Bachelor Thesis:

CUDA Geometry Sensor Service

Handed in: 22.10.2008

by

Christopher Bolte

Research Group Programming Languages / Methodologies
Dept. of Computer Science and Electrical Engineering
University of Kassel
Kassel, Germany

Thesis advisor:
Dipl.-Inform. Björn Knafla

Supervisor:
Prof. Dr. Claudia Fohry
Prof. Dr.-Ing. Dieter Wloka

UNIKASSEL
VERSITÄT

Selbständigkeitserklärung

Hiermit versichere ich, die vorliegende Arbeit selbständig, ohne fremde Hilfe und ohne Benutzung andere als der von mir angegebenen Quellen angefertigt zu haben. Alle aus fremden Quellen direkt oder indirekt übernommenen Überlegungen sind als solche gekenntzeichnet. Die Arbeit wurde noch keiner anderen Prüfungsbehörde in gleicher oder ähnlicher Form vorgelegt.

Frankfurt, den 18.10.2008

Christopher Bolte

Contents

1 Introduction **1**

2 Background **3**

2.1 Computer Games 3
2.2 Agent Model 3
2.3 Visual Agent Sensing 4
2.4 Compute Unified Device Architecture (CUDA) 9

3 The Sensor Service **14**

3.1 Scenario for the Sensor Service 14
3.2 The Sensor Service Framework 14
3.3 Visualization 16

4 The Sensing Algorithm **18**

4.1 Performance Techniques 18
4.2 Data-Parallel Algorithms 20
4.3 The Algorithm 23

5 Results **30**

5.1 Scaling per Sensor Count 30
5.2 Scaling for used Sensor Types 33
5.3 Scaling with the Number of used Threads 34
5.4 Comparison with other similar Algorithms 35

6 Conclusion **36**

Acknowledgment **A**

List of Figures **B**

Bibliography **D**

Chapter 1

Introduction

Todays computer games have become sophisticated real-time simulations, with nearly photo like graphics, realistic physics and believable artificial intelligence (AI). In the past, the main development focus laid on improving the graphics, but this is changing more and more in favor of sophisticated physics and AI.

A common way to implement the AI is to give each object which should have some sort of intelligent behavior its own decision making process. This is accomplished by giving each one a sensor to scan its surroundings. These objects, known as *agents*, are then deciding their next actions based on the sensed information. This approach is know as the *agent model* in AI literature [Dal03a].

Such a sensor can be modeled as a range check were all entities in a diameter around the scanning agent are considered visible. This thesis takes this approach a step further by providing a more realistic sensor which models a field of view (FOV), as shown in figure 1.1, for each agent. The algorithm for this is based on ideas from collision detection [Eri05a] and view frustum culling [Dal03b].

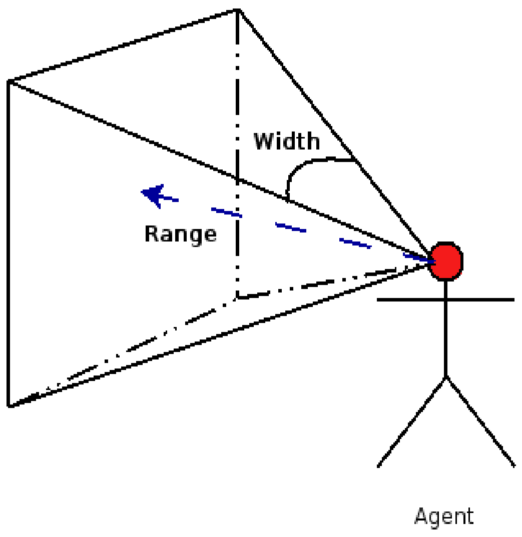


Figure 1.1: An Agent´s Field of View

The scanning algorithm is implemented with NVIDIA‘s Compute Unified Device Architecture (CUDA). This platform allows easy utilizing of the parallel processing power of current generation graphical processing units (GPU). This makes it possible to support a high number of agents, so that the algorithm can perform the visible scan for 20,000 agents in 50 milliseconds.

To provide easy usage of this sensing algorithm, a prototype service, which handles agent sensing, is developed; the *geometry sensor service*. This service provides an interface to customize the FOV parameter of each agent as well as querying the results of the agent sensing algorithm. On top of this, a visualization is provided to verify the correctness of the algorithm graphical.

This thesis starts with an introduction of the background knowledge in chapter 2. Chapter 3 gives an example where the sensor service is applicable and how it´s used. The sensing algorithm is then covered in detail in chapter 4. Moreover, chapter 5 publishes the results which are achieved with this implementation. The thesis closes with chapter 6, which provides a final summary of the implementation and a prospect about what further work could be done.

Chapter 2

Background

This chapter covers the background information required to implement the agent sensing. Foremost, section 2.1 explains how a typical game is implemented and which requirements a game imposes on its components. Thereafter, section 2.2 focuses on the agent model which is used in many game artificial intelligence subsystems. Furthermore, the mathematical knowledge to implement an agent´s sensing component (the so called field of view tests) is explored in section 2.3. Then section 2.4 covers an introduction to CUDA and the advantage of using CUDA instead of a classical CPU based solution.

2.1 Computer Games

Computer games are best described as interactive real-time simulations. This implies that a game reacts without noticeable delay to user inputs. To keep this impression, a game must update its state (entity positions, player health, etc) around 24 to 60 times per second (frames per second in game terminology, the standard timing metric). Thus there is only a time-frame of $\frac{1}{60}$ up to $\frac{1}{24}$ second per *main loop* step (the core loop of a game which calls all per frame update functions). *Main loop* operations include, but not exclusively, letting the AI system think about its following actions, updating the positions of all game entities through some physics formulas and computing the next frame to display.

2.2 Agent Model

The *agent model* is a common way to model an artificial intelligence (AI) system in games. This model considers each game entity with some intelligent behavior as an *agent*. These agents are making their own decisions through a sense-think-act process. To model this process each agent consists of at least the following four components [Dal03a]:

Sensing is responsible to provide the agent with informations about the state of game objects near it.

Memory is an optional component used to remember previous decision. This allows older choices to influence the current decision to ensure a consistent behavior.

Reasoning is the core component of an agent's decision making process. The informations from the sensing and memory components are evaluated to identify the next actions to be taken.

Output is used to execute the decisions made by the reasoning component.

2.3 Visual Agent Sensing

To provide human like sensing, one solution is to provide each agent with a visual sensor. This sensor defines a customizable view frustum to represent the agent‘s field of view. Then, whenever an agent collides with another agents view frustum it is considered visible. For a more efficient collision test, agents are represented as primitive geometry volumes. These are called *bounding volumes* and are discussed in detail later in this section. Determining if an object collides with a view frustum is also a problem in graphic rendering were it is used to find all objects which must be drawn on the screen [Dal03b].

The a view frustum is modeled by a set of planes which define a enclosed volume in space. Figure 2.1 illustrates a view frustum in 2D.

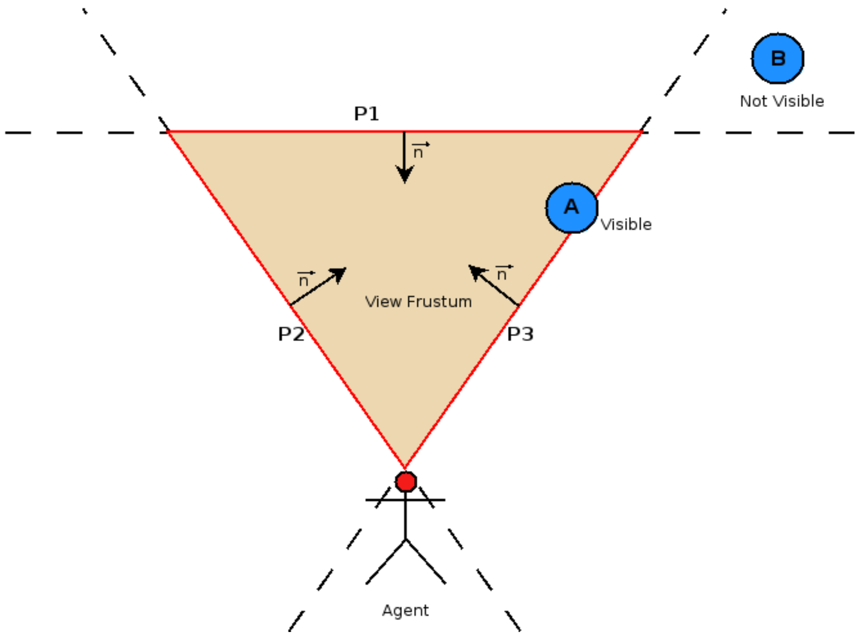


Figure 2.1: 2D View Frustum

Each plane of the view frustum posses a normal vector $\vec{n}$ which defines the front and back of it. As observable in figure 2.1, object A lies in front of plane P_1 and P_2 as well as intersecting with P_3 and so it can be considered visible. On the other hand, object B lies in front of plane P_2 but is behind plane P_1 and P_3 and therefore not visible. Thus the condition that an object is visible is defined as being not behind any view frustum plane. Since the view frustum culling is a visible test all partly visible objects are also considered visible. The same principle is applicable to 3D view frustums.

2.3.1 Planes

All view frustum tests are based on evaluating the position of a point relative to a plane. These planes are defined by a normal unit vector $\vec{n}$ (a vector standing perpendicular on the plane with the length one) and the distance of the plane to the origin of the coordinate system along the plane´s normal. This form is called the *hessian normal form* [Str97].

To determine the position of a point $\vec{v}$ regarding a plane, $\vec{v}$ is projected onto the plane´s normal $\vec{n}$. This projection is compared with the plane´s distance d to the origin. This leads to equations 2.1 [Str97].

$$(\vec{n} \cdot \vec{v}) - d \begin{cases} = 0 & \text{point lies on the plane} \\ > 0 & \text{point lies infront of plane} \\ < 0 & \text{point lies behind the plane} \end{cases} \tag{2.1}$$

2.3.2 Bounding Volumes

To evaluate the position of a complex object regarding a plane, it is necessary to test all vertices (points in space describing the objects hull) of it. Besides this computational complexity, such an approach can also be unsufficient for non-convex objects as shown in [Eri05b]. By using more primitive geometric objects to enclose the complex objects, it's possible to reduce the computation complexity and so speed up the object-plane test. This comes with the loss of some precession. This loss depends on how correct an object can be represented as a simpler one. Such simple objects are called *bounding volumes* [Eri05b] and include a variety of shapes with different drawbacks and advantages. This thesis uses spheres and special boxes (axis aligned bounding boxes) as bounding volumes.

2.3.3 Spheres

The *sphere* is the most basic bounding volume. It features the smallest memory requirement and fastest plane test over the more advanced bounding volumes. But, as shown in figure 2.2, a sphere can be a very inaccurate hull.

