Spatial Partitioning Strategies

Linus Kåreblom

Technical program Academic year 2024/2025
School: Österänggymnasiet
Class: TE3A
Supervisor: Tony Tieu

February 28, 2025

Abstract

Spatial partitioning is a widely used optimization strategy to efficiently optimize objects in space. It can be found in many applications, from simulating galaxies to video games. Checking collision between objects is an expensive operation that quickly becomes a bottleneck as the number of objects grow. This paper compares three different strategies, alongside a naive solution. The aim is to evaluate the use cases for the different strategies. Each strategy was implemented to fit the same interface to make it as fair of a comparison as possible. The results show how quadtrees out performs every other method when the objects are evenly and uneven distributed as the number of objects grow.

Contents

1 Introduction **4**

2 Purpose & Scope **4**

3 Theory **5**

3.1 Spatial Partitioning 5

3.2 Quadtree 5

3.3 The Stack and Heap 6

3.4 Big O notation 6

3.5 Static Grid 6

3.6 Hash Map 6

3.7 Spatial Hashing Grid 7

3.8 Vector 7

4 Material and Method **7**

4.1 Linux 7

4.2 Neovim 7

4.3 Git 8

4.4 CMake 8

4.5 C 8

4.6 SDL2 8

4.7 Python 8

5 Results & Analysis **9**

5.1 Average Runtime 9

5.1.1 Evenly Distributed Objects 9

5.1.2 Unevenly Distributed Objects 10

5.1.3 Analysis 10

5.2 Insertion 11
5.2.1 Evenly Distributed Objects 11
5.2.2 Unevenly Distributed Objects 11
5.2.3 Analysis 12
5.3 Querying 12
5.3.1 Evenly Distributed Objects 12
5.3.2 Unevenly Distributed Objects 12
5.3.3 Analysis 13
5.4 Collision 13
5.4.1 Evenly Distributed Objects 13
5.4.2 Unevenly Distributed Objects 14
5.5 Analysis 14

6 Discussion **15**

7 Sources **16**

8 Appendix **16**

1 Introduction

When I was developing a small physics engine, I encountered performance issues during the collision detection phase. In this phase the program figures out which objects are close enough to be touching and overlapping. This would not be an issue in a small simulation with few objects, but I was striving for something grand running in real time which made every millisecond count. Due to these constraints I had to find a way to speed things up.

On my optimization journey I encountered the concept of spatial partitioning - an elegant solution to my problem. Diving deeper I discovered the three techniques I will be comparing in this paper, along side the naive solution. The basic theory of the technique is simply to group objects close to each other to avoid checking two objects on the opposite side of the world. This drastically decreases the number of collision checks performed each frame.

As a consequence of this constraint I had to find a way to optimize this phase. That is what sent me down the rabbit hole of spatial partitioning as an optimization to do query objects close to each other in space. During my research I found out about the three strategies I am comparing in this paper. I am doing this paper within the programming course.

2 Purpose & Scope

The purpose of this paper is to find the most optimal strategy for dividing up space to improve the speed of spatial querying of nearby objects.

This study focuses mainly on CPU performance and runtime in exchange for less memory efficient implementation. Therefore time based evaluations of insertion, query time and general collision testing time will be performed.

The strategies are going to be evaluated on time based metrics - insertion, query and time spend doing collision testing. These evaluations are going to take place with two different distribution strategies of the objects though the same seed will be used for all tests so the pseudo-random numbers stay the same.

These strategies could be extended into the third dimension but this study focuses solely focused on two dimensional space.

The questions I'm going to answer are:

- At what point is it worth implementing a more advanced data structure?

- In what use case are the different algorithms most useful?

3 Theory

3.1 Spatial Partitioning

Spatial partitioning is an optimization technique used to make programs run faster. It is often found within physics applications. The theory behind this technique is to reduce the number of times one object is checked against another. Suppose we have one hundred objects in our world. The naive way to check each one of these against the others results in a ten thousand checks. Since checks are expensive to make, we want to perform as few of them as possible.

3.2 Quadtree

A quadtree is a recursive data structure, meaning it contains pointers to each objects of the same type as itself. It is in the family of tree data structures. The data structure is comprised of nodes. Each node has four pointers to other nodes, could also be referred to as branches or leaves. This strategy works by inserting points into a leaf until the leaf reaches its capacity, then it splits into four equally large sub leaves. This subdivision pattern is shown in figure 1.

