Computer Science Extended Essay

Comparing bins and quadtree optimizations of the boids simulation to the naïve approach.

Word count: 3931

1 Introduction

This investigation focuses on using the bins optimization and the point-region quadtree optimization to improve the performance of boids simulation, which links to Topic 5 – Abstract Data Structures.

Computer simulations are a prevalent tool used in many fields of today's life. They are used in astrophysics to determine the course of events of cosmic phenomena, in games or movies to make animated objects behave more realistically, and in meteorology to forecast the weather. In the majority of cases, the results of these simulations are needed as fast as possible, so they are expected to be as performant as possible. This is why designing and using optimization algorithms is crucial.

In this work, the static and dynamic variants of bins and quadtree optimizations will be compared to the naïve algorithm of conducting the boids simulation. The question, "To what extent do bins and quadtree optimizations improve the performance of the boids simulation?" will be answered.

The bins optimization was chosen for this research because it is frequently used in papers that simulate boids[1] and because the author of the original boids paper stated that such method should improve the performance of the simulation[2]. Quadtree was chosen as a recursive and adaptive data structure, to examine if added complexity causes it to be faster than a simpler technique.

[1] For example: (Prudence M Mavhemwa, 2018), (Alessandro Ribeiro da Silva, 2009)
[2] (Reynolds C. W., 1987)

This investigation aims to benchmark algorithms employing different data structures, identify scenarios in which they are better than others, and verify if more complex techniques can be faster than simpler ones.

2 Background

2.1 Boids

Boids is the name of a simulation first introduced by Craig W. Reynolds in a paper titled "Flocks, Herds, and Schools: A Distributed Behavioral Model", published in 1987[3]. The term originates from the word "bird-oids", meaning "objects behaving like birds". The simulation consists of many individual agents perceiving the environment around them and taking actions based on it. It is sometimes referred to as a more complex version of a particle system[4], as boids are particle-like, yet their behaviour is much more complex than particle's.

The simulation is designed to mimic flocking behaviour. By setting different hyperparameters of the simulation it is possible to model flocks of birds, schools of fish, herds of cattle, or crowds of people. This behaviour of grouping is an emergent property of the system, as flocking is not programmed explicitly into the simulation, but it emerges from many individual boids taking decisions in a set way.

Boids is a microscopic system, not a macroscopic one[5]. This means that each agent makes decisions about the actions to take by itself; there is no "central" system that

[3] (Reynolds C. W., 1987)
[4] (Devlin, 2016)
[5] (Prudence M Mavhemwa, 2018)

would apply forces or guide particles. This trait makes the boids' behaviour more natural, as this decision-making convention is representative of how flocking works in nature.

As of now, boids were introduced 36 years ago and they are still in use. The system is prevalent, as it was used in movies such as *Batman Returns (1992)* to model the behaviour of bats and penguins[6], in *The Lord of The Rings* trilogy to depict large groups of people fighting in a battle[7], or in games like *Grand Theft Auto* to construct background scenes of large groups of people[8].

Boids decide how to move based on three interactions between self and other boids that are in a set radius (neighbours), as illustrated in Figure 1. These forces are cohesion, separation, and alignment.

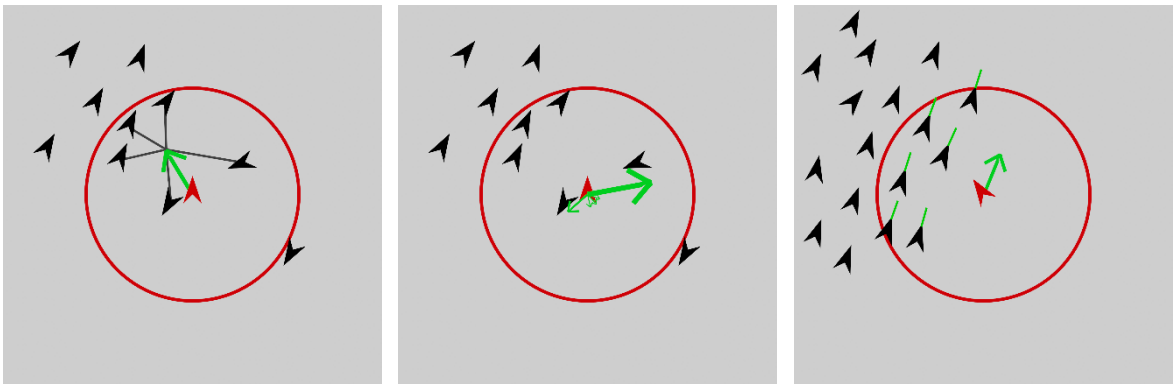

Figure 1: Cohesion (left), separation (middle), and alignment (right) are the three types of interactions between boids.
Source: self-made

