

Better spacial hashing with linear memory usage and parallelism

Mykola Zhyhallo¹^a and Bożena Woźna-Szcześniak¹^b

¹*Department of Mathematics and Computer Science, Jan Długosz University in Częstochowa, Armii Krajowej 13/15,
42-200 Częstochowa, Poland
m.zhyhallo@ujd.edu.pl, b.wozna@ujd.edu.pl*

Keywords: Spatial hashing, collision detection, proximity queries

Abstract: Spatial hashing is an efficient approach for performing proximity queries on objects in collision detection, crowd simulations, and navigation in 3D space. It can also be used to enhance other proximity-related tasks, particularly in virtual realities. This paper describes a fast approach for creating a 1D hash table that handles proximity maps with fixed-size vectors and pivots. Because it allows for linear memory iteration and quick proximity detection, this method is suitable for reaching interactive frame rates with a high number of simulating objects. The technique we propose outperforms previous algorithms based on fixed-size vectors and pivots. Furthermore, our algorithm significantly reduces the memory usage of the pivots table, resulting in decreased dependency on the size of the scene. This improvement allows for more efficient memory utilization, irrespective of the scene's dimensions.

1 INTRODUCTION

Larger and more complicated simulations become possible as computers get more powerful. However, quicker algorithms are necessary to ensure appropriate reaction times in applications where real-time response is critical.

Rendering, collision detection, decision-making or AI processes are the most prominent areas where performance might suffer for a high number of objects. When rendering, objects that are not visible to the camera must be immediately excluded from consideration to maintain an acceptable frame rate. Collision detection often involves comparing each object in the scene to every other object in the scene, resulting in a $O(N^2)$ method that can significantly decrease application performance as the number of objects increases. Similarly, AI or autonomous agent decisions can be exponential, requiring each object to be aware of its distance from every other object.


Spatial hashing is a widely used method for speeding up proximity searches in large-entity simulations (Hastings and Mesit, 2005). This technique uses a hash function to map the positions of 2D or 3D objects to a 1D hash table. Additionally, the environment is organized into interconnected cells, forming a spatial grid. This organization minimizes the num-


ber of necessary comparisons between objects, leading to faster proximity queries. By utilizing the spatial grid, the method optimizes the search process and improves the overall efficiency of proximity-related operations.

Spatial hashing has been extensively applied in a variety of domains to address different challenges, such as real-time collision detection for simulations (Kniewel et al., 2023) or games involving a significant number of mobile objects (Hastings et al., 2004). It has also proven valuable in handling collisions among flexible or deformable models (Teschner et al., 2003; Mesit et al., 2004), and dense mesh animations (Kondo and Kanai, 2004).

Beyond the realm of graphics and simulations, spatial hashing methods have found applications in other contexts as well. They have been utilized in tasks such as nearest-neighbor detection in spatial databases (Zhang et al., 2004), and spatial hash-joins in relational databases (Lo and Ravishankar, 1996). These diverse applications highlight the versatility and effectiveness of spatial hashing algorithms in addressing spatial data processing requirements across different fields.

The construction of hash tables can be approached in various ways, taking into account the changing number of objects during a simulation. In (Pozzer et al., 2014) Cesar T. Pozzer et al. have introduced a spatial hashing algorithm, which employs fixed-size

^a <https://orcid.org/0009-0008-3131-7701>

^b <https://orcid.org/0000-0002-1486-6572>

vectors and pivots to address collisions within the hash table. This algorithm facilitates upfront computation of the memory requirements for storing the algorithm’s output. The usage of pivots allows for efficient updates and queries for the number of elements within each cell. Moreover, it simplifies the linear iteration through entities and cells in memory, enabling the effective execution of proximity queries. Furthermore, the approach presented in (Pozzer et al., 2014) demonstrates superior performance (3-10 times) and improved linear scalability compared to the hash table implementations proposed in (Buckland, 2005) and (Hastings and Mesit, 2005).