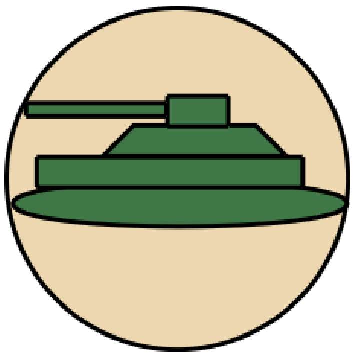
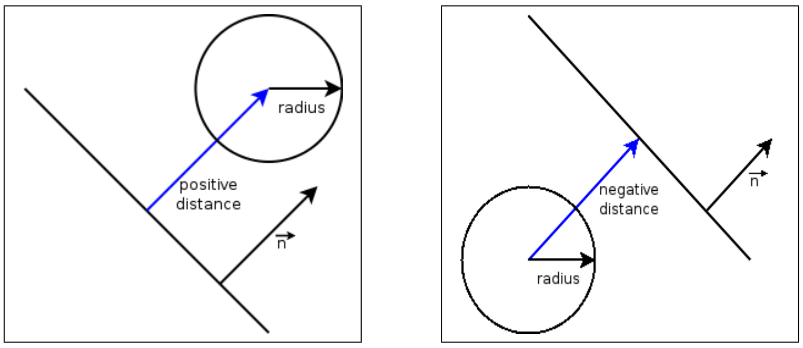


Figure 2.2: Sphere as Bounding Volume

To determine the position of a sphere regarding a plane, the sphere´s center must be projected onto the plane´s normal vector. This yields a signed distance to the plane. This distance is positive when the center is in front of the plane and negative if behind. Then the radius of the sphere is added to the signed distance which leads to the following two cases:

a)	distance + radius	=>	0	⇒	Sphere in front of plane or intersecting it
b)	distance + radius	<	0	⇒	Sphere behind plane

Figure 2.3 illustrates these cases in 2D. But the same principle is applicable in 3D.



(a) Sphere in front of Plane (b) Sphere behind the Plane

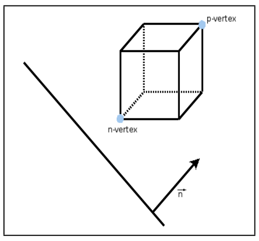
Figure 2.3: Sphere Plane Test

2.3.4 Axis Aligned Bounding Box

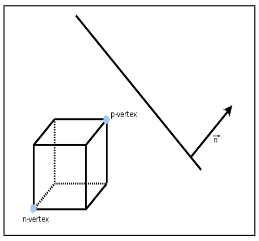
Axis Aligned Bounding Boxes (AABB) are another type of *bounding volume*. These are boxes which have each side aligned to the world's coordinate system. This allows storing the AABB without orientation informations.

An AABB can be represented in various ways [Eri05b]. This thesis uses the half-extends approach. An half-extend describes how far a side is away from the AABB's center. Since both sides are symmetric, it's sufficient to have one half-extend for each dimension. These are stored as a single vector besides the vector for the AABB's center. Hence an AABB needs only two vectors and can be stored efficiently in memory.

To evaluate the position of an AABB relative to a plane, it's necessary to find the *p* and *n* vertices. These are defined as the farthest vertices of the AABB along the positive and negative direction of the plane´s normal. When these are found it´s sufficient to check the position of both. When both lie in front of the plane the AABB is in front of it. The same goes for behind the plane. To compute these vertices the half-extends are projected onto the normal. Figure 2.4 illustrates these vertices.



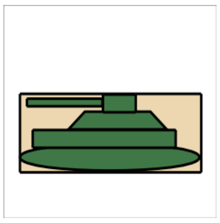
(a) AABB in front of Plane



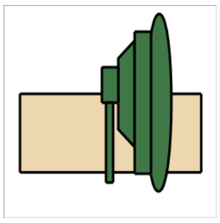
(b) AABB behind the Plane

Figure 2.4: AABB n,p-Vertices

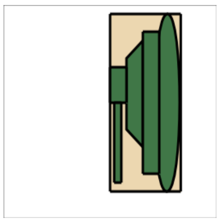
Since enclosed objects can be rotated, it's necessary to adjust the AABB after each rotation to prevent precession loss. Figure 2.5(a) shows the object before rotation where the AABB encloses it accurately. After the rotation in figure 2.5(b) the AABB has lost it's precession and so it must be recomputed as in figure 2.5(c).



(a)



(b)



(c)

Figure 2.5: AABB as Bounding Volume

Some other problems can occur when using AABBs for view frustum culling. In some situations, it´s possible that an AABB is not colliding with a view frustum but a culling test against all frustum planes reports otherwise (a *false positive*). As illustrated in figure 2.6, the AABB has always a point which lies in front of a plane but the whole AABB doesn't collide with the view frustum. Since such situations happen rarely in 3D and handling of these would require a lot of extra computation, the implemented algorithm ignores these cases.

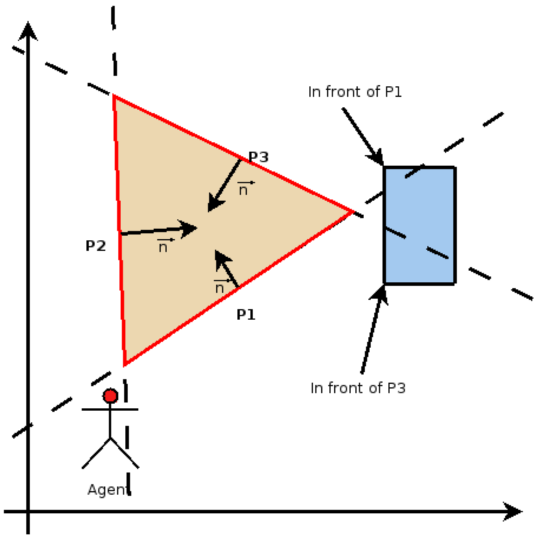


Figure 2.6: AABB View Frustum Culling Special Case

2.3.5 Regular Grid

Besides the plane test to determine visibility, FOV tests have to deal with another complexity. In theory, each agent might be able so see each other agent, so that all possible combinations must be considered. This yields a complexity of $O(n^2)$. Thus various techniques were developed to reduce this complexity. All these are based on the idea of spatial sorting to exclude object combinations which are too distant. A *regular grid* is one of these techniques and is used in this thesis.

The *regular grid* divides the world into cells of the same size. This size is chosen so that each cell is at least as big as the biggest object. This ensures that no object can span over a whole cell. This reduces the maximum cells in which an object lies to the number of shared corners, yielding four cells in 2D and eight in 3D. To allow a unambiguous mapping of each point to a cell the world must not be infinity.

Figure 2.7 shows a simple 2D world with three objects in a regular grid. With a native approach each object would be tested against each other to determine if they overlap and so yielding six overlapping tests. By utilizing a regular grid, each object just has to be tested against all objects which share a cell with it. So Object A doesn't need

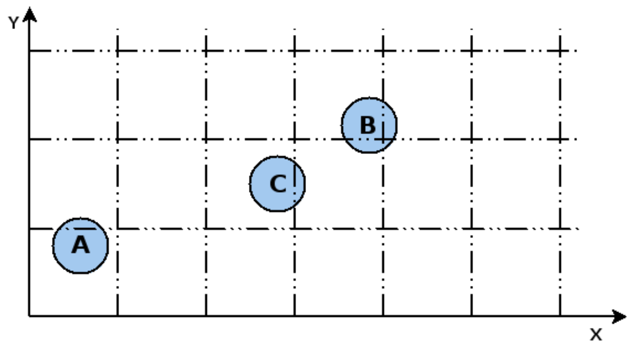


Figure 2.7: Using a Regular Grid for Spatial Sorting

any test at all and just the combinations (Object B, Object C) and (Object C, Object B) have to be evaluated. If the combination has commutative attributes the number of tests can be further reduced to one test.

To utilize a regular grid, it´s necessary to compute the coordinates of the cell in which a point lies. This is achieved by dividing all coordinates through the size of a cell (all cells are cubic and so the size is equivalent to the cell´s side length). This yields equation 2.2 which can be seen as a projection from world space to grid space.

$$\mathrm{cell}\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} \left\lfloor \frac{x}{\mathrm{cellsize}} \right\rfloor \\ \left\lfloor \frac{y}{\mathrm{cellsize}} \right\rfloor \\ \left\lfloor \frac{z}{\mathrm{cellsize}} \right\rfloor \end{pmatrix} \tag{2.2}$$

Because this thesis uses view frustums, it's possible that an object sees another object but not the other way around, so the test is non-commutative. To determine the minimum required cell size with view frustums, an conservative approach is used. The view frustum's range is used as a radius for a sphere around the agent's center. This sphere enclose the whole view frustum independent of it's orientation. The biggest sphere is then treaded as the biggest object and so defines the minimum cell size.

2.4 Compute Unified Device Architecture (CUDA)

The Compute Unified Device Architecture (CUDA) is NVIDIA´s platform for General Purpose computing on Graphics Processing Unit (GPGPU) programming. This platform defines how the hardware behaves and how a function is executed on the GPU. These GPU functions are called *kernel* in CUDA terminology. A *kernel* is developed with a variation of the C-language which has some GPU specific extensions.

2.4.1 Motivation for GPGPU Programming

Graphics Processing Units (GPU) were developed for fast graphics rendering. Computer graphics are defined by points (vertices) as well as faces and operations (for example rotations, transformations and color computing) which are applied to these. These operations can be applied to each vertex individually without interfering with other vertices and so these can be applied in parallel [CUDa].

This resulted in a GPU hardware design optimized for a high throughput of branching-light code with high arithmetic density for each data element. CPUs, on the other hand, are traditionally designed for high throughput of branching-heavy code on small data sets. As figure 2.8 illustrates, GPUs have, by these design decisions, improved dramatically over CPU in theoretical maximum computing power.

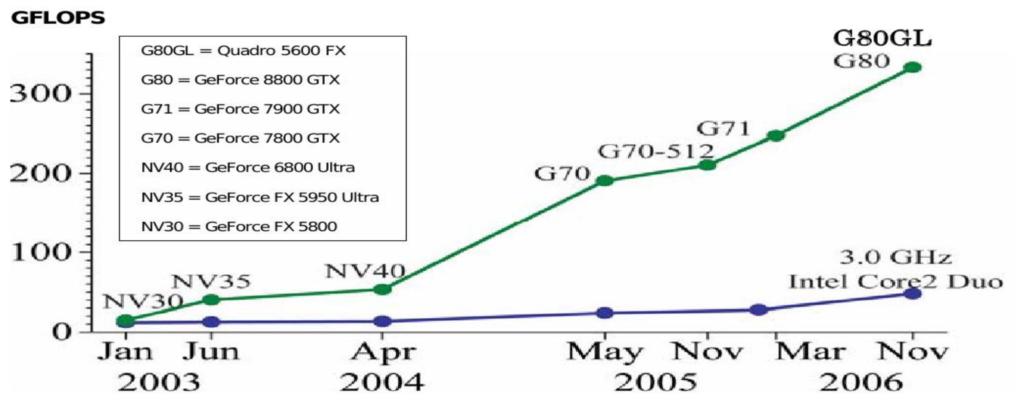


Figure 2.8: GPU and CPU GFLOPS/s [CUDa]

To utilize such computing power for more general problems, the idea of GPGPU programming was developed. The first GPGPU programs utilized graphic APIs like OpenGL [Shr07] to solve general non-graphics problems. This approach had the disadvantage of using an API not well suited for this problem domain and so lead to a high learning curve and complicated programming. To overcome this, the major graphic card manufactures started to develop platforms for easier GPGPU programming. CUDA, which is used in this thesis, is NVIDIA´s approach.

2.4.2 Hardware Layout

The highly data parallel graphic rendering requires a hardware layout to support such data-parallelism. Hence CUDA compatible GPUs are built as a set of multiprocessors. Each of these consists of eight computing cores. To store the data to process a GPU possesses a main memory (the *global memory*) which is accessible by all computing cores as well as a fast but small on-chip memory (the *shared memory*) for each multiprocessor.

2.4.3 Execution Model

CUDA distinguishes between two types of execution, *host*, which addresses programs run by the CPU, and *device*, which means execution of a kernel by the GPU. The kernel is executed individually by each core as a light-weight thread in parallel. Since each kernel performs its computation on different data this execution scheme resembles single instruction multiple data - SIMD.