[6] (Reynolds C. W., Boids - Background and Update, 1995)
[7] (Alessandro Ribeiro da Silva, 2009)
[8] (Alessandro Ribeiro da Silva, 2009)

Cohesion is an interaction that makes new flocks form, ensures that existing flocks do not fall apart, and causes nearby flocks to merge. Every boid calculates the average position of all its neighbours and then accelerates towards that point.

Separation ensures that boids do not collide. Each boid iterates over all its neighbours and accelerates away from each of them, with a magnitude inversely proportional to the square of the distance between itself and a neighbour. This proportionality makes the repulsion much stronger for very close pairs of boids than for ones that are further away. This interaction combined with cohesion yields some optimal distance of nearby boids, which causes the cohesion and separation to cancel. When the boids move closer or further away, one of the interactions becomes stronger than the other, and the boids come back to that optimal distance.

Alignment makes boids move in a common direction, which organizes the flocks. Each boid calculates the average velocity of its neighbours and then accelerates so that they move in the same direction.

Those three interactions are scaled according to scaling factors which are set in advance as hyperparameters. After calculating its acceleration, each boid updates its velocity, normalizes it to a set common magnitude, and then moves.

To calculate the interactions, each boid identifies its neighbours each frame. The next sections focus on different methods of retrieving a list of neighbours of a given boid.

2.2 Naïve approach

This is the algorithm used in the original paper that first introduced boids. When a boid needs to get a list of all other boids that are closer to it than a given distance, all other boids are checked, the distance is calculated, and compared to the threshold (Figure 2). The author of the original paper states that because all the boids need to calculate their distances to all the other boids, the time complexity of this neighbour-searching algorithm is $O(n^2)$ ($O(n)$ for one query, done for n boids). He admitted that "dynamic spatial partitioning of the flock" would be beneficial to the simulations' performance[9]. Two spatial partitioning techniques which will be examined in this investigation are described in the following two sections.

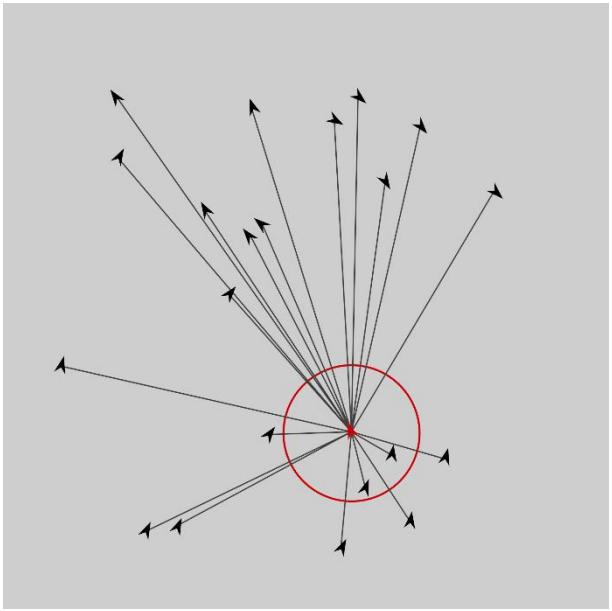

Figure 2: Naïve method – each boid calculates its distance to every other boid. Source: self-made

[9] (Reynolds C. W., Flocks, Herds, and Schools: A Distributed Behavioral Model, 1987)

2.3 Bins

This optimization lowers the time needed to compile a list of all neighbours of a boid by dividing the simulation space into $S \times S$ containers (bins) and, for each container, keeping a list of references to all the boids falling into it. When a boid queries the data structure, the program first checks which bins could contain any boid's neighbours, and then verifies them only inside these bins (Figure 3). It is faster than the naïve approach, as the program doesn't always check for all the potential neighbours, only for the ones that are already close to the querying boid. This reduces the time complexity of this method to $O(n)$ ($O(1)$ for one query, done for n boids)[10]. The constant time of a single query assumes a constant query area and some existing maximum density of objects in the data structure. Both assumptions are met in this use case, as every boid has the same perception radius, and boids keep some minimal distance between each other due to the cohesion-separation equilibrium.