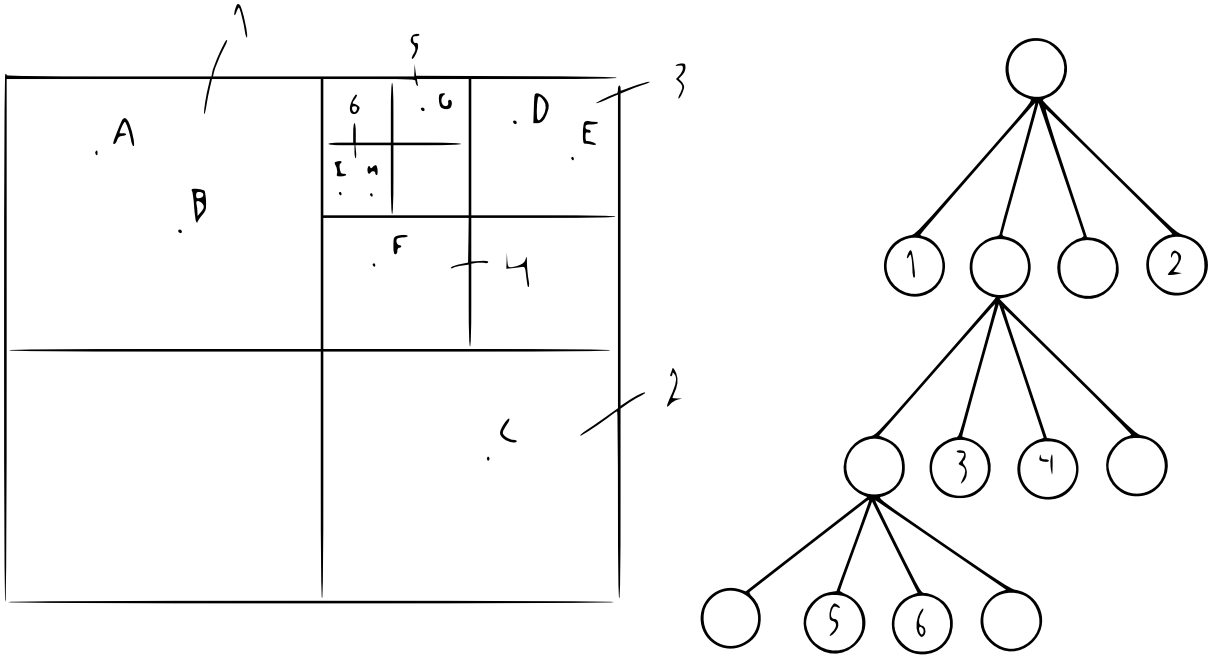


Figure 1: Visual representation of a quadtree.

3.3 The Stack and Heap

A program uses two different types of memory, the stack and the heap. When the program starts a fixed amount of bytes are allocated for the program to use, this is what is called the stack. When the program creates a variable it is allocated on the stack, which is done by incrementing the stack pointer by the size of the variable data type. Since a stack allocation is only a single instruction it is incredibly fast. The heap is much larger than the stack but slower because you have to request space within it from the operating system. Allocating is considered to be a slow operation since there is a lot of overhead introduced.

3.4 Big O notation

Big O notation is a notation system to describe the time complexity of an algorithm. If an algorithm has the time complexity of $O(n)$ then that means its time complexity grows linearly with the amount of inputs n. This could be deceiving due to a hidden constant within the expression. If $O(n)$ would be written as a function to describe the actual execution time of the algorithm then it would look like this $T(n) = C \cdot n$. A $O(n^2)$ algorithm would be considered worse than an $O(n)$ one. This assumption is, however, not always correct. If n is small enough then the $O(n^2)$ algorithm could run faster due to the constant being smaller than its $O(n)$ equivalent.

3.5 Static Grid

The static grid is built using a simple array. Indexing into the array is an $O(1)$ operation. The caveat with this approach is you are forced to know how big the area you want the to partition to be before compiling the program. Both insertion and querying only requires a simple index into the grid array which makes this approach super fast.

3.6 Hash Map

A hash map is a data structure which key value pairs allowing for aliasing of values. For example, a hash map using strings as keys and integers as values could map the key "life" to the value 42, meaning a query into the hash map for "life" would return 42. The way this works is by using a cryptographic hashing function to generate a unique hash for the object. This hash is a simple number which is usually very large. To make the hash useful you run modulo on it to make sure it's between zero and the number of buckets inside the hash map. The same method is used for retrieval, first running the key through the same hashing function, then wrapping it using modulo and returning what lies in the bucket.