These threads are distributed over the processors as *thread-blocks*. When a kernel is executed, the programmer defines how many blocks are created and the number of threads per block. Then the blocks are queued over the processors till all blocks are processed as shown in figure 2.9. Also a block is assigned exclusively to a multiprocessors. This allows using the shared memory to communicate between threads in a block. On the other side, no communication or synchronization between different blocks is possible.

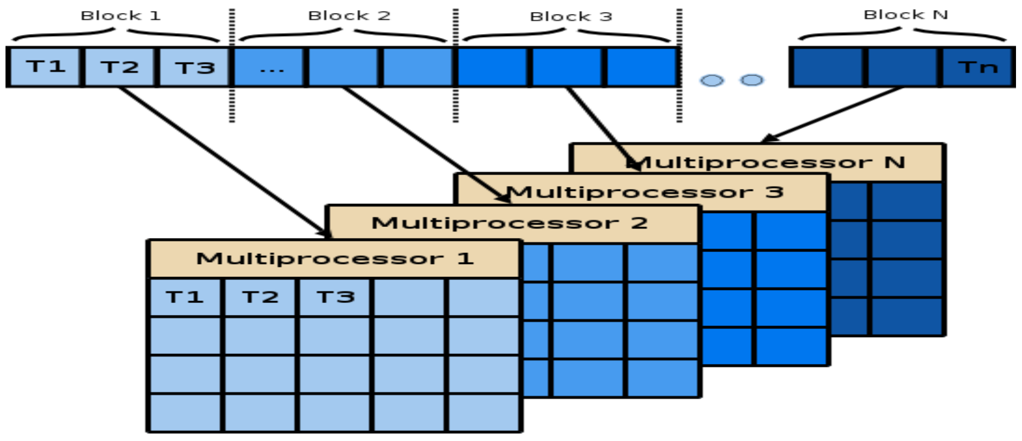


Figure 2.9: Mapping of CUDA Threads to GPU Multiprocessors

Moreover these blocks are divided into *thread warps* (sets of threads of the same size) which are scheduled on the cores of the multiprocessor. This allows hiding of memory access latency by computations of other *thread-warps*.

As stated earlier, only threads in a block can be synchronized. This can be achieved by *barriers* which force all threads of a block to wait till all threads have reached the barrier. The only way to synchronize all threads in a kernel is to divide the operation in two kernels and execute one after another (CUDA guarantees that a kernel only starts executing when a currently executing kernel has finished).

2.4.4 Memory Hierarchy

To classify memory, CUDA uses the same terminology as with the execution model. *Host* memory is the system´s main memory and *device* memory is located on the

GPU. To further classify the device memory, CUDA defines a variety of memory types which are shown in table 2.1.

Table 2.1: CUDA memory types

Name	Latency	Size	GPU access
register	4	8148 per block	Read/Write per thread
shared	4	16Kb per multiprocessor	Read/Write per block
global	400 - 600	GPU dependent, many MB	Read/Write for all
local	400 - 600	GPU dependent	Read/Write per thread
texture	4 - 600(cached)	GPU dependent	Read-only for all
constant	4 - 600(cached)	64kb per GPU	Read-only for all

Data transfer between host and device memory is only possible from the host and not all types of memory are accessible. A CUDA kernel has no possibility to access the system's main memory and is restricted to GPU memory.

All these CUDA memory types have some characteristics which must be considered to achieve good performance. Here is an overview for which usage each type is most fitting:

registers are used for small per thread data like loop counters.

shared memory is used to store the current working data of a thread-block. This can be seen as a programmer controlled data cache.

global memory is basically the GPU´s main memory and used to transfer data between the CPU and CPU.

local memory is used as a fallback when a block runs out of register memory. This memory resides in global memory and should be avoided because it introduces high memory access latencies.

texture memory is an alternative for global memory. This memory is read-only and has it's own on-chip cache. It can be used to improve memory latency when global memory is accessed irregularly which would make prefetching of it hard or impossible.

constant memory is read-only for all threads and only writable from the host. Its purpose is to provide a fast storage for control variables.

For further optimization of on-device memory transfers, CUDA features implicit *memory transfer coalescing*. If consecutive threads of a block are each transferring a element of continuous global memory to continuous shared memory, CUDA can replace all single transfers with one block data transfer to improve memory transfer latency. Figure 2.10 illustrates memory transfer coalescing.

When using the shared memory, the special restrictions of this memory type must be taken care of. The shared memory is organized as sixteen banks which store successive

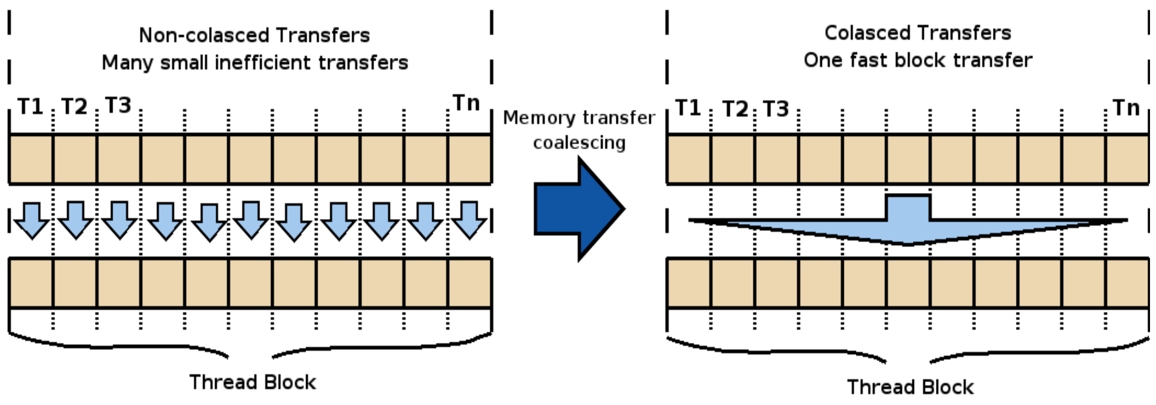


Figure 2.10: CUDA Memory Transfer Coalescing

words (32-bit of data) in successive banks. The access to these banks is mutual exclusive and so when more than one thread tries to access a bank the threads are serialized. It´s advisable to organize access to the shared memory in a way that each thread of a warp (a *bank conflict* can only happen in a warp since the multiprocessors executes only one at once) access a bank exclusively.

Chapter 3

The Sensor Service

This chapter introduces the prototype framework for the developed sensor service. This service provides a centralized framework to handle the sensing component of the *agent model*.

The chapter starts which a scenario which shows where the sensor service could be utilized in section 3.1. An introduction of how the sensor service is built and used follows in section 3.2. Thereafter the visualization is described in section 3.3

3.1 Scenario for the Sensor Service

Typical scenarios where the sensor service could be used include *steering simulations* [BK] and *computer games*. This section uses an imaginary computer game as an example.

The game used as an example is a strategy game in which two players fight over the control of a region. For this, each player can assemble robots from different components. The AI for each robot can be configured. When both player have set up their robot armies the fighting starts. During the fight the player has no control over the robots anymore, all actions are determined by each robot individually. To realize this, each robot's AI is represented as an *agent*. Since the robot battle should have an epic feeling, each side should control many hundreds and thousands of robots. To realize this in real-time, or nearly real-time, the sensor service could be used to offload the sensor part of each agent to the GPU. This would allow a higher framerate since the sensing test runs highly parallel on the GPU. Also some CPU time is freed up which could be used for other AI computations.

3.2 The Sensor Service Framework

To allow efficient GPU based agent sensing the service provides a prototype framework. It consists of a core component which handles creation of sensor objects as well as invocation of the sensing algorithm. This algorithm updates all viewing informations for each agent in a single update call.

3.2.1 Overview about the Sensor Service

To provide assignable visual sensors, a hierarchy of sensor objects is used. These sensors define a *bounding volume* for the agent attached to, allow customizing the view frustum and moving of the sensor. The sensing algorithm checks each sensors´s view frustum against all sensor´s *bounding volumes.* Figure 3.1 gives an overview about the services components.

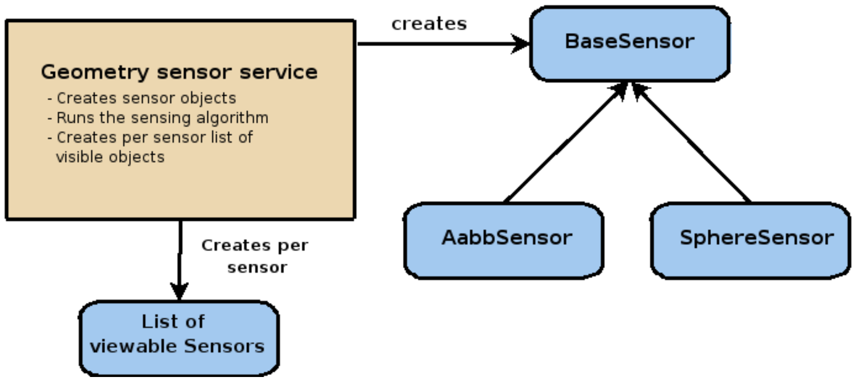


Figure 3.1: The Geometry Sensor Service

3.2.2 Usage of the Sensor Service

The usage of the sensor service consists of a few simple operations. First a sensor must be created and attached to each agent.

```
agent.assign_sensor( GeometryService::createSphereSensor(.));
// or
agent.assign_sensor( GeometryService::createAABBSensor(.));
```

To keep the sensors in sync with the represented agent, the position of the sensor must be updated whenever the agent updates it's state.

```
agent::update() {
    // some code
    sensor->set_position( my_position );
    sensor->set_view_direction( my_view_direction );
    // some other code
}
```

Then, somewhere in the main loop, the update function must be called to compute the viewing informations for each sensor.

```
// ...
GeometryService::update_sensor_data();
// ...
```

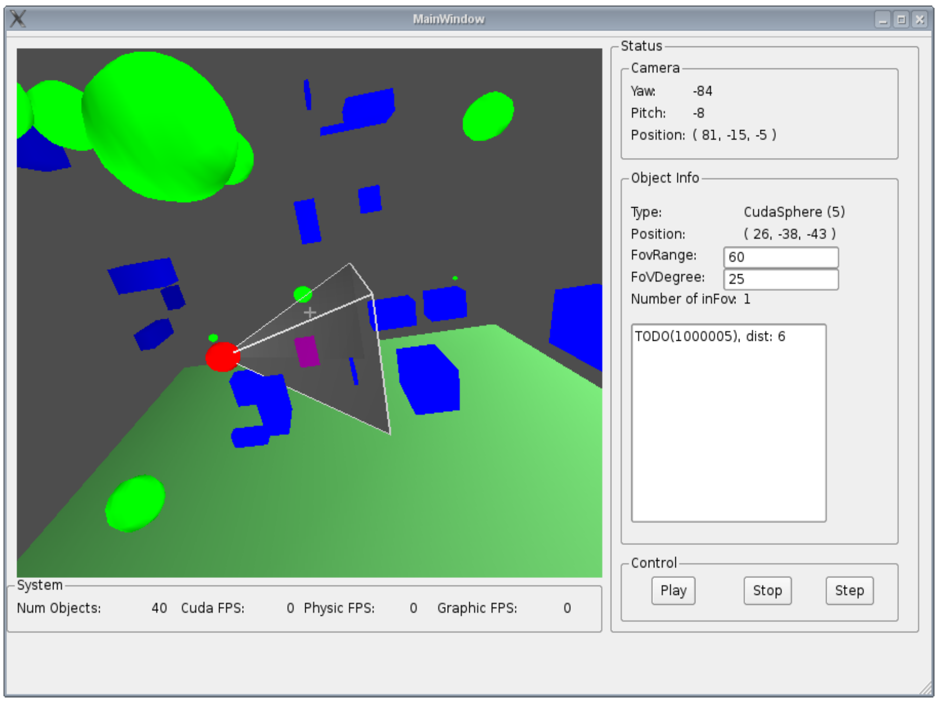
These information can be queried during the *think* phase of each agent.

```
agent::think_phase() {
    viewable_sensors = sensor->query_viewable();
}
```

Since all viewing informations are computed in a single function, all previous computed results become invalid by updating a sensor setting (like moving or customizing the sensor´s view frustum). Thus it´s necessary to run the sensing algorithm in each *main loop* step to ensure correct sensing informations.