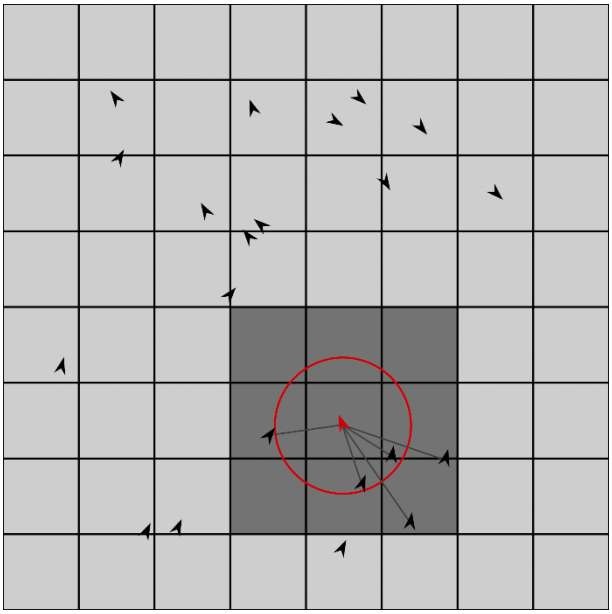

Figure 3: Bins optimization – every boid first identifies bins containing possible neighbours (dark grey), and then verifies each of them. Source: self-made

This data structure is object independent, as it does not change its structure based on the properties or distribution of contained objects. Because of that, in the worst-case

[10] (Reynolds C. , 2000)

scenario, when all the boids form a flock inside one of the bins, the structure will not adapt, and the time complexity will degrade to that of the naïve algorithm. However, this condition is possible only if the size of a bin is big enough to contain all the simulated boids, which would happen only in the case of a low number of bins.

2.4 Quadtree

This recursive data structure is object-dependent – it changes based on the distribution of the objects it holds. Each quadtree node represents a corresponding square region of the simulation space. Each node can either be a leaf or have 4 children. When it is a leaf, it can hold references to a limited number of boids that are located inside the region represented by this node. When the number of boids held by a node exceeds its capacity, the node creates four children, dividing its area into four equal ones and attributing each sub-area to one child. It then passes all the held boids to its children.

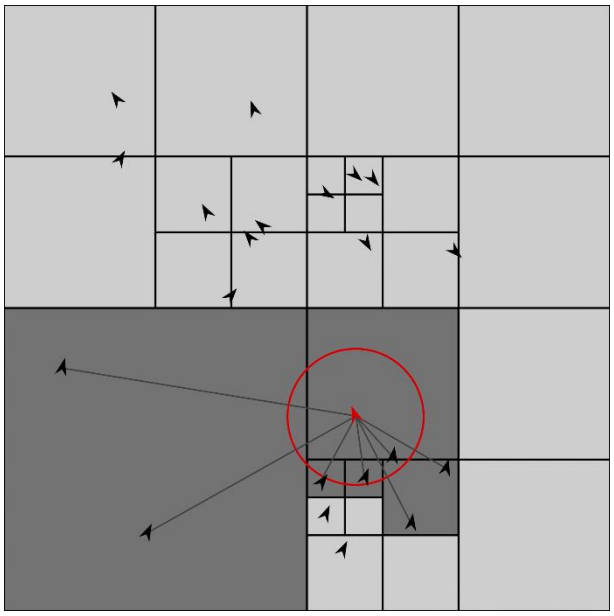

Figure 4: Quadtree optimization (node capacity is equal to 2). Source: self-made

To provide a boid with a list of its neighbours, the program traverses the tree, first calling the quadtree's root. If a called node has children, it checks which of their areas intersect with the queried area. It then calls the intersecting ones. If a called node is a leaf, it checks the distances of its boids to the querying boid and appends the fitting ones to a list which is then returned. The time complexity of this method is $O(n \log n)$

($O(\log n)$ for one query, repeated for n boids)[11]. The distance comparisons are shown in Figure 4. Figure 5 illustrates a quadtree with all its nodes.