The only caveat with this is there could be collisions within the hash map. Two hashes could wrap into the same index. There are a multitude of ways to handle collisions like open addressing and separate chaining.

3.7 Spatial Hashing Grid

The spatial hashing grid is built using the same technique as a hash map. The method I used to write my spatial hashing grid was to allocate an arbitrarily sized array. When inserting into the structure I first hash the coordinate of the object, then run a modulo operation on it giving me a number within the array bounds, and then it is inserted into the cell. No collision resolution is taking place inside the structure meaning two cells could have the same bucket index which means queries for the one of the cells returns the objects in the other as well.

3.8 Vector

A vector refers to a dynamic array, not the mathematical concept of a vector. To allocate an array on the stack you need to know the size of it at compile time which means vectors have to live on the heap. The vector works by having a capacity which slowly fills up. Once it is full it resizes. This operation can be quite expensive since it needs to request new memory from the operating system and move all existing items into the new memory region.

4 Material and Method

4.1 Linux

Linux is the operating system used to develop the application and write this paper. The program uses a specific Linux API to acquire the current time as a UNIX time stamp. The way to do this would differ on other operating systems.

4.2 Neovim

Neovim was the editor used to edit the code. It is a modal text editor and not a full integrated development environment, or IDE as they are called. A plugin which utilizes the language server protocol was used to provide me with intellisense and to aid in development by warning me about syntax errors.

4.3 Git

Git is a version control system which helps track changes to the project. In this project it is utilized to keep a time line of the development process and make sure changes could be synced between devices. The local git repository is mirrored on GitHub making it accessible from other devices, and the source code can be read from the browser.

4.4 CMake

CMake is a buildsystem, used to generate generate the final executable from the source code. It aids in compiling and linking the program making cross compilation and dependency management trivial.

4.5 C

C was the program of choice for this project due to its low level control. Since memory is managed manually in C I could be sure the results would not be skewed by a garbage collector running in the middle of a benchmark.

4.6 SDL2

SDL2 - Simple DirectMedia Layer - is a cross-platform library supplying an easy API to create a window and render to it. This was crucial when developing since visualizing the data structures made it easier to spot errors than simply looking at the code. It also gives a nice visual representation of the program.

4.7 Python

Python is an interpreted programming language which was used for making a script to generate graphs. All the graphs used in this paper was generated from data collected from a C program, which dumped said data into a JSON file, which was then read by a Python script using the matplotlib library to generate nice looking graphs.

5 Results & Analysis

This section presents the results and offers an analysis of the benchmarking done on the three strategies - quadtrees, spatial hashing and static grids. The metrics measured was average runtime: the sum of all operations needed to run the program. Insertions: how long it took to insert all the objects into the data structure. Collision testing: total time spent checking objects against each other, which includes querying. Querying: average time spent retrieving objects from the data structure. Clearing: how long it takes to reset the data structure making it ready for rebuilding. Each metric was measured with a range of object count, labeled as 'objects' in the graphs, from ten to 2500 with an increase of ten objects between each measurement. At each stage was run 32 times and the results were averaged to give smoother graphs. Two benchmarks was run, one with an even distribution of objects and one with an uneven distribution.

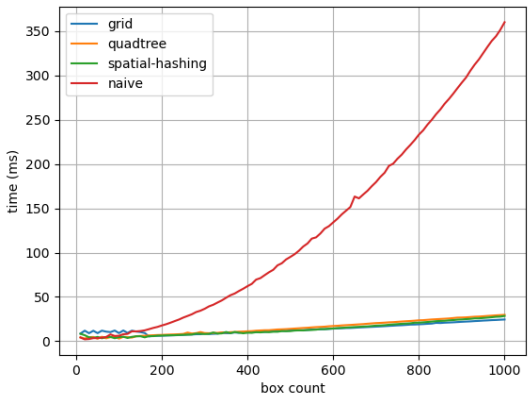


Figure 2: Runtime of naive solution and partitioning strategies.

Figure 2 shows a drastic difference in runtime between the naive solution and the optimization strategies. It is hard to evaluate which strategy performs best due to the stark difference between the naive and the optimized versions. Due to this discrepancy the naive solution will not be included in the other graphs presented. This comparison was only run with 1000 objects due to the time taken per execution.