3.3 Visualization

This section covers the visualization which was developed for graphical debugging and testing purposes.



Camera
Yaw: -84
Pitch: -8
Position: (81, -15, -5)

Object Info
Type: CudaSphere (5)
Position: (26, -38, -43)
FovRange: 60
FoVDegree: 25
Number of inFov: 1

TODO(1000005), dist: 6

System
Num Objects: 40 Cuda FPS: 0 Physic FPS: 0 Graphic FPS: 0

Figure 3.2: Geometry Sensor Service Visualization

To implement the visualization some third party libraries were used. These include the QT4 library [QT4] for implementing the graphical user interface (GUI), and an OpenGL [Shr07] *QT4 widget* to render the simulation. Also the *bullet continuous physics engine* [Bul] is used to implemented simple physics based agent movement.

The visualization can be used to customize the agent´s view frustum parameter like *view range*, *view width* and *orientation*. Also the correctness of the computed viewing information can be verified since the view frustum is drawn and all visible agents are colored purple. Besides this, a list of viewable agent´s is also shown.

Chapter 4

The Sensing Algorithm

This chapter covers the developed CUDA sensing algorithm which is used by the geometry sensor service. Since the algorithm should run as fast as possible, some techniques were applied to improve the performance. Some of these are introduced in section 4.1. This follows an introduction of some data-parallel algorithms in section 4.2, which are used to implement the various phases of the algorithm. The chapter closes with a detailed explanation of the developed sensing algorithm in section 4.3.

4.1 Performance Techniques

This section covers some performance techniques which are used to improve the runtime of the sensing algorithm.

4.1.1 Using aligned Structures

To optimize memory access, all structures are aligned to make use of CUDA's capability for 128-bit loads. With these a vector class (typically 16 byte) can be loaded in a single instruction instead of a load for each element. Figure 4.1 illustrates how this improved the performance for a small benchmark kernel which loads an array of 1,000,000 floats and computes the square for each float.

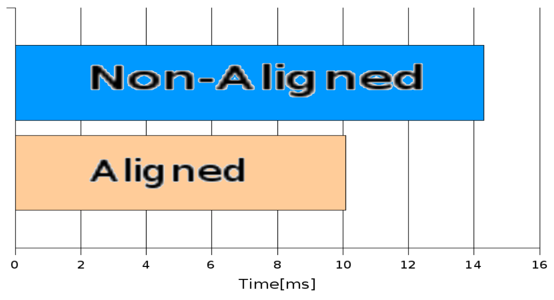


Figure 4.1: Performance of using Aligned Structures

4.1.2 CUDA Fast Heap

The sensing algorithm makes heavy use of temporary data during execution. So it's critical that such memory can be allocated fast. Since CUDA's memory allocation is relative slow, an own heap implementation was developed.

This heap preallocates a memory region and handles access to this memory, so only a single CUDA memory allocation is needed. The heap is optimized for temporary data which is allocated over a frame and freed at the end of it. This allows sparing sophisticated memory fragmentation prevention. By implementing the heap as a stack, each allocation only consists of simple pointer arithmetic. Figure 4.2 illustrates how the *CUDA fast heap* behaves during multiple memory allocations and deallocation. The memory hole there will be closed when the whole heap is freed at the end of the frame, or, when all higher blocks are deallocated.

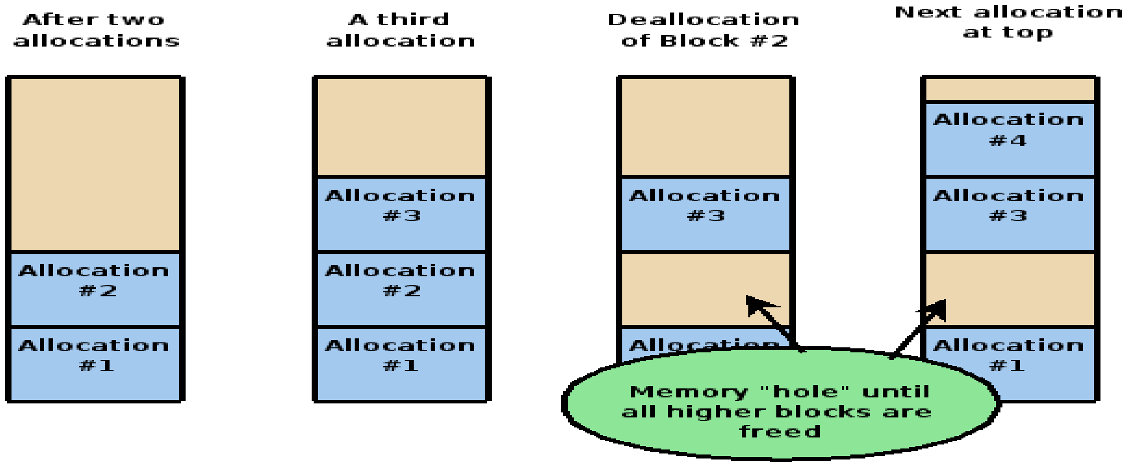


Figure 4.2: *CUDA fast heap* Memory Handling

This heap implementation improved memory allocation/deallocation dramatically. Figure 4.3 shows the runtime for a memory allocation with CUDA and a memory allocation with the CUDA fast heap.

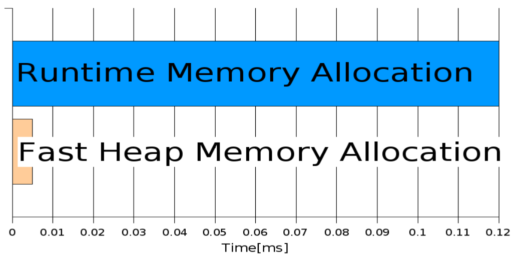


Figure 4.3: Comparison CUDA Runtime Memory Handling vs *CUDA fast heap*

4.2 Data-Parallel Algorithms

During development of the sensing algorithm, some data-parallel algorithms were needed at various parts. These algorithm were implemented as building blocks to be reuseable by the higher layers of the sensing algorithm. This section covers these algorithm in detail.

4.2.1 Foreach

Foreach is the most basic data parallel algorithm. It assigns a thread to each element of an array. Then the threads execute a function on their element and store the result back in the array or in a secondary result array. Figure 4.4 illustrates this algorithm.

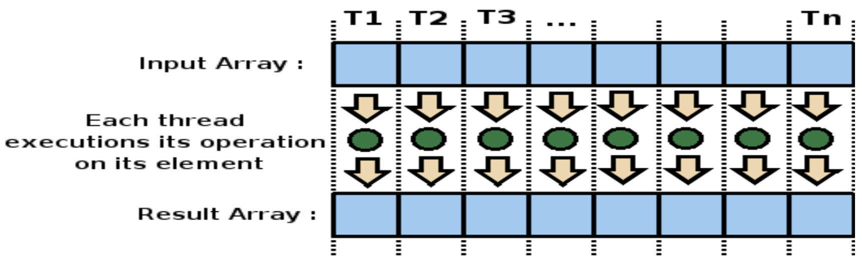
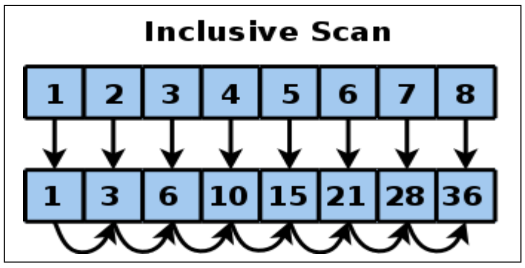


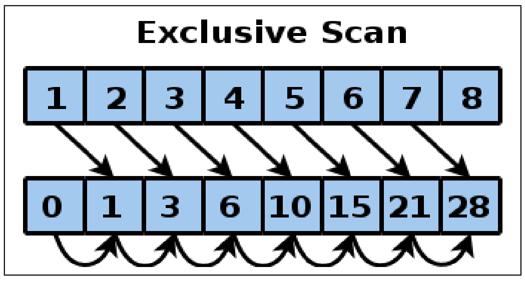
Figure 4.4: Foreach Algorithm

4.2.2 Scan

The *scan* algorithm (also known as *prefix-sum*) is an important building stone for more sophisticated data parallel algorithm. It takes an array of integer values and returns a new array in which each element holds the sum of all preceding elements. Furthermore, the *scan* pattern comes in two variations. The *exclusive scan*, which sums-up all preceding elements without the actual one and the *inclusive scan* which also adds the actual element. Figure 4.5 illustrates both variations.



(a) Inclusive Scan



(b) Exclusive Scan

Figure 4.5: Scan Algorithm

The CUDPP [CUDb] library is used to provide the scan algorithm. A full introduction of a CUDA scan implementation can be found at [MH07], since it would be out of the scope for this thesis.

4.2.3 Compare

The *compare* algorithm is used to compare adjacent pairs. Each thread is assigned an element which he compares with its predecessor. The result of the comparison is stored in a result array. This information can be used to find start and end indices of groups of similar data in an array. It´s also possible to build *offset pointer* (an integer pair which stores the start and end offset of a region in an array) with two *compare* passes and the *scan* algorithm as shown in figure 4.6.

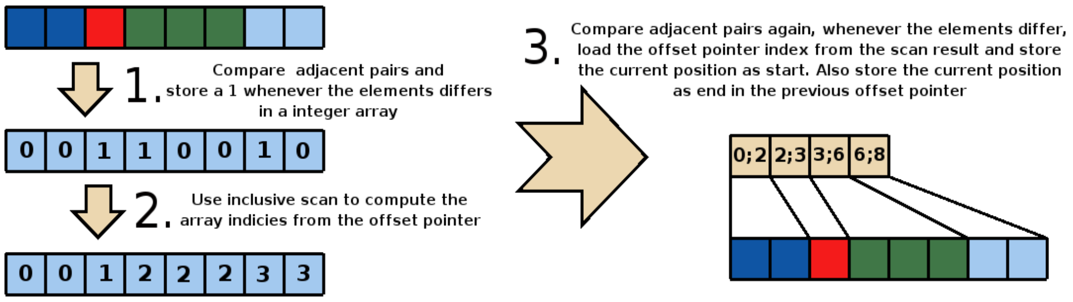

1. Compare adjacent pairs and store a 1 whenever the elements differs in a integer array
0 0 1 1 0 0 1 0
2. Use inclusive scan to compute the array indicies from the offset pointer
0 0 1 2 2 2 3 3
3. Compare adjacent pairs again, whenever the elements differ, load the offset pointer index from the scan result and store the current position as start. Also store the current position as end in the previous offset pointer
0;2 2;3 3;6 6;8

Figure 4.6: Computing Offset Pointer

4.2.4 Stream Compaction

Foreach can be used in combination with *scan* to remove elements from an array which are no longer needed; this is known as the *stream compaction* algorithm and is explained in figure 4.7