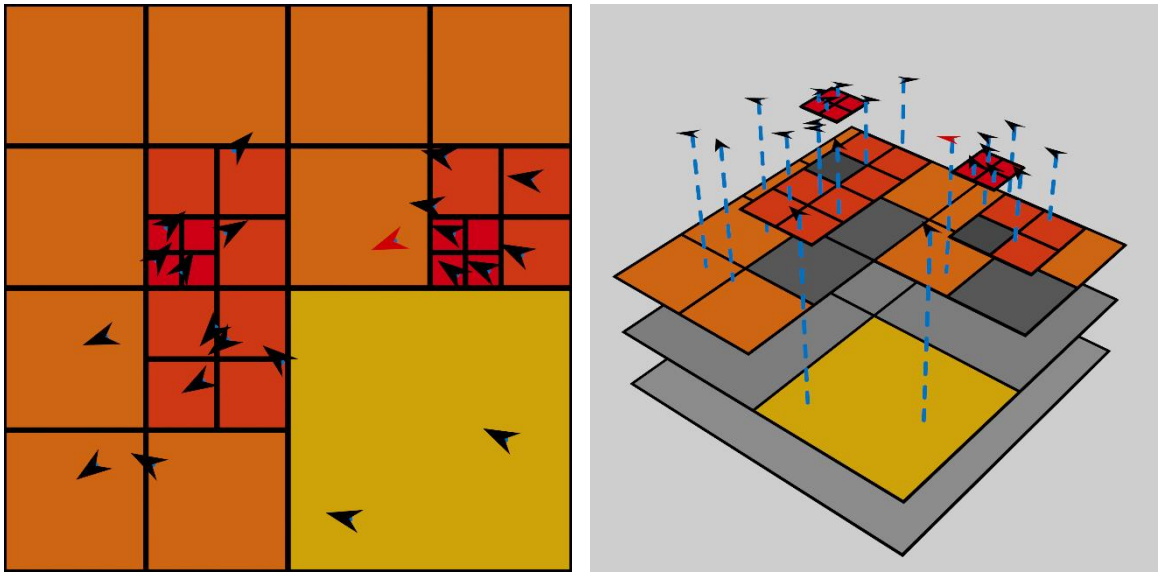

Figure 5: Left – quadtree seen as subdivided regions, only leaves are shown (yellow – sparsely subdivided, red – densely subdivided). Right – all quadtree nodes are shown (grey – parent nodes).
Source: self-made

The adaptivity of quadtrees is beneficial in dealing with uneven distributions of boids. More resources are used in regions of high boid densities, saving them in more sparsely populated areas. The worst-case scenario of the bins optimization can be easily dealt with by a quadtree because this data structure adaptively subdivides so that each node holds up to some limited number of boids.

2.5 Static and Dynamic Data Structures

Static data structures, once constructed, cannot be modified. Dynamic data structures can have their objects' positions changed and they react accordingly. When static data structures are used in a use case which involves constantly moving agents, like in this

[11] (R. A. Finkel, 1974)

investigation, they need to be rebuilt every frame. This is unwanted, as each rebuilding of the data structure involves reallocating memory and re-insertion of all boids.

Implementing dynamic data structures usually makes the code more complicated, as new algorithms have to be introduced. However, there are potential gains resulting from this solution, as usually there is less work to do to modify the data structure than to completely rebuild it.

In the case of the bins optimization, the updating algorithm is straightforward – at the end of a frame, for each bin, the program checks all the contained boids. If a boid is still contained in the bin, nothing changes. Otherwise, its reference is deleted from this bin and added to another, appropriate bin.

In the case of a quadtree, however, the update algorithm is much more complicated. In fact, during the research for this investigation, no paper documenting such algorithm was found. The only two found sources explaining dynamic quadtrees were a StackOverflow answer[12], which explained the algorithm of merging empty leaves of the tree but did not concentrate on the relocation of objects in the tree; and a youtube video[13], which presented code for a dynamic quadtree. In a use case in which every agent moves every frame, the solution from the javidx9's video would not be better than a static quadtree, as every frame every boid would be completely removed from the tree, and then re-inserted (potentially leaving empty leaves, as clean-up was not implemented), even if it did not move outside its current node.

[12] (user4842163, 2018)
[13] (javidx9, 2022)

As a solution, a custom update algorithm was developed for this investigation. For example, consider a boid crossing its leaf’s boundary, as shown in Figure 6.