The algorithm described in (Pozzer et al., 2014) adopts a 3-step approach, which involves the creation of two tables: one for pivots and another for data. This implementation utilizes a space complexity of $O(T^2) + O(N)$, where T represents the size of the scene in cells and N denotes the number of objects. However, this approach results in significant memory consumption, particularly in large simulations where the number of objects is relatively smaller than the size of the scene.

In this paper, we present a novel method for the efficient creation of a 1D hash table, specifically designed to handle proximity maps using fixed-size vectors and pivots. Our approach allows for efficient linear memory iteration and rapid proximity detection, making it suitable for achieving interactive frame rates even with a high number of simulating objects.

The proposed solution addresses the issue of memory consumption by reducing its dependency on scene size. Moreover, it introduces a new storage structure that enables parallelized implementation with relaxed memory management, leveraging the processing resources offered by modern hardware. As a result, our solution exhibits significantly improved performance for large-scale simulations.

To validate the effectiveness of our approach, we will first implement and utilize a novel memory storage strategy for pivot tables. Next, we will introduce parallelization to the solution, capitalizing on the benefits of concurrent processing. Finally, we will compare our results against a basic implementation utilizing a hash map from the standard library and the algorithm described in the work of C. Pozzer et al ((Pozzer et al., 2014)).

Overall, our proposed solution offers a more efficient and scalable approach to spatial hashing for large-scale simulations, reducing memory consumption and enabling parallelized implementation.

2 PROPOSED SOLUTION

The method we propose extends the method presented in (Pozzer et al., 2014), which introduced a 3-step approach for constructing a hash table:

1. for each object, compute its cell ID, and locate this cell in the pivots table. Increment the object counter for this cell by 1;
2. traverse the pivots table from the beginning to the end. Determine the index where all objects within the cell specified by the pivot will be positioned in the data table, taking into account the number of objects in each pivot;
3. move the information of each object to the data table based on the stored index position in the corresponding pivot.

Our method changes the order of the statements inside each step, utilizes a modified data structure to store results, and introduces several new steps to synchronize concurrent work.

2.1 Parallelized algorithm

Our primary aim is to optimize the (Pozzer et al., 2014) construction of the spatial hash table by harnessing the computational power of multiple CPU cores. The (Pozzer et al., 2014) algorithm encompasses three distinct steps, ultimately yielding two essential tables: pivots and data. The pivots table indicates the precise number of objects within each associated cell. Additionally, it provides the necessary offset within the data table, facilitating efficient retrieval of pertinent information about these objects.

2.1.1 Step 1. Pivots calculation

As the initial step, a pivots table is allocated, comprising objects characterized by two atomic fields: count (representing the number of objects in the cell) and index (indicating the specific offset in the data table). In contrast to (Pozzer et al., 2014) algorithm, our method does not utilize columns named *Initial* and *Final*, thereby reducing the overall size of the pivots table by 33%. The dimension of this table is determined by $T * T$, where T corresponds to the size of the simulation world measured in cells. Prior to the commencement of the algorithm, it is crucial to ensure the proper initialization of objects within this table, ensuring their values are set to zero.

All the objects are divided into chunks, with the number of chunks determined by the available threads. Alternatively, a thread pool data structure can be employed to partition the objects table into chunks

of a pre-defined size. The start and end indexes of the chunk will be used in the algorithm to determine the bounds of the thread operation. Next, for each object (as demonstrated in Listing 1), the cell is calculated, and the counter for that cell in the pivots table is incremented by one. In order to reduce synchronization overhead, relaxed memory ordering can be used for atomic operations.

Listing 1: Pivots calculation algorithm

```

procedure CalculatePivots(Objects, Pivots, start, end)
  for i := start, i < end, i += 1:
    c := CalculateCellIndex(Objects[i])
    // Please, note that this operation is
    // an ATOMIC increment
    Pivots[c].count += 1
end procedure

```

The pivots table will be filled in a manner similar to the depiction shown in Figure 1. In contrast to (Pozzer et al., 2014) algorithm, our method employs atomic counters for pivots and runs in parallel for all objects.