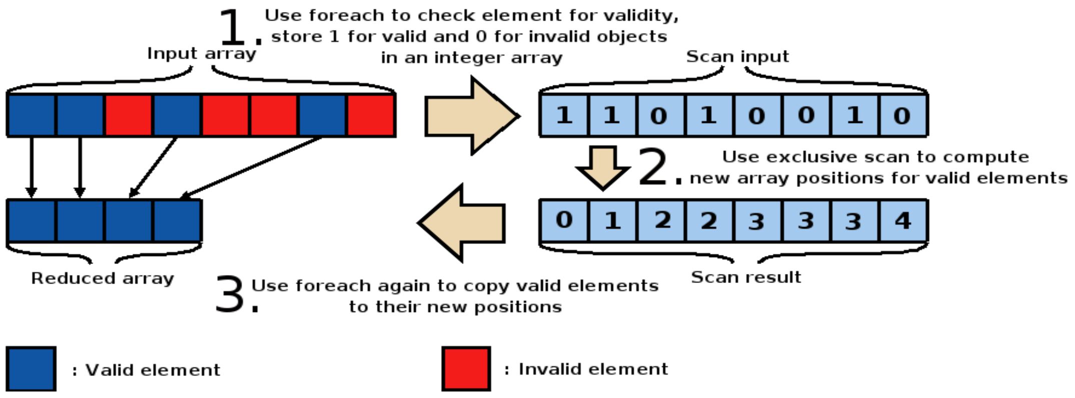

1. Use foreach to check element for validity, store 1 for valid and 0 for invalid objects in an integer array
Input array
Scan input
1 1 0 1 0 0 1 0
2. Use exclusive scan to compute new array positions for valid elements
0 1 2 2 3 3 3 4
Scan result
Reduced array
3. Use foreach again to copy valid elements to their new positions
: Valid element
: Invalid element

Figure 4.7: Stream Compaction

Another variant of this algorithm is to use *compare* instead of *foreach* to mark all adjacent duplicates for removal. When this algorithm is applied to a sorted array all duplicates are removed since all duplicates are adjacent.

4.2.5 GPU Sort

The choice of sorting algorithms available for GPGPU programming is restricted by the highly data parallel hardware design and the lack of thread synchronization as well as memory allocating during *kernel* execution. So classical recursive algorithms like quick sort aren't applicable and other sorting algorithms which produce a constant work for each element are needed. One of these is *radix sort* [Rad] which belongs to the class of *bucket sort* [Buc] algorithms. These use informations about the data type to reorder elements in so called "buckets".

Radix sort has some very CUDA friendly characteristics. It requires only inter-kernel synchronization and produces the same work-load for each element. This thesis uses the variant *global radix sort*. This variant sorts a whole array in n passes, where n is the number of bits in the sorting key. For this, all objects are grouped together by the current sorting bit while keeping the relative order of previous passes. This results in an sorted array when all bits are proceeded.

Besides global-radix-sort, other GPU sorting algorithm exists. There is *merge-radix-sort* which is used in CUDPP [CUDb] and *bitonic sort* which is introduced in [Bit]. Since bitonic sort is only applicable on small arrays, the choice was between merge-radix-sort and global-radix-sort. As shown in figure 4.8, the global-radix-sort variant scales better for big arrays and so was chosen for this thesis.

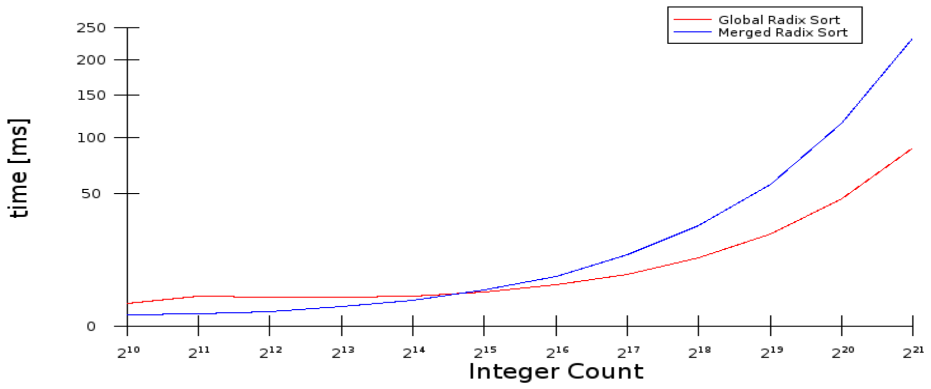


Figure 4.8: Runtime of Radix Sort, measured using CUDPP [CUDb]

4.3 The Algorithm

This chapter covers a detailed introduction of the developed sensing algorithm which is used in the geometry sensor service. This algorithm is implemented fully in CUDA and is built in three phases which are responsible for different steps of the algorithm. Figure 4.9 illustrates these phases.

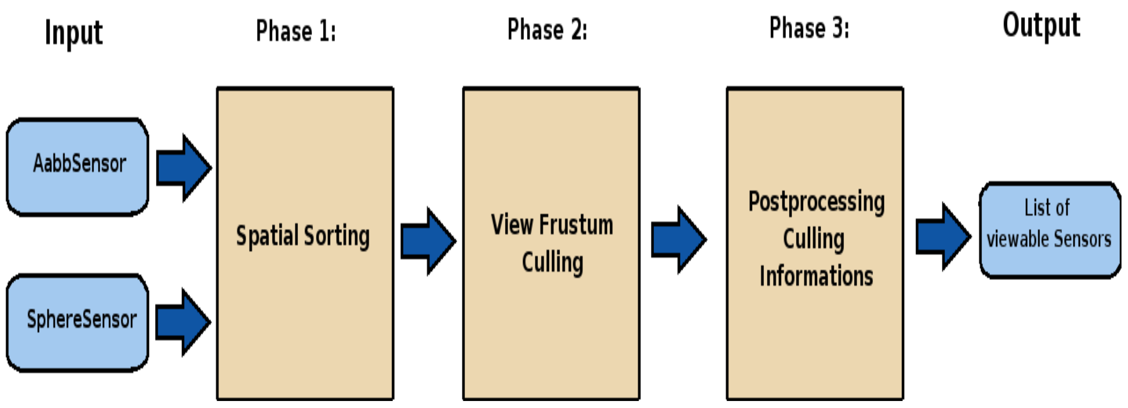


Figure 4.9: The Sensing Algorithm

These phases have the following purposes:

Spatial Sorting

- excludes pairs from the sensors which are too distant by utilizing a *regular grid*.

View Frustum Culling

- performs culling tests on each remaining pair from the *spatial sorting* phase

Post-Processing Culling Results

- uses the culling results to generate lists of viewable sensors for each agent.

The following sections covers the algorithm´s phases in detail.

4.3.1 Spatial Sorting

This phase excludes all sensor pairs which are too distant to see each other. The concept of the *regular grid* is used for this. The base idea of this technique is from [Gra07]. First, it´s examined which cells could be visible for each sensor. Then (cell; sensor) pairs are built which store a cell identifier and a sensor id. These pairs are then grouped together by their cell identifiers so that all sensors in the same cell are adjacent. Furthermore an array of *offset pointers* is built to access each cell directly.

Building the (cell; sensor) Pairs Array

To allow easier sorting by cells as well as a more memory efficient representation the cell coordinates are encoded as a *cell hash*. This hash is computed as in figure 4.10 and requires only 32-bit for all three cell coordinates as well as the sensor type (encoding the sensor type in the lowest bits provides some advantages when sorting the pairs array).

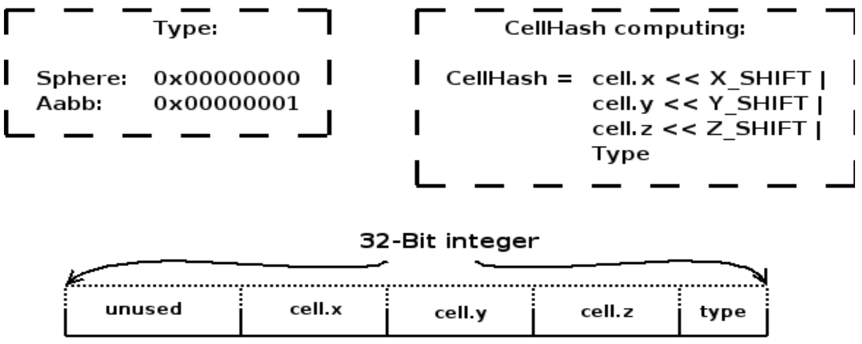


Figure 4.10: Computing a *Cellhash*

The cell hash computation requires only positive coordinates to work correctly, so each sensor is transformed that all sensor coordinates are positive.

For an efficient and deterministic computing of all cells in which a sensor's view frustum resides so called *viewing boxes* are used. These are AABBs with the sensors view range as half-extend on each axis. Figure 4.11 illustrates such a box.

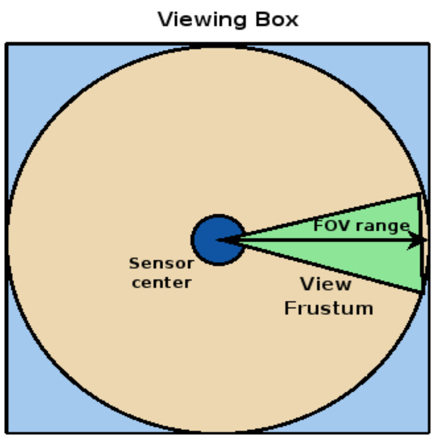


Figure 4.11: Computing a Viewing Box

Since the view range was also used to determine the minium cell size, it is guaranteed that a viewing box can only span over up to eight cells. Also, since the side length is smaller than the cell size, it is sufficient to check all eight corners of the viewing box. Then cell hashes are computed for each of these eight corners, yielding eight (cell;

sensor) pairs per sensor. Since this algorithm produces some duplicates, a *stream compaction pass* follows, which removes the adjacent duplicates and so removing the number of pairs by 30% to 50%.

Sorting the (cell; sensor) Pairs Array

To group the pair array by its cells, a sorting step is performed. By this, all sensors which share a cell are moved adjacent in the array. And, since the type is encoded in the lowest-bits of the cell hash, the objects are ordered per type in each cell.

In figure 4.12 such a sorting is illustrated for a simple 2D world. As shown there, the cells are grouped together and also the types are adjacent in the cells (for example *Object B* and *Object C* in cell(10,10)) after sorting.

