return exp_bsv->edges;
}
```

In order to get rid of this problem and avoid using mutexes, it is necessary to pre-compute all primitives of each object, so when the collision module is asking for geometry information of a primitive, no initialization occurs.

Taking all these factors into account, the previous implementation of the detailed collision detection function was replaced by the following multi-thread version:

```
const size_t coll_test_pair_nr = pairs_for_testing.size();

std::vector<CollisionResult> coll_results(coll_test_pair_nr, CollisionResult::NO_HIT);
std::vector<CollHitPoint> hit_results(coll_test_pair_nr);  // Hit coordinates and distances

std::atomic_int atomic_coll_test_pair_ix(-1);
std::atomic_int nr_sent_with_allowed_contact_atomic(0);

auto test_coll_pairs_task = ([&](size_t) {

    CollisionResult coll_result;
    int coll_test_pair_ix;
    while ((coll_test_pair_ix = ++atomic_coll_test_pair_ix) < coll_test_pair_nr) {
        const TestPair& test_pair = pairs_for_testing[coll_test_pair_ix];
        const int& mod_A = test_pair.m_modA;
        const int& mod_B = test_pair.m_modB;

        // Prepare to deal with an allowed contact between this pair.
        const MMT_allowed_contact_t* p_allowed = test_pair.m_pAllowed;
        if (p_allowed)
            ++nr_sent_with_allowed_contact_atomic;

        coll_result = collide2(mod_A, mod_B, p_allowed, &parameters, primitive_cache, &
     hit_results[coll_test_pair_ix]);
        coll_results[coll_test_pair_ix] = coll_result;
    }
});

if (threadPool == NULL) {
    test_coll_pairs_task(0);
}
else {
    size_t required_threads = std::min<size_t>(threadPool->getNumThreads(), pairs_for_testing.
     size());

    if (required_threads > 1) {
        // precomputes data for BSV primitives - avoids using mutexes
        precompute_data_bsv_primitives(pairs_for_testing, primitive_cache, threadPool);
    }
    threadPool->parallelFor(test_coll_pairs_task, required_threads);
}

ctc.nr_sent_with_allowed_contact = nr_sent_with_allowed_contact_atomic;