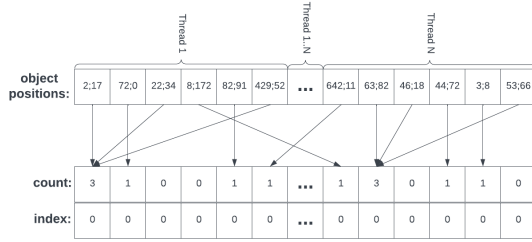


Figure 1: Building pivots table

2.1.2 Step 2. Calculating data offsets

In this step, we will calculate offsets for the objects within the data table. Leveraging the pivots table, we will set the index to the value $\text{offset} + \text{count}$ to each cell where the count exceeds 0. To prevent offset overlap, an atomic counter will be defined to indicate the subsequently available offset within the data table. The algorithm for filling the table with offsets is outlined in Listing 2, and the ultimate outcome can be observed in Figure 2.

To understand where indexes came from, we have to compare our solution to (Pozzer et al., 2014) algorithm. The crucial difference is that we are using an atomic counter that is shared between all the threads working simultaneously. That means that for every pivot, the index is calculated by atomic increment and fetch counter, thus the values in the indexes appear to be random. And in the end, the counter is equal to the number of elements in the pivots table.

Listing 2: Offsets calculation algorithm

```

procedure CalculateOffsets(Pivots, start, end, Counter)
  for p := start, p < end, p += 1:
    count := Pivots[p].count
    if count != 0:
      offset := Counter.FetchAdd(count)
      Pivots[p].index = offset + count
end procedure

```

As stated before, in contrast to the (Pozzer et al., 2014) algorithm, our method employs a global counter with a `FetchAdd` operation, which is an atomic operation that retrieves the current value and simultaneously adds another value to it. This ensures that other threads searching for the index do not overlap. Notably, our approach does not utilize the *Initial* and *Final* fields in the pivots table, as we solely rely on the *Index* field.

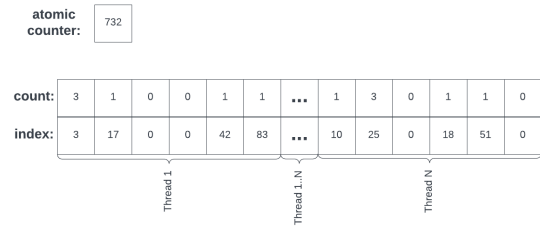


Figure 2: Precalculating data offsets

2.1.3 Step 3. Copy objects into hash table

Utilizing the offsets contained within the pivots table, we can proceed to copy the object information to the designated data table. This data migration process is executed concurrently for each individual object, as shown in Listing 3. First of all, for each object, we are calculating its cell. Then we get the next available position in the data table by leveraging `FetchAdd` operation on the pivots index value. By utilizing this operation, we can guarantee the selection of a unique index in the data table for each object, even when multiple threads are involved. The final step involves transferring the object information to the data table.

Listing 3: Data migration algorithm

```

procedure MigrateData(Objects, Pivots, Data, start, end)
  for i := start, i < end, i += 1:
    c := CalculateCellIndex(Objects[i])
    // FetchAdd returns the value stored before
    // operation,
    // "-1" is required to actually subtract it
    index := Pivots[c].index.FetchAdd(-1) - 1
    Data[index] = Objects[i]
end procedure