5.1 Average Runtime

5.1.1 Evenly Distributed Objects

Figure 3 shows the average runtime for the program using the different strategies with an even object distribution. This includes all other measuring metrics. The X axis describes the amount of objects being processed. The Y axis describes how much time was spent in milliseconds executing the particular step in the program.

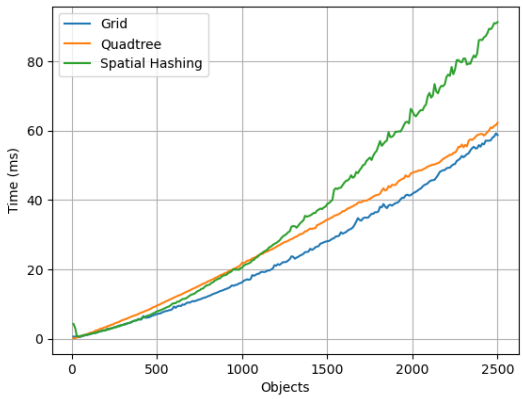


Figure 3: Average runtime of spatial partitioning strategies with evenly distributed objects.

5.1.2 Unevenly Distributed Objects

Figure 4 presents the average runtime for the program using the different strategies with an uneven object distribution. This includes all other measuring metrics. As the object count increases the spatial hashing trends towards a $O(n^2)$ runtime complexity.

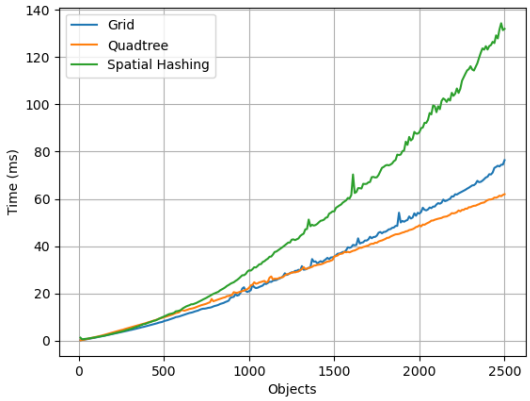


Figure 4: Average runtime of spatial partitioning strategies with unevenly distributed objects.

5.1.3 Analysis

With evenly distributed objects The three strategies perform very similarly, though, spatial hashing does start showing more of an exponential runtime as the object count increases. Could be due to the implementation of the spatial hashing. Since it is using a hash map as its underlying data structure two different cells could have the same index within the same hash map meaning collision checks are happening for objects which

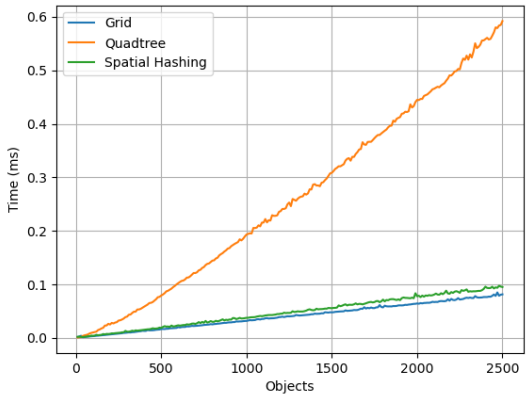
are not close to each other in space. This happens because there is a limited amount of buckets available to store objects inside. The result of this scarcity is two hashes wrapping to the same bucket index. As we can see in 4 this issue becomes more apparent as the objects are less spread out.

5.2 Insertion

5.2.1 Evenly Distributed Objects

The runtime complexity of the insertion algorithms for evenly distributed objects is shown in figure 5. The metric being times is the insertion of all objects.

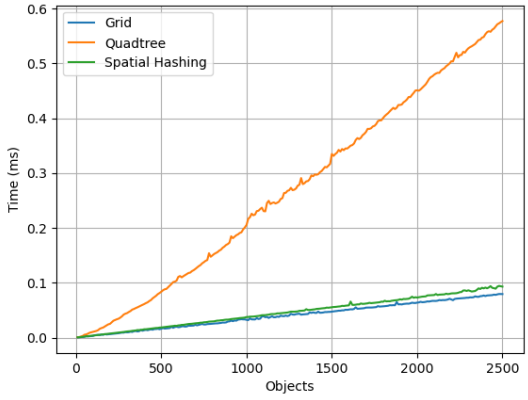
Figure 5: Average insertion time per object with evenly distributed objects.



5.2.2 Unevenly Distributed Objects

The runtime complexity of the insertion algorithms for evenly distributed objects is shown in figure 6. The metric being times is the insertion of all objects.