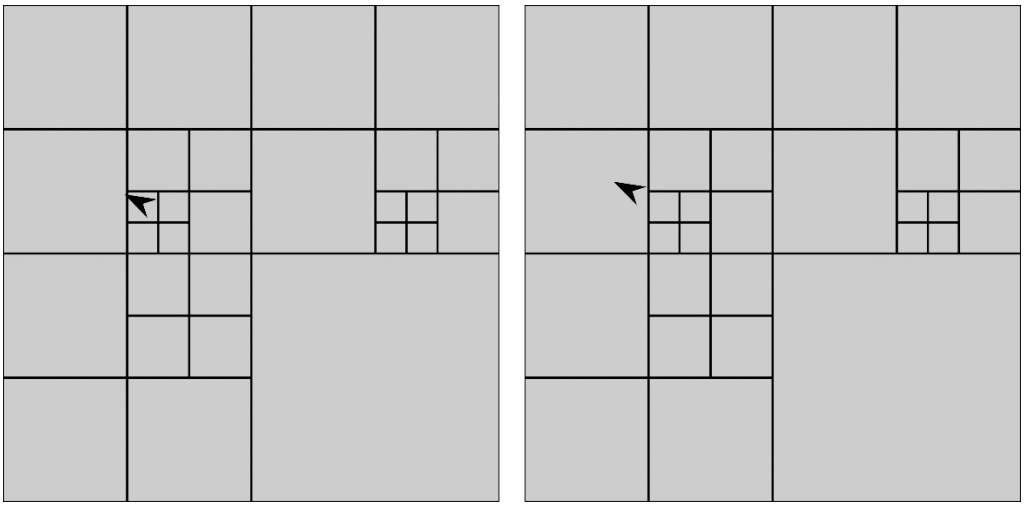

Figure 6: A boid moving across two areas in a quadtree (left: before, right: after). Other boids are not shown for clarity.

Source: self-made

At the end of this frame, the program traverses the tree checking all boids in all leaves. It then identifies this boid as one that is assigned to a wrong node (Figure 7, left). The boid is removed from this node and then passed to the node's ancestors until one that contains the boid is found. This node then inserts the boid into its corresponding child (Figure 7, right). After the relocation of the boid, the parent of the leaf from which the boid was removed checks if the number of boids contained in its children is less than or equal to its capacity. If not, nothing changes. If yes, it intercepts all the boids, marks itself as a leaf, and deletes its children (Figure 7, bottom).

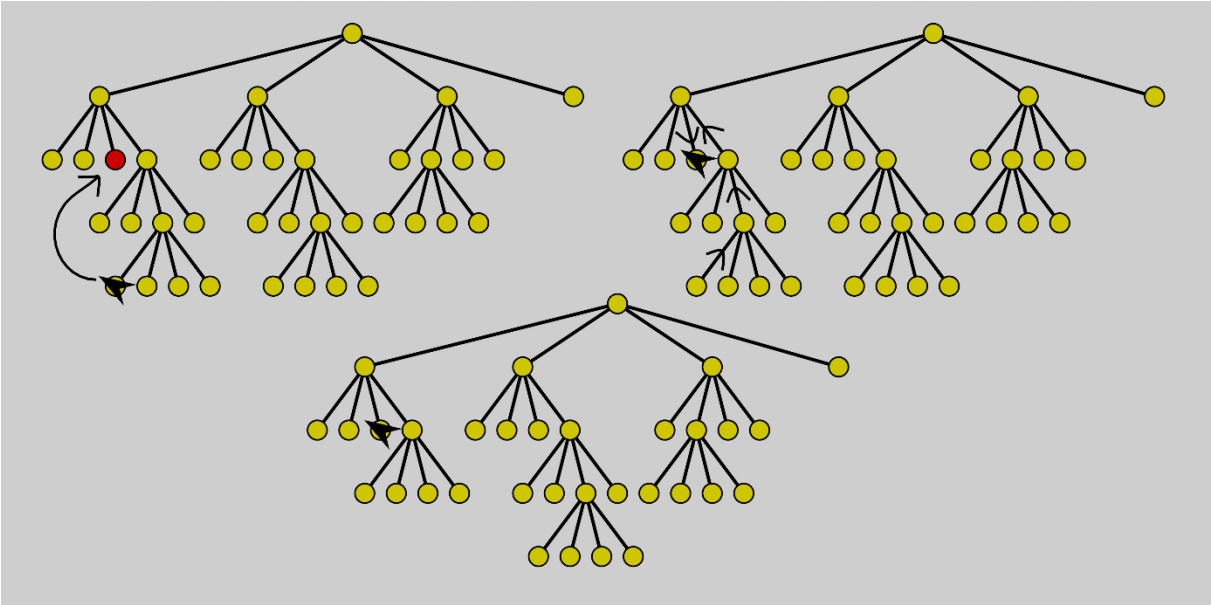

Figure 7: The way the relocation of a boid is conducted.
Source: self-made

3 Hypothesis

The naïve approach is expected to be the least performant of all techniques. Both bins and quadtree methods are expected to run faster due to the reduced amounts of comparisons with potential neighbours per query. The larger the boids number, the more apparent the difference should be. It is expected that bins will perform better than the quadtree because bins optimization has a better time complexity than quadtree optimization. Apart from that, methods employing dynamic data structures are expected to be faster than the ones with static ones, because they only move the objects stored inside, instead of being completely deleted and rebuilt.