```

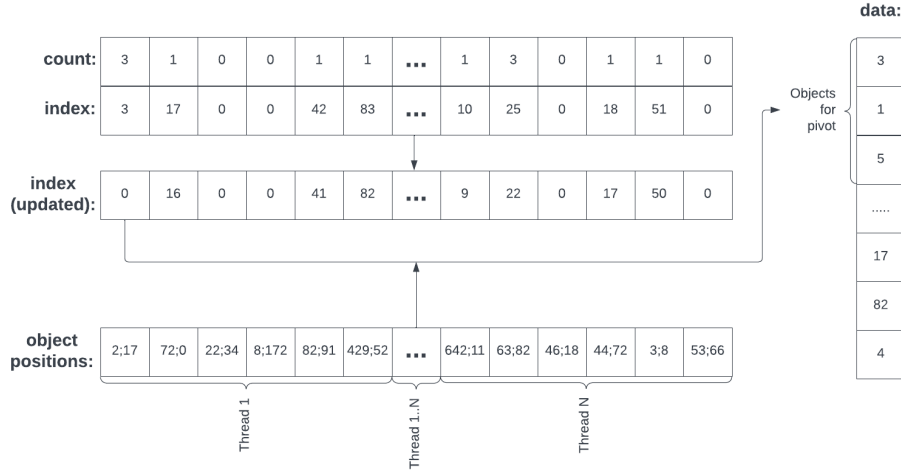


Figure 3: Building the final pivots and data tables

As a result, we will have a properly constructed pivots table with offsets pointing to the filled regions in the data table, as depicted in Figure 3.

2.2 Parallelization with better pivots storage. Chunked algorithm

To effectively handle collisions within the hash table, the approach described in (Pozzer et al., 2014) utilizes a methodology centered around fixed-sized vectors and pivots. This technique enables the precomputation of the memory needed to store the algorithm’s output, thereby facilitating efficient resource allocation. Regarding the storage of pivots, the (Pozzer et al., 2014) method always uses a table of dimensions $T * T$, where T represents the size of the scene measured in cells. In cases where the scene is relatively small, the corresponding table size is also small, ensuring that all entries within the table are used. However, in situations where the scene is considerably larger and consists of only a few objects, a significant portion of the table becomes redundant or unused.

This observation prompts a fundamental question: How can we avoid the allocation of memory for the nonessential portions of the scene? The most straightforward solution relies on the implementation of memory paging. By partitioning the scene into uniformly sized chunks, we can allocate memory solely for the chunks that encompass objects of interest.

It is essential to select an appropriate strategy for splitting space into chunks:

- when performing computations that do not necessitate the utilization of floating-point arithmetic, particularly in scenarios involving stable physics

simulations, it is wise to partition the space into chunks with a size that aligns with a power of two. In this case, the calculation of the chunk index for an object can be achieved efficiently without utilizing division, but rather by leveraging binary shift operations;

- it is recommended that the size of the table storing chunk pivots be padded and aligned to accommodate the CPU cache line size. This alignment optimization helps facilitate improved cache utilization and subsequently enhances overall performance;
- the precise determination of the chunk size ought to be deduced through a comprehensive assessment of performance metrics and the specific requirements of the application at hand.

The aforementioned optimizations are crucial as they introduce an additional step in the application’s workflow, which involves calculating the cell chunk index to select a chunk from the chunks table, and then using the pivots table from the specific chunk. Since this operation is executed frequently, optimizing the underlying data structure becomes essential to ensure efficient performance.

We shall proceed with a step-by-step reconstruction of the algorithm proposed in (Pozzer et al., 2014), meticulously revisiting each step while incorporating necessary adaptations and refinements. Also, we will leverage our work on parallelization and show it working with this new data structure.

2.2.1 Step 0. Clear

This particular step was omitted from the work conducted by C. Pozzer et al. ((Pozzer et al., 2014)) due to its minimal time requirement when dealing with small-scale scenes. However, for larger scenes, despite the operation's swift execution on contemporary computers that leverage the full potential of SIMD (Single Instruction, Multiple Data), it can still consume multiple milliseconds. Such time intervals prove to be excessive for certain real-time applications, such as video game physics simulations.