// reports collision results
r = save_coll_results(pairs_for_testing, coll_results, hit_results, coarse_info, output_file);
```

It was noticed when calculating collisions between two primitives inside `collide2`, that there are some cases in which there are many recursive calls in the space partitioning of the test space, slowing down the computation. In some extreme cases, it can take dozens of seconds to test collisions between primitives of two objects.

When spatially two large objects are close to each other but are not colliding and the box test does not recognize that, then the recursion needs to do a lot of work to figure out that there are no collisions.

4 Results

Cadmatic hidden-line removal and collision detection are two important algorithms whose results have important implications in CAD and CAM. Collision detection takes an important role in CAD, allowing engineers and designers to detect flaws and mistakes in their designs. Hidden-line removal affects 2D drawings, and 2D drawings are required for manufacturing. Manufacturers produce designed components by taking these 2D drawings as reference.

Due to the importance of the results produced by hidden-line removal and collision detection, it is essential to assure that the parallelization of these two algorithms provides the same consistent results. In addition to that, it is important to measure their performance in different scenarios and using data of different complexities.

The testing of both algorithms was done using real Cadmatic projects in which customers were experiencing performance issues when working with hidden-line removal or collision detection. Due to confidentiality reasons, these testing environments are not shown in this thesis. Obtained results were recorded in order to check that they were consistent with their previous implementation, and their execution times were measured before and after the parallelization in order to estimate their parallelization speedup.

The testing machine used for measuring the execution times of both algorithms is a Dell Precision M7510 Core i7 16GB. Its processor is an Intel Core i7-6820HQ with 4 physical cores and 4 virtual cores, with a total of 8 threads. The base frequency of the processor is 2.7GHz.

4.1 Hidden-line removal

Execution times of the hidden-line removal algorithm vary accordingly to the number of objects in the selected view. Views with more objects require more time than smaller ones in order to compute hidden-line removal. For this reason, it was decided to apply hidden-line removal on different views that have different sizes.

One big cruise ship project with many drawings was selected as the environment in which the parallel version of hidden-line removal was tested. Only 13 views of different sizes from several drawings were tested because of the time required for measuring their execution times and checking the consistency of their results. It was necessary to publish each drawing and carry out a pixel comparison between newer publications and older publications in order to check that the results of the new parallel version are the same as the original ones.

As it was mentioned previously, hidden-line removal has two main parts: local and global hidden-line removal. Optimization results obtained in local and global hidden-line removal were measured independently in order to know more in detail the speedup of each one.

The obtained speedup results for **local hidden-line removal** are illustrated in the figure 4.1:

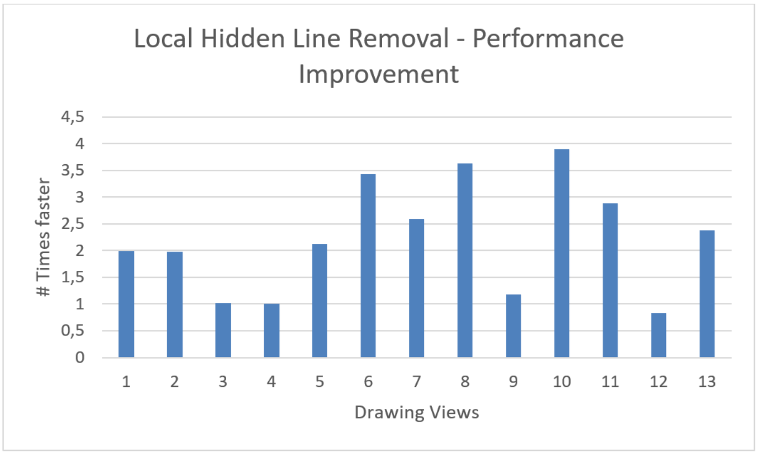


Figure 4.1: This chart shows the speedup obtained when applying the parallel version of **local hidden-line removal** on 13 different views using 4+ threads. The average speedup value is 2.226 and the standard deviation is 1.035, having a minimum speedup value of 0.836 and a maximum speedup value of 3.891.

The obtained speedup results for **global hidden-line removal** are illustrated in the figure 4.2:

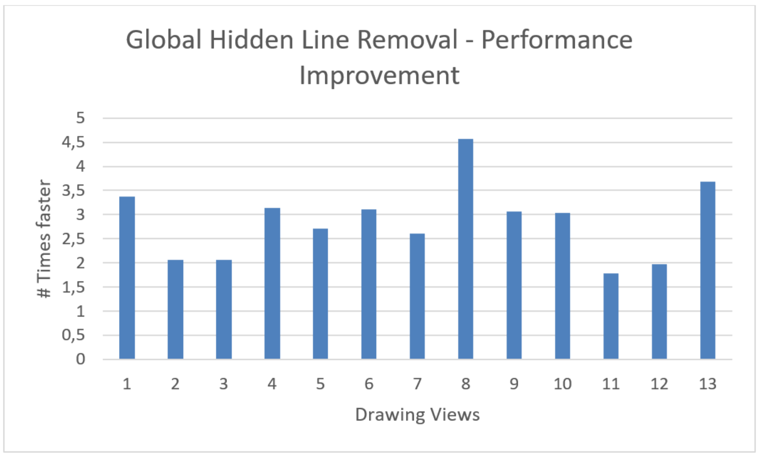


Figure 4.2: This chart shows the speedup obtained when applying the parallel version of **global hidden-line removal** on 13 different views using 4+ threads. The average speedup value is 2.859 and the standard deviation is 0.784, having a minimum speedup value of 1.774 and a maximum speedup value of 4.568.

The speedup of each element in these charts (figures 4.1 and 4.2) was obtained by calculating the ratio between the execution time of the original version and the execution time of the parallel version. A speedup of one means that the new execution time is equal to the old execution time.