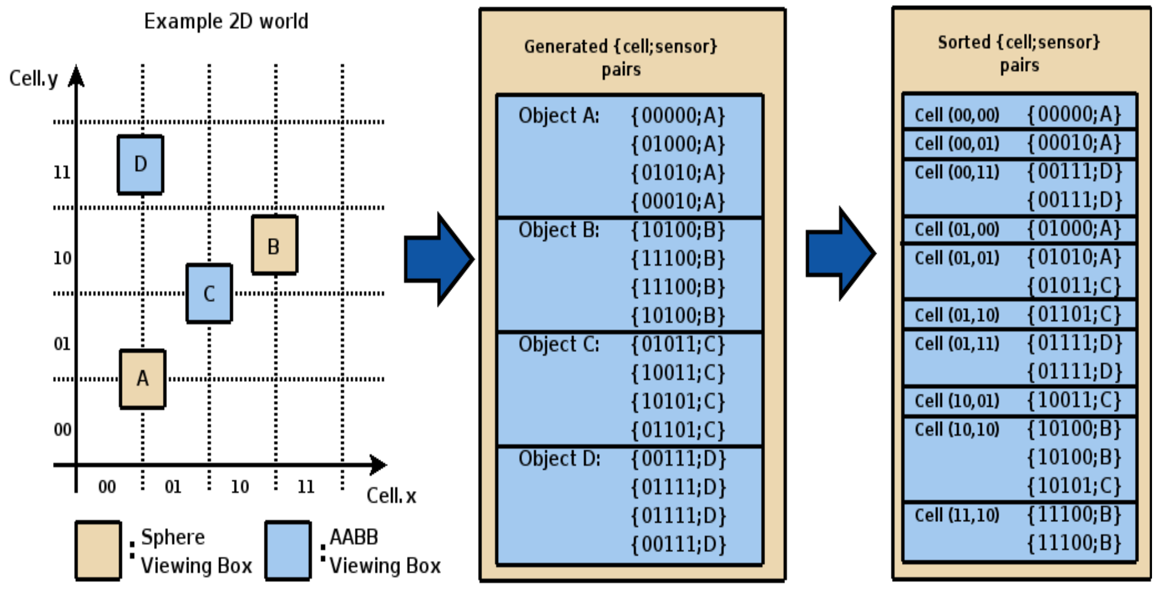

Example 2D world
Cell.y
11
10
01
00
00
01
10
11
Cell.x
D
B
C
A
Sphere Viewing Box
AABB Viewing Box
Generated {cell;sensor} pairs
Object A: {00000;A} {01000;A} {01010;A} {00010;A}
Object B: {10100;B} {11100;B} {11100;B} {10100;B}
Object C: {01011;C} {10011;C} {10101;C} {01101;C}
Object D: {00111;D} {01111;D} {01111;D} {00111;D}
Sorted {cell;sensor} pairs
Cell (00,00) {00000;A}
Cell (00,01) {00010;A}
Cell (00,11) {00111;D} {00111;D}
Cell (01,00) {01000;A}
Cell (01,01) {01010;A} {01011;C}
Cell (01,10) {01101;C}
Cell (01,11) {01111;D} {01111;D}
Cell (10,01) {10011;C}
Cell (10,10) {10100;B} {10100;B} {10101;C}
Cell (11,10) {11100;B} {11100;B}

Figure 4.12: Grouping Sensors together by their Cells

The sorting step follows a *stream duplicate removing* pass to remove the remaining duplicate entries which could not be removed prior since they were not adjascent.

Building Cell Array

The last step in this phase uses the *compare* algorithm to build *offset pointer* for the cells. This allows accessing each cell in the (cell; sensor) array directly. To access the different sensor types more efficiently, an extra offset is stored which indicates where in the cell the type of sensor changes. This data structure is shown in figure 4.13 and used as input for the *view frustum culling* phase.

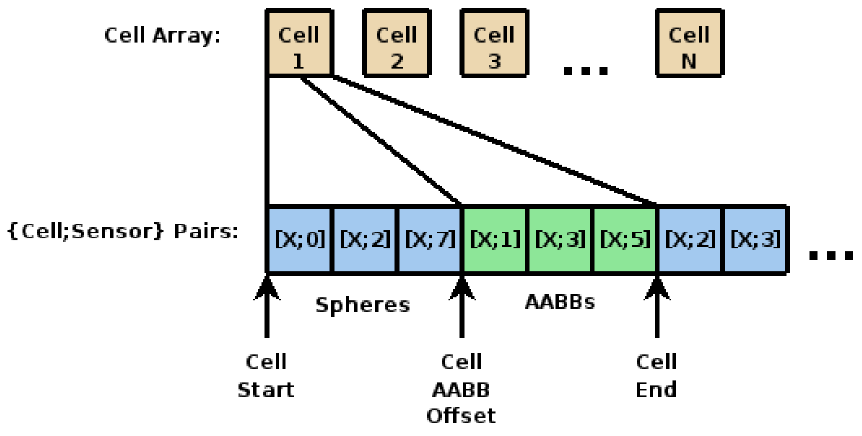


Figure 4.13: Spatial Sorting Results

4.3.2 View Frustum Culling

This phase uses the generated cell structure for view frustum culling. Since the sensors' view frustums are stored only as *view range*, *view width* and an *orientation*, it´s necessary to compute all view frustum planes before the culling tests. Then the culling tests are performed for all sensor pairs in each cell.

Precompute the Sensor´s View Frustum Planes

To optimize memory transfer between GPU and CPU all view frustums are stored as *view range* r, view width α and a rotation q. This eliminates the need to transfer all five viewing planes per sensor and spares the CPU the need to update all planes whenever a sensor changes his view frustum.

To perform the culling tests, without computing the view frustum for each sensor every time it is needed, all view frustums are precomputed. An default look coordinate system defined by look $\vec{L} = (0, 0, -1)$, up $\vec{U} = (0, 1, 0)$ and side $\vec{S} = (1, 0, 0)$ is used for this. First, the default look coordinate system is rotated by q to match the sensor's viewing direction. Then the back-facing culling plane is computed by rotating the look vector $\vec{L}$ by 180° around the up vector $\vec{U}$. The resulting vector is transformed by the view-range r yielding the normal vector $\vec{n}_{back}$ for the *back-plane*.

Thereafter the side-planes are computed. This is achieved by rotating $\vec{S}$ around $\vec{L}$ by the half viewing width $\frac{\alpha}{2}$. This yields the normal vector $\vec{n}_{left}$ for the left culling plane. The right plane is computed the same way but using the inverse side-vector $\vec{S}^{-1}$ and rotating by $\frac{-\alpha}{2}$. To compute top and bottom planes the same procedure as for left and right is used but with rotating the up vector $\vec{U}$ around the side vector $\vec{S}$.

Build possible visible Pairs

After computing the view frustum planes the algorithm starts to build arrays of potential visible pairs for each type combination. This allows implementing the culling test as a branch-free *kernel* with the *foreach* algorithm, since no type checks must be performed, to maximize performance.

These pairs are built in two passes. The first pass uses the equations from table 4.1 to compute the number of pairs per type combination in a cell. The second pass uses an *exclusive scan* for each type combination to compute where the pairs should be stored and the number of pairs per combination.

Table 4.1: Computing quantity of combinations in a cell

quantity of (sphere; sphere) in a cell	=	spheres	∗	(spheres − 1)
quantity of (sphere; AABB) in a cell	=	spheres	∗	AABBs
quantity of (AABB; AABB) in a cell	=	AABBs	∗	(AABBs − 1)
quantity of (AABB; sphere) in a cell	=	AABBs	∗	spheres

Now *foreach* is used to produce the pair structures and store these in an result array. This yields four arrays, as in figure 4.14, with potential visible pairs which are used for frustum culling.

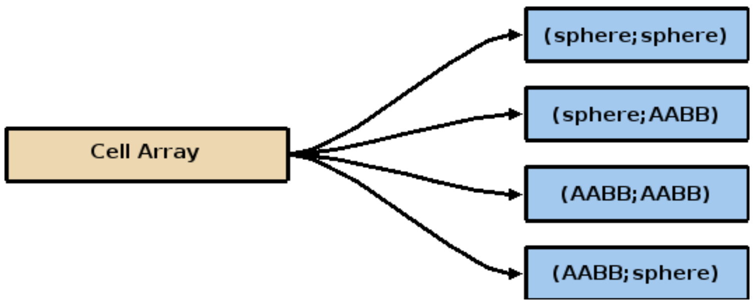


Figure 4.14: Building Type Combinations

Perform View Frustum Culling

The phase ends with the actual view frustum culling. For this, a thread is assigned to each possible visible pair to perform the culling test with *foreach*. This yields an result array of positive and negative culling tests. *Stream compacting* is used to remove all negative results yielding only visible pairs. Since this steps reads the sensor's view frustums repetitive and in random order the *texture memory* is used since it provides a on-chip cache with speeds up memory latency over global memory in this case.

4.3.3 Post-Process Culling Results

The culling results from the previous phase are still ordered by the *cell hash*. Since the culling information is needed per sensor, it's necessary to reorder the culling results so that these are grouped together per sensor. Thereafter, *offset pointer* are built to access the sensor results directly.

Reorder Culling Results

Before reordering the culling results, it´s necessary to merge all results for one sensor type as in figure 4.15.

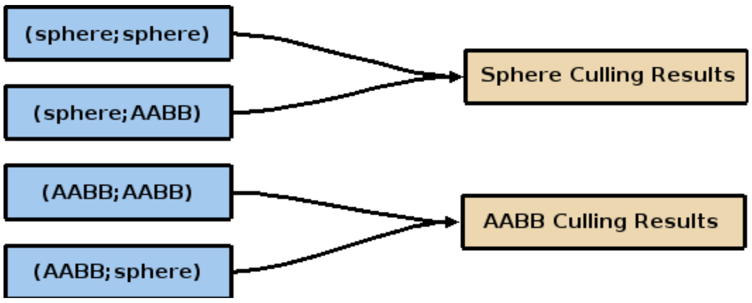


Figure 4.15: Coping Culling Results together

Then the *sort* algorithm is used to order these sensor by their *sensor id*, which groups all results for each sensor together.

Because of situations as in figure 4.16, it's possible that the culling results contain duplicates. In this example *Object A* and *Object B* are colliding in two different cells yielding different entries in the (cell; sensor) pair array which lead to duplicate results. Such duplicates are now removed by *stream duplicate removing*.

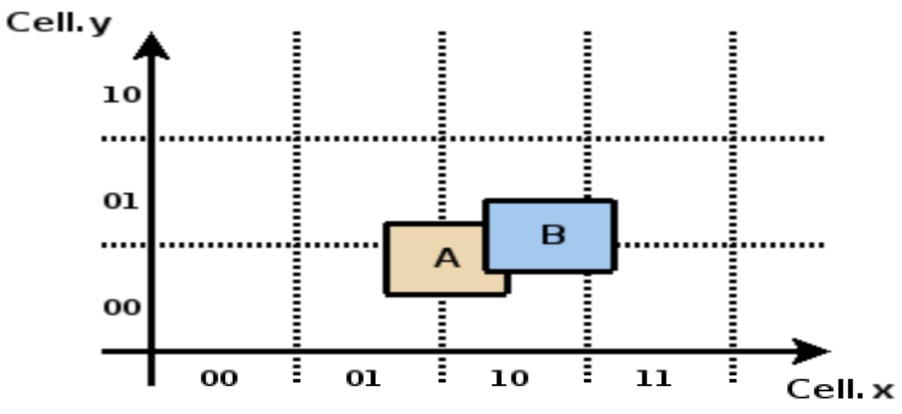


Figure 4.16: Duplicate Culling Results

Build Culling Results Offset Pointer

As a last step in the sensing algorithm, *offset pointers* are built for the sensor results as in figure 4.17.

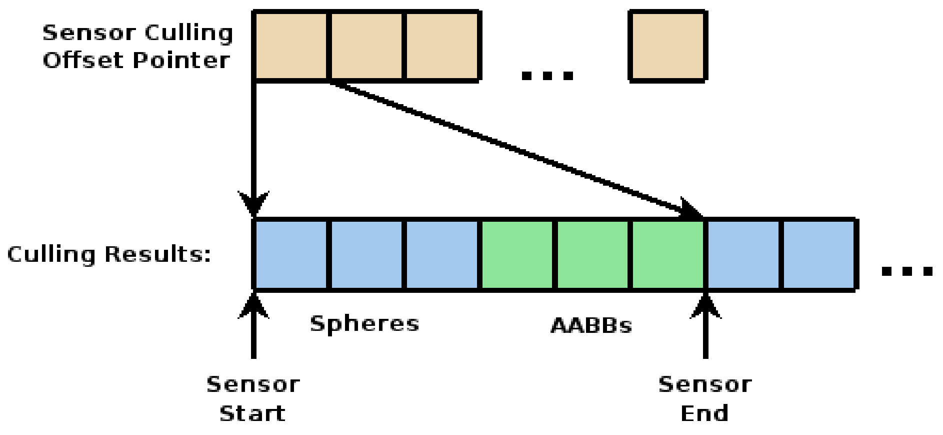


Figure 4.17: Algorithm Results

This structure allows the sensor service framework to query the viewing results for each sensor efficiently.