In this step, the pivots table necessitates zeroing. This step is performed on every iteration of the algorithm. The current approach significantly enhances its efficiency by exclusively zeroing the allocated space. Furthermore, when the chunk count is substantial, the operation can be distributed among multiple threads. In such cases, each thread is assigned a distinct set of chunks to zero, thereby enabling parallel processing and optimization of the zeroing process.

2.2.2 Step 1. Pivots calculation

As we stated earlier, we assume that the objects in the pivots table will have only two fields: an atomic counter for objects and an atomic counter for indices. This modification will result in a reduction of the pivot table's size by 33% compared to the original solution.

Our algorithm presented in Listing 4 utilizes atomic incrementation (i.e., the `AtomicInc(1)` method) and compare-and-swap (i.e., the `CAS(whereToSwap, oldValue, newValue)` method) operations to allocate memory for chunks, increment the counter in the pivot, and increment total objects counter for each chunk.

Listing 4: Pivots calculation algorithm

```
procedure CalculatePivots(Objects, start, end, Chunks)
  for i := start, i < end, i += 1:
    c := CalculateChunkIndex(Objects[i])

    if Chunks[c] == NULL:
      newChunk := AllocateChunk()
      if not CAS(Chunks[c], newChunk, NULL):
        DeallocateChunk(newChunk)

    p := CalculatePivotIndex(Objects[i])
    Chunks[c].pivots[p].count.AtomicInc(1)
    Chunks[c].objectsCount.AtomicInc(1)
  end procedure
```

The object list is partitioned into slices, with each slice allocated to an individual thread. This enables parallel processing, with each thread independently operating on its assigned slice of objects.

The utilization of a memory pool data structure for chunk allocation and deallocation operations is essential at this stage. The absence of a memory pool would hinder the algorithm's performance compared to alternative solutions, as the repeated allocation of all the chunks in each iteration would be excessively time-consuming. That's why under the methods `AllocateChunk()` and `DeallocateChunk(newChunk)` we use the memory pool and not just a simple allocator.

2.2.3 Step 2. Synchronization of the chunks starting indexes

The original approach employs a single large vector with a static size to store all object data. In our solution, all the data can still be stored in a single vector, but an additional step is required to synchronize the pivots tables across all chunks. This involves iterating through all the chunks and assigning a starting index based on the total objects count within each chunk, as outlined in Listing [5].

Listing 5: Indexes calculation algorithm

```
procedure SyncPivotTables(Chunks)
  accum := 0
  for i := 0, i < SizeOf(Chunks), i += 1:
    Chunks[i].startIndex = accum
    accum += Chunks[i].objectsCount.Get()
  end procedure
```

This step can be parallelized; however, since it involves iterating through a small list of initialized chunks, the operation is expected to be very fast;

2.2.4 Step 3. Calculation of indexes inside chunks

In this step, we will calculate offsets for the chunk objects within the data table. Leveraging the chunk pivots table, we will initially set the index to the chunk `startIndex` value for each chunk. To prevent offset overlap, a counter will indicate the subsequently available offset within the data table. The algorithm for filling the table with offsets is outlined in Listing 6

Listing 6: Indexes calculation algorithm

```
procedure CalculateIndexes(Chunk)
  if Chunk == NULL:
    return
  counter := Chunk.startIndex
  for p := 0, p < SizeOf(Chunk.pivots), p += 1:
    counter += Chunk.pivots[p].count.Get()
    Chunk.pivots[p].index.Set(counter)
  end procedure
```

As in the previous step, the chunks can be distributed among multiple threads. It is noteworthy that

we only operate on the allocated chunks, skipping the unallocated ones. This significantly reduces the overall execution time.

2.2.5 Step 4. Assign objects in data tables

Once the data and pivots tables are prepared, the final step involves adding each object to the data tables using the corresponding indexes from the pivots tables, as demonstrated in Listing 7.