Optimization results obtained in local and global hidden-line removal were satisfactory, on average 2.226 and 2.859 times faster on tested drawings than the non-multi-thread method implementation of local and global hidden-line removal respectively. It is important to mention that in local and global hidden-line removal there were big deviations between samples due to the fact that views have different numbers of objects and complexities. Hidden-line removal is also affected by the way objects overlap with each other.

In addition to that, the results obtained in global hidden-line removal were better than the ones obtained in local hidden-line removal because of a higher level of granularity, in which over and under segments are compared. It is important to know that local hidden-line removal is computed only once, and it is not carried out until the view is totally regenerated. Achieving less performance in local hidden-line removal in some odd cases would not cause any notice by Cadmatic users.

It is important to point out that there are some cases in which removing hidden-lines for one single object takes a notably longer time when compared with other objects. These problematic objects contain an unnecessary amount of details that make the computation more complex and longer. Typically these objects are imported into Cadmatic from other CAD tools such as AutoCAD.

4.2 Collision Detection

Execution times of the collision detection algorithm vary accordingly to the number of objects that are going to be considered, their complexity and their position. For testing the performance of collision detection, four big cruise ship projects were considered. Each of these projects has more than one million objects, that were tested for collisions against each other. The obtained speedup results are illustrated in the figure 4.3:

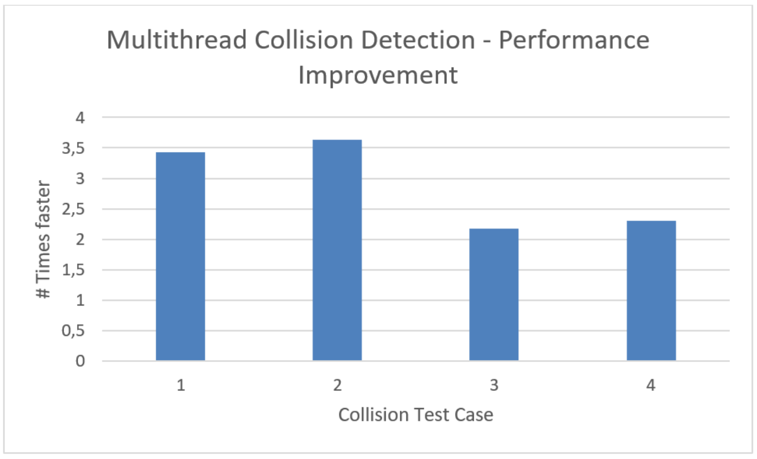


Figure 4.3: This chart shows the speedup obtained when executing the parallel version of collision detection on four different models using 4+ threads. The average speedup value is 2.911 and the standard deviation is 0.727, having a minimum speedup value of 2.179 and a maximum speedup value of 3.63.

The speedup of each element in this chart (figure 4.3) was obtained by calculating the ratio between the execution time of the original version and the execution time of the parallel version. A speedup of one means that the new execution time is equal to the old execution time.

Collision detection optimization results were satisfactory, obtaining an increase in performance of an order of 2.911, almost three times faster when compared to the original single thread version. In addition to that, the new collision data matches the original one, suggesting the multi-thread implementation provides the same output results as before.

It is important to point out that there are some special cases in which the time required by a thread to compute a collision between two primitives can be extremely large. It seems to appear when spatially two large objects are close to each other but are not colliding and the box test does not recognize that, then the recursion needs to do a lot of work to figure out that there are no collisions. That could explain the deviation in the data.

5 Discussion

The results obtained in the two study cases (hidden-line removal and collision detection) seem to validate the guidelines and approaches described in this thesis in order to optimize and parallelize complex and data-intensive algorithms.

Hidden-line removal and collision detection algorithms suffer from data asymmetry, making their parallelization more challenging due to work-load imbalances among threads. The suggested techniques in this thesis such as the "competitive work-load distribution of data" seem to mitigate the problem. However, the speedup obtained in the parallel version of local hidden-line removal is slightly distant from the ideal one. Even though results at local hidden-line removal are not ideal, they are still better than the previous implementation. In addition to that, local hidden-line removal is executed only once, and after that, it is not necessary to compute it again. The results obtained in global hidden-line removal are closer to the ideal ones than the ones obtained in local hidden-line removal because global hidden-line removal has fewer mutexes and a higher level of granularity than local hidden-line removal.