Figure 6: Average insertion time per object with unevenly distributed objects.



5.2.3 Analysis

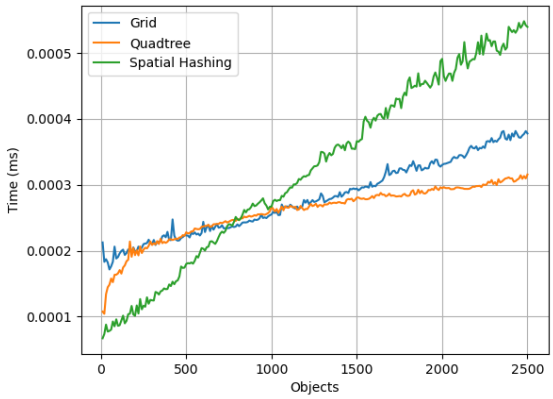
There's virtually no difference in performance between the even and uneven insertion times. This is because the codepath taken for each insertion is the same, save for a different hash and bucket index. Because there is only a fixed amount of objects able to be stored in each grid cell in both grid types the insertion time is run in $O(1)$, constant time. The case isn't the same for the quadtree since its a recursive algorithm. Each node has four sub-nodes which get traversed each insertion leading to a $O(\log_4 n)$ runtime since each step in the algorithm eliminates three of the sub-nodes. Once the correct node has been found the insertion time is a constant $O(1)$ since it uses the same technique as the other two.

5.3 Querying

5.3.1 Evenly Distributed Objects

Figure 7 represents the average time taken to query the space per object with an even distribution.

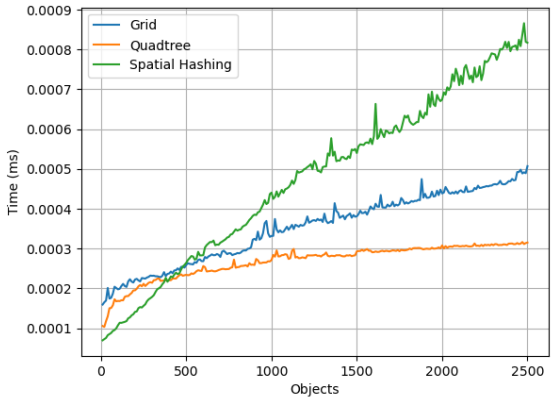
Figure 7: Average query time per object with evenly distributed objects.



5.3.2 Unevenly Distributed Objects

Figure 8 represents the average time taken to query the space per object with an uneven distribution.

Figure 8: Average query time per object with unevenly distributed objects.



5.3.3 Analysis

In both cases the quadtree query has almost exactly the same time with the same $O(\log_4 n)$ runtime complexity. The other two, however, differ quite a bit time wise. This discrepancy is due to the use of vectors. Since a vector is used to return the overlapping objects the act of building this vector becomes more expensive as the amount of objects increases. Since there are more objects near each other in the unevenly distributed objects case there will naturally be more objects added to the vector. Quadtrees do not suffer from the same weakness since it has a limit on the amount of objects stored in each leaf meaning all queries has an upper limit to the number of objects, eliminating the need for vector resizing.

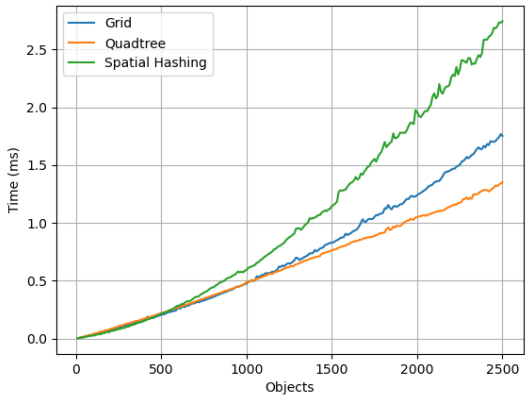
The reason for the jaggedness in the graphs is because the execution time is so low the benchmark picks up on other things like the OS scheduler and other application doing work in the background taking away resources from the program.

5.4 Collision

5.4.1 Evenly Distributed Objects

Figure 9 shows the total time spent checking for colliding objects per iteration with an even distribution. This time also includes the time taken for querying.