Listing 7: Indexes calculation algorithm

```

procedure AssignObjects(Objects, Chunks)
  for i := 0, i < SizeOf(Objects), i += 1:
    c := CalculateChunkIndex(Objects[i])
    p := CalculatePivotIndex(Objects[i])
    // Calculating index in the data table.
    // FetchAdd returns the value stored before
    // operation,
    // "-1" is required to actually subtract it
    d := Chunks[c].pivots[p].index.FetchAdd(-1) - 1
    Chunks[c].data[d] = Objects[i]
end procedure

```

The object list is partitioned into chunks, and each chunk is allocated to a separate thread. This enables parallel processing, allowing each thread to operate independently on its assigned chunk of objects.

3 EXPERIMENTS AND RESULTS

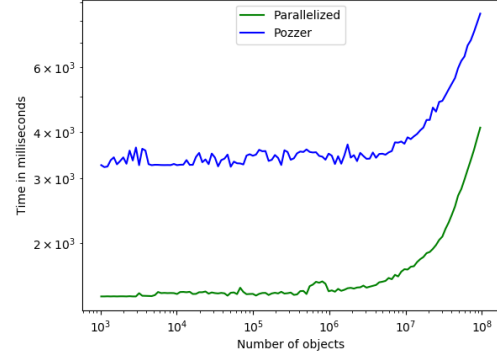
To assess the performance of the proposed algorithm, a series of experiments were conducted. These experiments involved varying the number of objects and the size of the simulation scene. The simulation scene size has ranged from 2048x2048 up to 14336x14336 cells. All experiments were performed on a notebook PC with an AMD Ryzen 7 5000 CPU, 32 GB of RAM, running on 16 threads. The following two testing scenarios were considered:

- Real world - where the distribution of objects in the simulation space is non-uniform, resulting in regions with a significantly higher concentration of objects and regions with no objects at all.
- Worst case - objects are uniformly distributed across the simulation space.

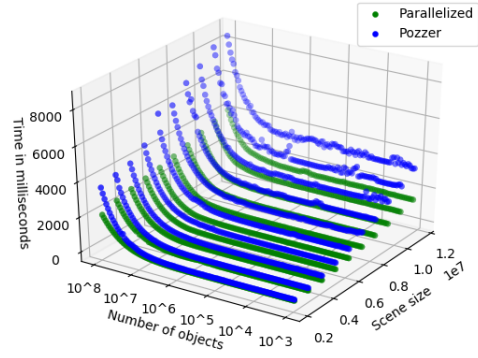
3.1 Parallelism

We present a comparison between our newly developed parallelized method and the method proposed in (Pozzer et al., 2014), focusing on scene size, object count, and simulation runtime. Our parallelized method, as illustrated in Figure 4a, demonstrates a

significant reduction in simulation time, ranging from 2-4 times faster. In this simulation, objects are uniformly distributed across the simulation space. In order to create Figure 4a, the algorithm was run for different scene sizes and object counts, resulting in 1200 simulations.



(a) Fixed scene size



(b) Different scene sizes

Figure 4: Results of the parallelized algorithm

The algorithm implemented in our method exhibits random memory access, causing a bottleneck in RAM input/output (I/O) operations rather than CPU power. Consequently, increasing the number of CPU cores does not guarantee a decrease in simulation time, even with our algorithm's lock-free design and utilization of relaxed memory access.

To reinforce this observation, we refer to Figure 4b, which shows the simulation time increase as the simulation space expands, resulting in higher RAM allocation. Simulation time rises due to the CPU's inability to cache memory regions for extended periods, given the constant loading of new regions. Thus, the necessity for an improved data structure is evident, as demonstrated in this study.

In cases where the simulation space is extremely small, parallelization may not yield significant benefits. Additionally, the implementation of our algorithm may introduce unnecessary complexity, potentially leading to increased simulation time and CPU usage.

3.2 Better data structure

Let's analyze the worst-case scenario for the chunked algorithm, where objects are uniformly distributed in the simulation space, and storage allocation optimization is not feasible. Consequently, all regions must be allocated without the possibility of achieving efficiency gains. This is evident in Figure 5a, where regions exhibit a gradual increase in memory usage.