In the case of collision detection, the results are also satisfactory and the obtained speedup is better than the one obtained in hidden-line removal. However, the Cadmatic collision detection has some corner cases when spatially two large objects are close to each other but are not colliding and the box test does not recognize that, then the recursion needs to do a lot of work to figure out that there are no collisions.

It seems that spheres collision tests are behind this problem, providing fast collision results when primitives are distant but requiring extra time when primitives are close to each other.

One of the side effects of using multi-threading to speed up hidden-line removal and collision detection is an increase in the total memory required by both algorithms. Each thread needs to have its own local variables and temporal buffers to store its computations. In the case of both algorithms memory is not typically a problem, so an increment in the use of memory proportional to the number of threads and a constant factor of the input size does not represent an issue.

6 Conclusion and Future Work

The parallelization of complex and data-intensive algorithms is an arduous task prone to errors. In addition to that, parallelizing and tailoring each algorithm of each specific application is a time-consuming task and its implementation is domain-specific because it can not be reused outside the specific problem in which the algorithm is defined.

General and efficient parallelization techniques and approaches were defined in this thesis in order to provide reusable parallelization patterns that can be extrapolated and applied to other different algorithms in order to evolve sequential code into multi-thread code in an efficient and consistent way.

Hidden-line removal and collision detection algorithms were used as examples in which these parallelization techniques could be applied. It is important to point out that both algorithms are difficult to parallelize due to the asymmetry of their data. In the case of hidden-line removal, different objects (*segments*) have different complexities, requiring shorter or longer time than others in order to remove hidden lines. In the case of collision detection, comparisons between primitives using spheres in the collision test yield bad performance results when spatially two large objects are close to each other but are not colliding and the box test does not recognize that. A competitive approach, in which threads compete which each other for the next element to compute was applied, minimizing data asymmetry and obtaining better performance and work-load distribution between threads.

The obtained results showed that the proposed principles and patterns can be easily applied to both algorithms, transforming their sequential to multithread implementations, obtaining consistent optimization results proportional to the number of processing elements.

From the work done in this thesis, it is concluded that the suggested parallelization patterns warrant further study and development in order to extend their usage to heterogeneous platforms such as a GPU. OpenCL is the most feasible framework to explore in the future due to its interoperability among different platforms.

References

[1] M. M. Lehman, “Programs, life cycles, and laws of software evolution”, *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980. DOI: 10.1109/PROC.1980.11805.

[2] B. Catanzaro and K. Keutzer, “Parallel computing with patterns and frameworks”, *XRDS*, vol. 17, no. 1, pp. 22–27, Sep. 2010, ISSN: 1528-4972. DOI: 10.1145/1836543.1836552. [Online]. Available: https://doi.org/10.1145/1836543.1836552.

[3] L. Coyle, M. Hinchey, B. Nuseibeh, and J. Fiadeiro, “Guest editors’ introduction: Evolving critical systems”, *IEEE Computer*, vol. 43, pp. 28–33, May 2010. DOI: 10.1109/MC.2010.139.

[4] B. Massingill, T. Mattson, and B. Sanders, “Reengineering for parallelism: An entry point into plpp for legacy applications”, *Concurrency and Computation: Practice and Experience*, vol. 19, pp. 503–529, Mar. 2007. DOI: 10.1002/cpe.1147.

[5] A. Meade, J. Buckley, and J. Collins, “Challenges of evolving sequential to parallel code: An exploratory review”, Sep. 2011, pp. 1–5. DOI: 10.1145/2024445.2024447.

[6] S. Marschner and P. Shirley, *Fundamentals of Computer Graphics*. 2008, p. 242, ISBN: 9788170088547.

[7] A. Munshi, “OpenCL 1.2 Specification”, *Version 1.2*, p. 380, 2012. [Online]. Available: `http://scholar.google.com/scholar?hl=en%7B%5C&%7DbtnG=Search%7B%5C&%7Dq=intitle:The+opencl+specification%7B%5C#%7D0`.