Chapter 5

Results

This chapter covers the tests which were run to benchmark the sensing algorithm. Each test was run 20-times and the average of the results were computed. All bounding volume and view frustum properties like view range were randomized. Furthermore, a cubic world was used so that the world size could be specified in edge length. This world was filled with random uniformly distributed sensors.

Section 5.1 shows how the algorithm scales with a different number of sensors. Furthermore, section 5.2 inspects how the number of used sensor types influence the algorithm´s performance. Then section 5.3 shows how the algorithm behaves with different numbers of threads. Moreover, section 5.4 compares the algorithm with a similar solution.

5.1 Scaling per Sensor Count

This section shows how the algorithm scales with the number of sensors. Since it´s a *spatial sorting* based algorithm, the size of the world can have heavenly influences on the runtime (a bigger world allows the sensors to be more distant so that the *spatial sorting phase* can exclude more combinations).

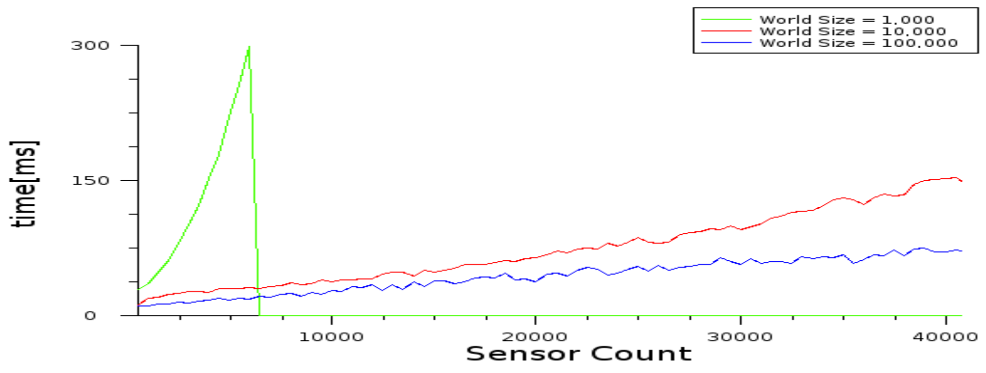


Figure 5.1: Scaling per Sensor Count

As illustrated in figure 5.1, the runtime of the algorithm improves with the world´s size for a constant number of sensors. The smallest world (size = 1,000, green graph) doesn't provide enough space for *spatial sorting* to work efficiently and so the runtime is near $O(n^2)$, also only 6,000 sensors were computable since storing all possible pairs required too much memory. In contrast, the both bigger worlds (size = 10,000 and size = 100,000) allow to exclude many pairs by *spatial sorting* and so the algorithm´s runtime changes to nearly $O(n)$. The next figures will inspect how each phase influences the runtime for a given world size.

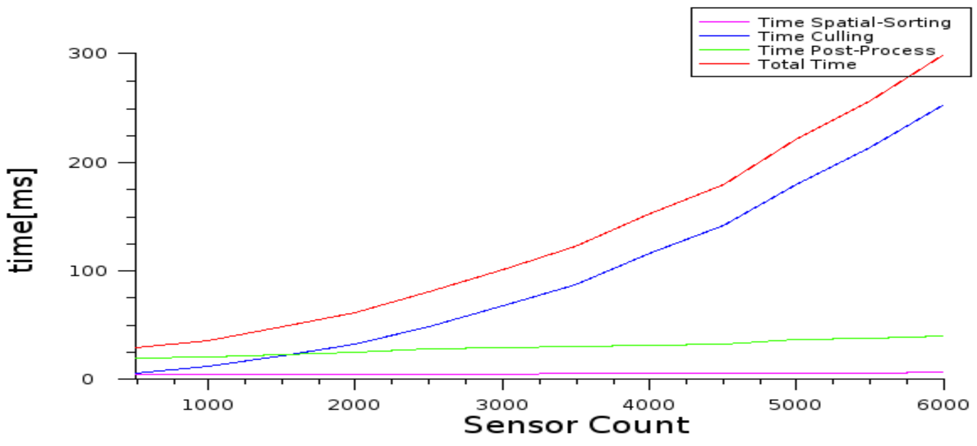


Figure 5.2: Scaling of Each Phase for World Size = 1,000

Figure 5.2 shows clearly that the *culling phase* makes up most the the computing time in a small world (size = 1,000) and produces the $O(n^2)$ work-load. Thus the *spatial sorting phase* could not exclude many sensor pairs. The *post process* phase also only makes a linear work-load since the results of the *culling phase* only grows linearly with the number of sensors.

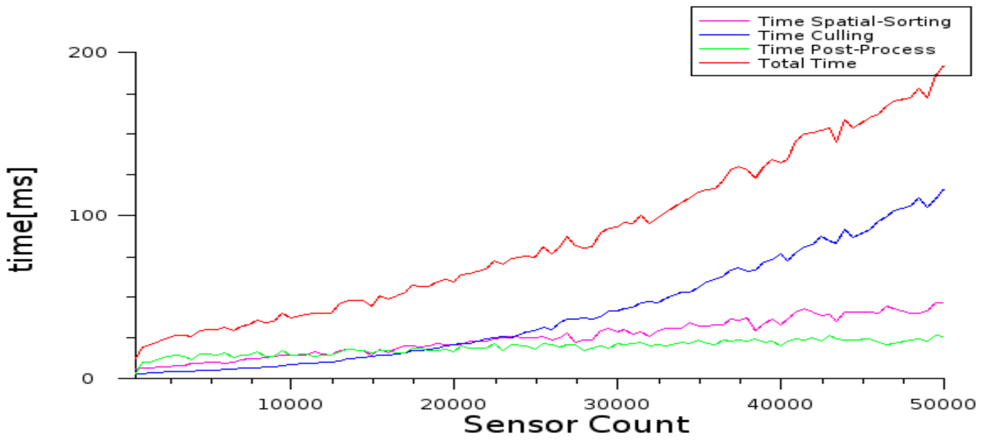


Figure 5.3: Scaling of each Phase for World Size = 10,000

When using a bigger world as in figure 5.3 the situation changes a bit. The *culling phase* still makes up for the major part of the computation time, but it scales more linearly. Also the *spatial sorting phase* takes more computational time. This leads to the observation that more sensor pairs then in a small world could be excluded. The *post process phase* still scales linearly, as in the small world, since the *culling* results scale mostly linearly with the number of input sensor in such an uniformly distributed world.

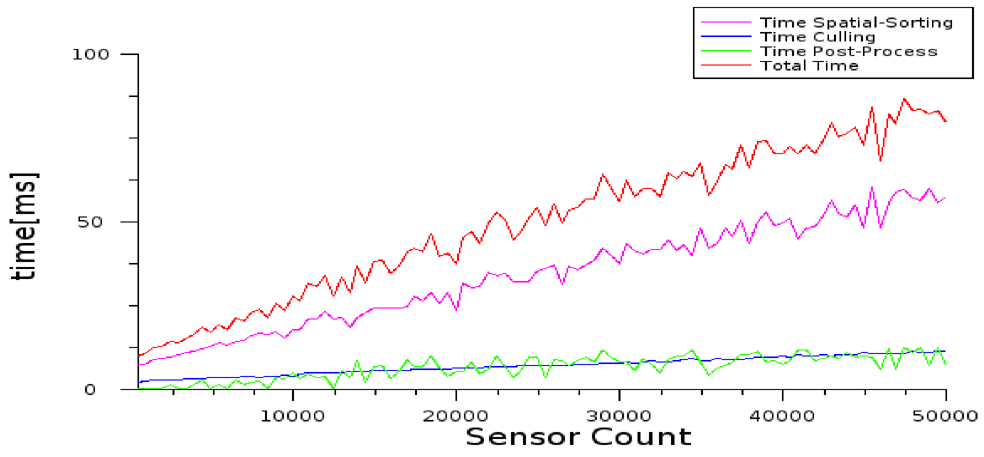


Figure 5.4: Scaling of each Phase for World Size = 100,000

Now an even bigger world is shown in figure 5.4. Here the situation reverses so that the *spatial sorting phase* produces the most work-load while the *culling phase* produces the fewest work-load and scales nearly linearly. This leads to the assumption that nearly all sensor pairs were excluded by *spatial sorting*, resulting in barely work for the following phases.

The observable jitter in the runtime of the various phases comes from the random distribution of sensors. This spatial distribution affects how many duplicate pairs can be reduced in *spatial sorting* and if/how many pairs are reported successful in the *culling phase*. Also, if no sphere or AABB tests are successful, no *post process* is needed for this type and so reducing the runtime of this phase dramatically.

As observed, the size of the world as well as the number of objects influence the runtime. The *cell sensor density* can be used to better understand this behavior. This density describes approximately (since each sensor can reside in more than one cell no correct assumption can be made) how many objects reside in each grid cell and is computed as shown in equation 5.1.

$$\text{cell object density} = \frac{\text{number of sensors}}{\text{number of cells}} \tag{5.1}$$

Figure 5.5 illustrates how the *cell sensor density* influences the algorithm. As observable, a higher *cell sensor density* leads to a higher work-load from the *culling phase*, while the *post-process* and *spatial sorting* phases aren't influenced by it. So it can be said that as denser the sensors lie the fewer pairs can be excluded leading to more work for the *culling phase*. Since the number of sensors is kept constant during the *cell sensor density* test, the runtime reaches a sort of "saturation" at which fewer grid cells aren't leading to more sensor pair combinations.

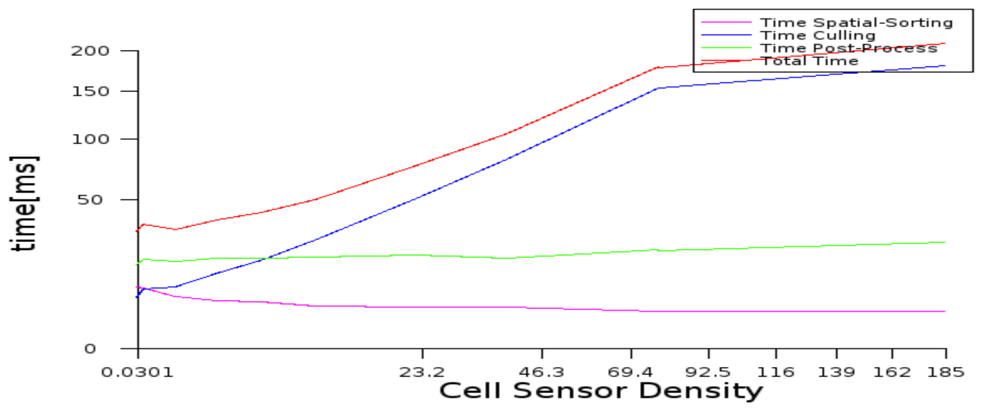


Figure 5.5: Scaling per *Cell Object Density*

5.2 Scaling for used Sensor Types

To inspect how usage of different sensor types influence the algorithm, some test were run with only AABB or sphere sensors present. A reference test was also run while using both types. Figure 5.6 shows the reference test.