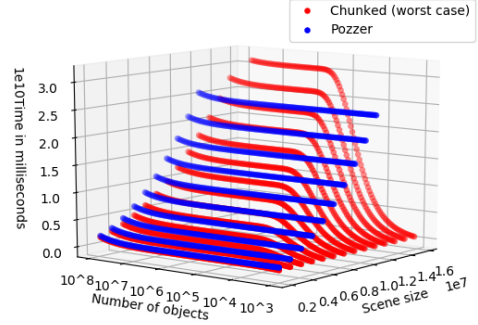
In real-world simulations and tasks, where simulation spaces typically contain concentrated groups of objects, memory usage tends to vary significantly. In this case, as an example of real-world usage, we selected a physics engine for a 2D game. Figure 5b demonstrates that the memory usage is no longer dependent on the simulation space but rather linearly depends on the number of objects. This indicates that our chunked algorithm achieves efficient memory allocation by effectively managing the concentrated object groups within the simulation space.

A comparative analysis of simulation time (Figure 6a) and memory usage (Figure 6b) is provided for various scene sizes and algorithms, focusing on a specific scenario with a small, predetermined number of objects (10000 objects). On the charts, we compared the algorithm from (Pozzer et al., 2014) work (named *Pozzer*), with the algorithm from section 2.2 working in worst case scenario (*Chunked (worst case)*) and real-world scenario (*Chunked (real case)*). We can see, that memory and CPU usage for the proposed *Chunked* algorithm, in contrast to *Pozzer*, don't directly depend on the simulation size in real-world scenarios.

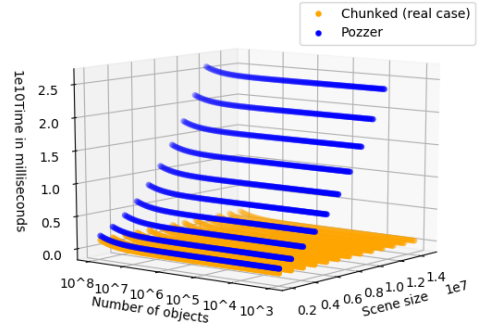
4 Results summary

In this study, we have introduced a novel spatial hash algorithm that employs fixed-size vectors and pivots to handle collisions within the hash table. The proposed approach offers several advantages, including reduced memory consumption, support for parallelization, and enhanced performance in large-scale simulations. Our algorithm is designed to leverage modern hardware resources, resulting in improved efficiency and scalability for spatial hashing tasks.

To validate the effectiveness of our approach, we



(a) The worst case scenario



(b) A real-world scenario

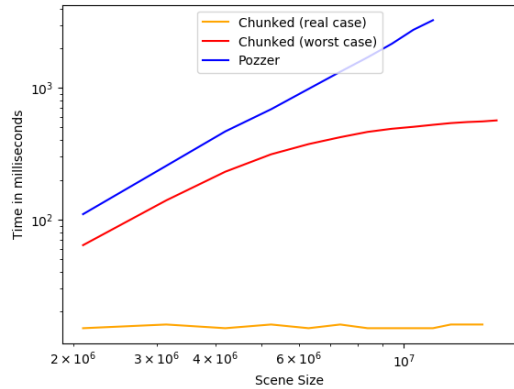
Figure 5: Memory usage of the chunking algorithm

implemented a new memory storage strategy for pivot tables, introduced parallelization techniques, and conducted a comparative analysis against existing implementations. Our findings highlight the need for a more efficient data structure, as we identified RAM I/O operations rather than CPU power as the primary bottleneck of the algorithm.