[8] R. S. D. Melo, “A study of replacing CUDA by OpenCL”, 2014.

[9] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, “Timegraph: GPU scheduling for real-time multi-tasking environments”, *Proceedings of the 2011 USENIX Annual Technical Conference, USENIX ATC 2011*, pp. 17–30, 2019.

[10] Cadmatic, *Cadmatic Marine and Plant Industries*, 2021. [Online]. Available: `https://www.cadmatic.com`.

[11] A. Lew and H. Mauch, *Dynamic Programming: A Computational Tool.* Jan. 2006.

[12] R. Galimberti, “An algorithm for hidden line elimination”, *Communications of the ACM*, vol. 12, no. 4, pp. 206–211, 1969, ISSN: 15577317. DOI: `10.1145/362912.362921`.

[13] P. Jiménez, F. Thomas, and C. Torras, “3D collision detection: A survey”, *Computers and Graphics (Pergamon)*, vol. 25, no. 2, pp. 269–285, 2001, ISSN: 00978493. DOI: `10.1016/S0097-8493(00)00130-8`.

[14] A. Sohn, M. Sato, N. Yoo, and J. L. Gaudiot, “Data and workload distribution in a multithreaded architecture”, *Journal of Parallel and Distributed Computing*, vol. 40, no. 2, pp. 256–264, 1997, ISSN: 07437315. DOI: `10.1006/jpdc.1996.1262`.

[15] A. Fallis, *C++ Concurrency in Action*, 9. 2013, vol. 53, pp. 1689–1699, ISBN: 9788578110796. DOI: `10.1017/CBO9781107415324.004`. arXiv: `arXiv:1011.1669v3`.

[16] M. Haghbayan, S. Teräväinen, A. Rahmani, P. Liljeberg, and H. Tenhunen, "Adaptive fault simulation on many-core microprocessor systems", in *In Proc of International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*, 2015, pp. 151–154.

[17] T. Ungerer, B. Robič, and J. Šilc, "A survey of processors with explicit multithreading", *ACM Comput. Surv.*, vol. 35, no. 1, pp. 29–63, Mar. 2003, ISSN: 0360-0300. DOI: 10.1145/641865.641867. [Online]. Available: https://doi.org/10.1145/641865.641867.

[18] G. Glaeser, "Hidden-line removal", in *Fast Algorithms for 3D-Graphics*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 185–200, ISBN: 978-3-662-25798-2. DOI: 10.1007/978-3-662-25798-2_7. [Online]. Available: https://doi.org/10.1007/978-3-662-25798-2_7.

[19] P. P. Loutrel, "A Solution to the Hidden-Line Problem for Computer-Drawn Polyhedra", vol. C, no. 3, pp. 407–415, 1970.

[20] F. Dévai, "QUADRATIC BOUNDS FOR HIDDEN-LINE ELIMINATION", *2nd Symposium on Computational Geometry 1986: Yorktown Heights, NY, USA*, pp. 269–275, 1986.

[21] A. Appel, "The notion of quantitative invisibility and the machine rendering of solids", pp. 387–393, 1967. DOI: 10.1145/800196.806007.

[22] C. Hornung, "An approach to a calculation-minimized hidden line algorithm", *Computers and Graphics*, vol. 6, no. 3, pp. 121–126, 1982, ISSN: 00978493. DOI: 10.1016/0097-8493(82)90005-X.

[23] F. Dévai, *An O (log N) Parallel Time Exact Hidden-Line Algorithm.* 1988, pp. 65–73.

[24] John H. Reif and Sandeep Sen, "An efficient output-sensitive hidden-surface removal algorithm and its parallelization", *Proceedings of the 4th Annual Symposium on Computational Geometry*, pp. 193–200, 1988.

[25] M. Teschner, S. Kimmerle, B. Heidelberger, G. Zachmann, L. Raghupathi, A. Fuhrman, M.-p. Cani, N. Magnenat-thalmann, W. Strasser, and L. Raghu-, "Collision Detection for Deformable Objects", *Eurographics 2004*, pp. 119–135, 2009.

[26] A. P. del Pobil, M. Perez, and B. Martinez, "Practical approach to collision detection between general objects", *Proceedings - IEEE International Conference on Robotics and Automation*, vol. 1, no. April, pp. 779–784, 1996, ISSN: 10504729. DOI: 10.1109/robot.1996.503868.

[27] S. Trenkel, R. Weller, and G. Zachmann, "A benchmarking suite for static collision detection algorithms", *15th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision 2007, WSCG'2007 - In Co-operation with EUROGRAPHICS, Full Papers ProceedingsWSCG Proceedings*, pp. 265–270, 2007.

[28] C. Ericson, *Real-Time Collision Detection*. 2004, ISBN: 1558607323.

[29] M. Raynal, *Concurrent Programming : Algorithms , Principles , and Foundations*. Springer Science & Business Media, 2013, ISBN: 9783642320262.

[30] R. H. B. Netzer and B. P. Miller, "What are race conditions? some issues and formalizations", *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 1, pp. 74–88, Mar. 1992, ISSN: 1057-4514. DOI: 10 . 1145 / 130616 . 130623. [Online]. Available: https://doi.org/10.1145/130616.130623.

[31] M. Walmsley, "Multi-threaded programming in c++", Jan. 2000. DOI: 10 . 1007/978-1-4471-0725-5.

[32] Y. Ling, T. Mullen, and X. Lin, “Analysis of optimal thread pool size.”, *Operating Systems Review*, vol. 34, pp. 42–55, Apr. 2000. DOI: 10.1145/346152.346320.