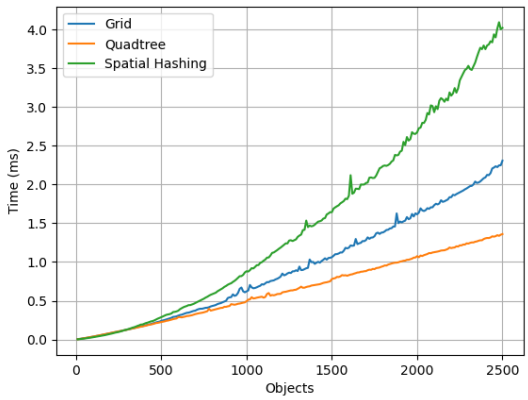
Figure 9: Average collision time for all objects with evenly distributed objects.



5.4.2 Unevenly Distributed Objects

Figure 10 shows the total time spent checking for colliding objects per iteration with an uneven distribution. This time also includes the time taken for querying.

Figure 10: Average collision time for all objects with unevenly distributed objects.



5.5 Analysis

Just like with querying the same thing happens with collisions. Because there are more objects close together there's a higher number of collision checks taking place. Since the quadtree limit the amount of objects in a cell the runtime doesn't change much between the two cases.

6 Discussion

The design choices for this project was entirely focused on CPU performance leaving memory as an afterthought. This was done in order to mimic a modern game running on consumer hardware where memory is often an abundant resource. Everything inside the spatial partitioning strategies were allocated when the structure was initialized so no heap allocations were being issued in the middle of a hot loop. There are, however, two notable exception to this philosophy. When running queries I used vectors, which have to allocate memory on the heap, when doing queries. Looking back, there could be more efficient ways to handle this. Instead of having the query function handle the memory management, let the called handle that complexity. Have the caller allocate an array with sufficient enough size and pass that as a parameter for the function to modify. Though this would work it's not really a good solution, in my subjective opinion. Because there is guarantee of how many objects could be returned. The better solution would be to use a bump, arena or block allocator to further speed up this operation instead of using the general purpose malloc and free pattern provided by the C standard library. The other notable exception is the benchmarking tool I made for this. It uses a hash map to store how many times a function was run, the total time it ran and the average runtime per call. Because of my underlying implementation of the hash map it will require memory allocation when enough entries has been added. Not only that, but it was a nested structure which created a new hash map for each benchmark scope.

The benchmarks are a thing I would have done differently if I would be redoing this project. There are a few decisions that I can't justify now. One of these weird choices is how I chose to benchmark insertions. Instead of running the benchmark for each object inserted I measured to total time taken to insert all objects. This doesn't sound weird until you look at how the rest of the benchmarks are measured. The queries are measured once per object instead of encapsulating all objects in a single benchmark. The insert metric is the only one which differs in this regard. It would be an easy fix code wise, but having to rewrite the paper based off of a new metric would be too much work to justify the change.

I found that although quadtrees outperform all the other strategies in this particular program, it may not be the most suitable for all programs. Both grids and quadtrees limit you to a fixed known world size which could be a problem for larger games or simulations. If the quadtree is covering a massive region of the world then that would introduce significant overhead because there would have to be a lot of subdivisions happening. Chunking could fix this issue though there are other strategies more suitable for this. The spatial hashing grid solves this issue at the cost of performance. Grids come with the same world size restriction that the quadtree does but has the benefit of being extremely trivial to implement. At with almost all questions the answer to 'In what use case are the different algorithms most useful?' the answer is: it depends. For static objects which

cover a known area, the quadtree is by far the most useful. For dynamic objects which move a lot the spatial hashing grid is more suitable. And lastly, the simple grid is really handy when you quickly want to increase the speed of a system without having to spend a lot of time writing the optimization strategy, which works really well for applications with a smaller amount of objects. A combination of multiple strategies could be used to gain the benefit of them, like utilizing the quadtree for static terrain while having all dynamic objects be stored in a spatial hashing gird.

Then at what point is it worth implementing a more advanced data structure? Well that also depends heavily on your particular use case. If the problem only involves checking a handful of objects then the naive solution works just fine. As seen in 2 the naive solution is rarely viable as the object count grows. The only real way to know when it is worth to implement a new optimization strategy is to measure your current solution and make a decision based off of that.

7 Sources

Nystom, Robert (2014) *Game Programming Patterns*, genever benning

Nystom, Robert (2021) *Crafting Interpreters*, genever benning

8 Appendix

The source code written to get these metrics, alongside the python scripts used to generate the graphs, can be found at `https://github.com/faintforge/spatial-partitioning`.