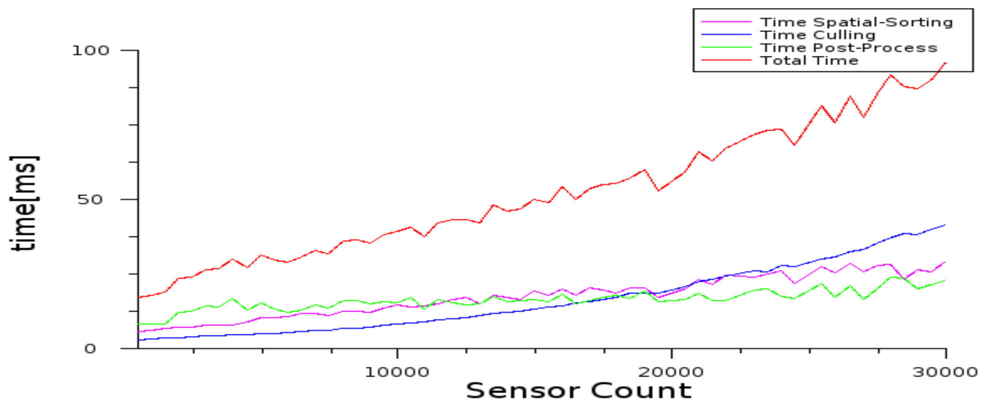


Figure 5.6: Scaling while using both Sensor Types

Figure 5.7 shows the same test with only sphere sensors (when using only AABBs the result is the same). As noticeable, the runtime of the *spatial sorting* and *culling phase* isn't influenced at all. The work-load from the *post process phase* is only reduced by a constant amount since each type is processed individually in this phase. So it can be said that the developed algorithm scales uninfluenced of the used sensor types.

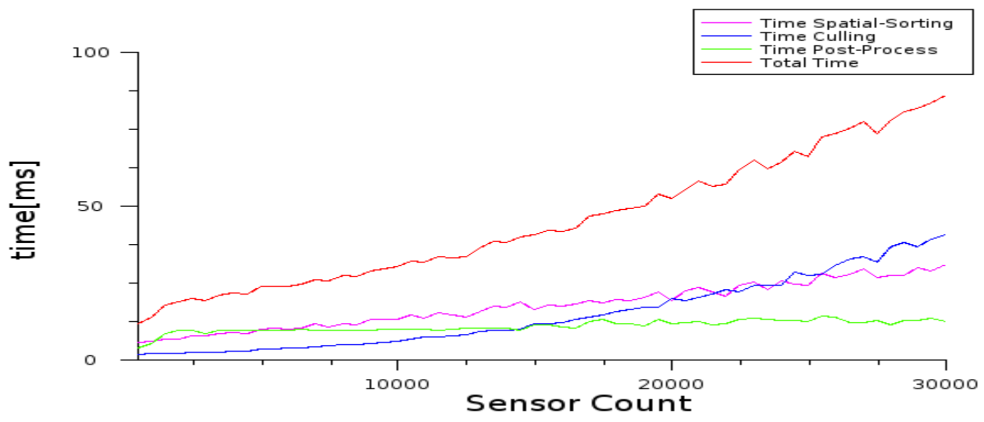


Figure 5.7: Scaling while only using Sphere Sensors

5.3 Scaling with the Number of used Threads

This section inspects how the number of used threads in each *kernel* influences the runtime performance. Since the algorithm makes heavy usage of *shared memory*, the number of usable threads is capped at 192 threads. Figure 5.8 shows that using under-populated *thread-warps* hurts the runtime performance. But as soon as a thread-warp is fully populated (16 thread in the current CUDA generation) the runtime isn't influenced anymore.

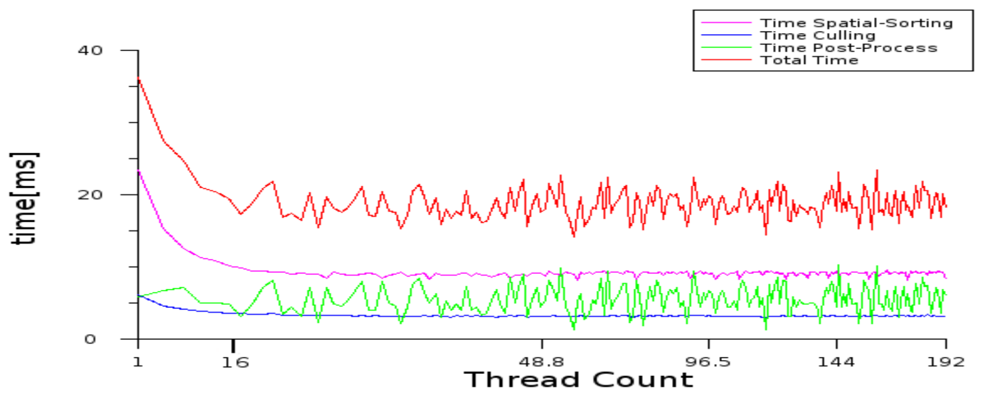


Figure 5.8: Scaling per Thread Count

5.4 Comparison with other similar Algorithms

Since the ideas from Scott Le Grand [Gra07] are a used as a base for the *spatial sorting phase*, it's natural to compare the results achieved with his.

The problem is, that he used only spheres as bounding volumes and didn't have a sophisticated post-process phase. Also he didn't provide the size of the world used for benchmarking. In his test he achieved 79 frames per second (around 12 milliseconds) for 30,720 objects.

The introduced algorithm needed 90 milliseconds for 30,000 sensors for a world of size of 10,000. But when looking only at the *spatial sorting phase*, the algorithm comes down to 25 milliseconds. When kept in mind that this implementation works with different sensor types and with a complicated overlapping test, the results are very satisfiable.

Chapter 6

Conclusion

The developed sensor service provides a prototype framework to encapsulate the sensing component from the sense-think-act process of the *agent model*. To compute these sensing informations GPGPU programming is used, hence the whole algorithm is executed on the GPU. This achieves, by utilizing *spatial sorting*, a efficient, nearly linear scaling and type independent algorithm. On the other hand, a high overhead for small data sets is observed, thus this GPU-based implementation is best applicable for bigger data sets.

Furthermore, the algorithm could be improved in various ways performance and function wise. Since the various sorting steps during the *spatial sorting* and *post-process* phases produce the highest work, when using a reasonable large world for the number of agents, a more sophisticated sorting algorithm would be a good starting point for further optimizations. For example an adaptive algorithm which chooses the best implementation based on the data set size. Another idea would be to try to optimize the GPU-based grid by using an update algorithm instead of recomputing the whole data structure each frame. Such an algorithm was researched by Jens Breitbart [Bre]. Also frame-coherence (using results from the last frame to reduce work in the current one) could be a thinkable optimization.

Then, to improve the functionality of the *sensor service*, various features could be implemented. These include support for more *bounding volumes* types as well as hierarchies of *bounding volumes*. In such a hierarchy an object would be represented as layers of bounding volumes. The most outher one wraps the object completely and inaccurate while the more inner ones wrap only parts of the object. This allows fast culling of objects as well as accurate culling at the cost of a more complex test and a higher memory requirement.

Acknowledgment

I would like to thank the following persons for their support in writing this thesis as well as their contribution to my university career.

Björn Knafla

for taking his time reviewing my thesis and suggestions how to improve it. Also for encouraging me to start a career in the game development industry as well as finding an internship there.

Michael Süß

for supervision my work in various projects related to parallel programming and so starting my interest for this fascinating field.

Anna Karl

for her love and her appreciation of the time I applied to work instead of her.

My family

for their support in my studies and allowing me to come as far as I did.

I would also like to thank **Josh Fahey** for proofreading this thesis.

List of Figures

1.1 An Agent´s Field of View . . . 1

2.1 2D View Frustum . . . 4
2.2 Sphere as Bounding Volume . . . 6
2.3 Sphere Plane Test . . . 6
2.4 AABB n,p-Vertices . . . 7
2.5 AABB as Bounding Volume . . . 7
2.6 AABB View Frustum Culling Special Case . . . 8
2.7 Using a Regular Grid for Spatial Sorting . . . 9
2.8 GPU and CPU GFLOPS/s . . . 10
2.9 Mapping of CUDA Threads to GPU Multiprocessors . . . 11
2.10 CUDA Memory Transfer Coalescing . . . 13

3.1 The Geometry Sensor Service . . . 15
3.2 Geometry Sensor Service Visualization . . . 16

4.1 Performance of using Aligned Structures . . . 18
4.2 *CUDA fast heap* Memory Handling . . . 19
4.3 Comparison CUDA Runtime Memory Handling vs *CUDA fast heap* . . 19
4.4 Foreach Algorithm . . . 20
4.5 Scan Algorithm . . . 20
4.6 Computing Offset Pointer . . . 21
4.7 Stream Compaction . . . 21
4.8 Runtime of Radix Sort . . . 22
4.9 The Sensing Algorithm . . . 23
4.10 Computing a *Cellhash* . . . 24
4.11 Computing a Viewing Box . . . 24
4.12 Grouping Sensors together by their Cells . . . 25
4.13 Spatial Sorting Results . . . 26
4.14 Building Type Combinations . . . 27
4.15 Coping Culling Results together . . . 28
4.16 Duplicate Culling Results . . . 28
4.17 Algorithm Results . . . 29

5.1 Scaling per Sensor Count . . . 30
5.2 Scaling of Each Phase for World Size = 1,000 . . . 31
5.3 Scaling of each Phase for World Size = 10,000 . . . 31

5.4 Scaling of each Phase for World Size = 100,000 32
5.5 Scaling per *Cell Object Density* 33
5.6 Scaling while using both Sensor Types 33
5.7 Scaling while only using Sphere Sensors 34
5.8 Scaling per Thread Count 34

Bibliography

[Bit] Nvidia sdk samples. Website. http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html#b%itonic.

[BK] C. Leopold B. Knafla. Parallelelization a real-time steering simulation for computer games with openmp. Website. http://www.plm.eecs.uni-kassel.de/plm/index.php?id=bknafla.

[Bre] Jens Breitbart. Case studies on gpu usage and data structure design. http://www.plm.eecs.uni-kassel.de/plm/fileadmin/pm/publications/chuerkamp/Ma%sterarbeiten/Jens_Breitbart_thesis.pdf.

[Buc] Wikipedia: Bucket-sort. Website. http://en.wikipedia.org/wiki/Bucket_sort.

[Bul] Bullet continuous physics engine. Website. http://www.bulletphysics.com/Bullet/wordpress.

[CUDa] Cuda programming guide. Website. http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Progr%amming_Guide_1.0.pdf.

[CUDb] Cudpp: Cuda data parallel primitives library. Website. http://www.gpgpu.org/developer/cudpp.

[Dal03a] Daniel Sanchez-Crespo Dalmau. *Core Techniques and Algorithms in Game Programming*, chapter Fundamental AI Technologies. New Riders, 2003.

[Dal03b] Daniel Sanchez-Crespo Dalmau. *Core Techniques and Algorithms in Game Programming*, chapter 3D Pipeline Overview. New Riders, 2003.

[Eri05a] Christers Ericson. *Real-Time Collision Detection*. Morgan Kaufmann, 2005.

[Eri05b] Christers Ericson. *Real-Time Collision Detection*, chapter Bounding Volumes. Morgan Kaufmann, 2005.

[Gra07] Scott Le Grand. *GPU Gems 3*, chapter Broad-Phase Collision Detection with CUDA. Addision-Wesly, 2007.

[MH07] John D. Owns Mark Harris, Shubhabrata Sengupta. *GPU Gems 3*, chapter Parallel Prefix Sum (Scan) with CUDA. Addision-Wesly, 2007.

[QT4] Qt4 library. Website. `http://trolltech.com/products/qt`.

[Rad] Wikipedia: Radix-sort. Website. `http://en.wikipedia.org/wiki/Radix_sort`.

[Shr07] Dave Shreiner. *OpenGL Programming Guide.* Addision-Wesly, 2007.

[Str97] Walter Strampp. *Höhere Mathematik mit Mathematica.* vieweg Lehrbuch, 1997.