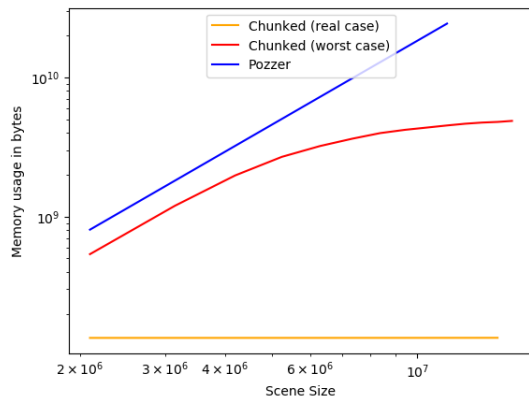
Additionally, our study presents observations related to simulation time increases with the expansion of the simulation space and memory allocation. We discuss the worst-case scenario for the chunked algorithm and emphasize that memory usage varies significantly in real-world simulations, particularly when dealing with concentrated groups of objects.

Overall, this research addresses the limitations of current spatial hashing techniques and proposes an efficient solution for large-scale simulations by mitigating memory consumption, facilitating parallelization, and optimizing proximity queries.

Nevertheless, there are potential ways for further improvements in our algorithm. For instance, explor-



(a) Simulation time for 10000 objects



(b) Memory usage for 10000 objects

Figure 6: Resource usage comparison for fixed objects count

ing the possibility of leveraging GPU computations, because the algorithm is designed in a way to support GPU integration. Additionally, making slight modifications to enable the utilization of SIMD operations could also enhance performance. These improvements could be explored in future research endeavors.

REFERENCES

- Buckland (2005). *Programming Game AI by Example*. Wordware game developer’s library.
- Hastings, E. and Mesit, J. (2005). Optimization of large-scale, real-time simulations by spatial hashing. In *Proceedings of Summer Computer Simulation Conference*, volume 37(4) of SCSC’05, pages 9–17. Society for Modeling & Simulation International (SCS).
- Hastings, E., Mesit, J., and Guha, R. K. (2004). T-collide: A temporal, real-time collision detection technique for bounded objects. In *Proceedings of the 5th Game-On International Conference on Computer Games: Artificial Intelligence, Design and Education (CGAIDE)*, pages 41–48. The University of Wolverhampton, School of Computing and Information Technology Printed in Wolverhampton, UK.
- Kniewel, C., Pejic, A., Krüger, L., Ziegler, C., and Adamy, J. (2023). Boids flocking algorithm for situation assessment of driver assistance systems. *IEEE Open Journal of Intelligent Transportation Systems*, 4:71–82.
- Kondo, R. and Kanai, T. (2004). Interactive physically-based animation system for dense meshes. In *Proceedings of Eurographics 2004*, pages 93–96. The Eurographics Association.
- Lo, M.-L. and Ravishankar, C. V. (1996). Spatial hash-joins. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, volume 25 of SIGMOD’96, page 247–258. ACM.
- Mesit, J., Guha, R. K., and Hastings, E. (2004). Optimized collision detection for flexible objects in a large environment. In *Proceedings of the 5th Game-On International Conference on Computer Games: Artificial Intelligence, Design and Education (CGAIDE)*, pages 49–54. The University of Wolverhampton, School of Computing and Information Technology Printed in Wolverhampton, UK.
- Pozzer, C., De, C., Pahins, C., Heldal, I., Mellin, J., and Gustavsson, P. (2014). A hash table construction algorithm for spatial hashing based on linear memory. In *Proceedings of the 11th Conference on Advances in Computer Entertainment Technology (ACE’14)*, ACE’14. Association for Computing Machinery.
- Teschner, M., Heidelberger, B., Müller, M., Pomerantes, D., and Gross, M. H. (2003). Optimized spatial hashing for collision detection of deformable objects. In *The 8th International Fall Workshop on Vision, Modeling, and Visualization*, VMV, pages 47–54. Aka GmbH.
- Zhang, J., Mamoulis, N., Papadias, D., and Tao, Y. (2004). All-nearest-neighbors queries in spatial databases. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management (SSDBM’04)*. IEEE Computer Society.