The null hypothesis predicts that there is no significant difference between the performances of different methods of retrieving the neighbours of a boid.

4 Implementation

The algorithms were implemented in Java by a single person, employing the same conventions. The program is divided into three modules: main, which contains simulation logic and experiments; finder, which consists of the spatial query techniques implementations; and visualisation, which is used to play the generated animations for debugging.

Each technique of finding objects in a given area was implemented as a separate class in the finder module, implementing the `NeighbourFinder` interface, and employing algorithms for creation, maintenance, and using an according data structure. The classes that use the quadtree optimization define an inner class representing a quadtree node. The complexity of implementing a finder can be described by its line count, as shown on Figure 8.

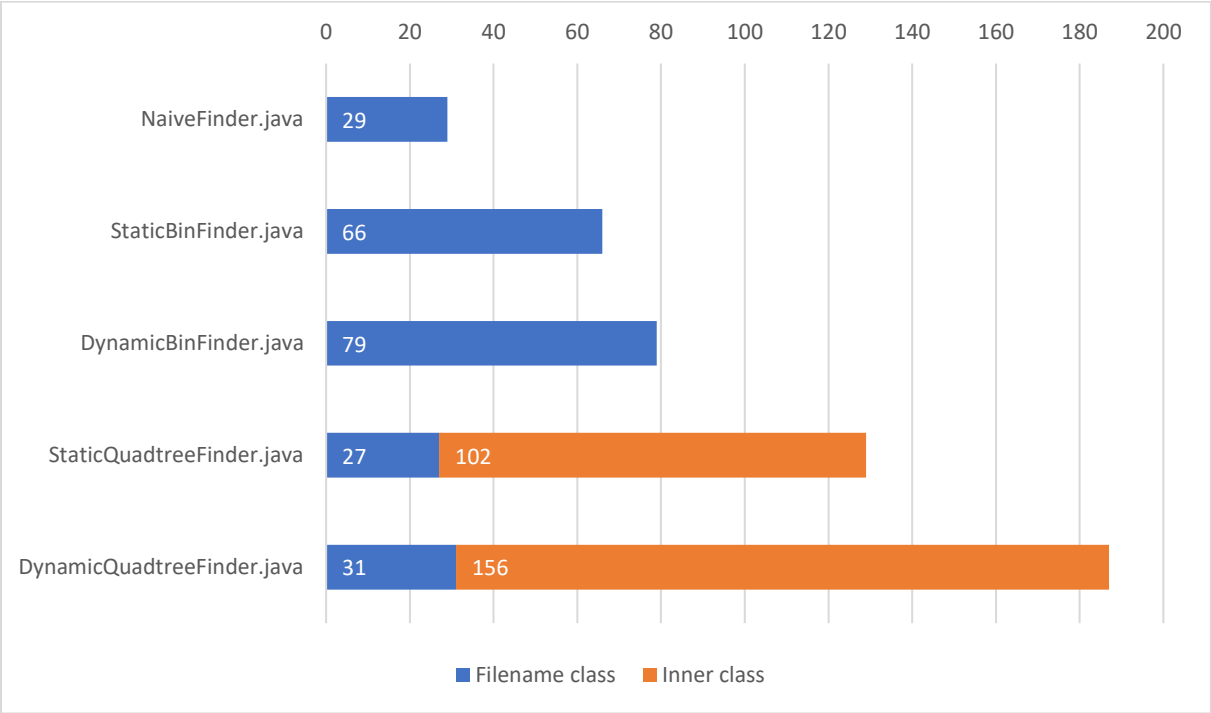

Figure 8: Line count by class. Empty lines, imports and comments are omitted. Source: self-made

The following pseudocode presents how data is passed in the program:

```
finder = new StaticQuadtreeFinder(10)

boidCount = 500
frames = 1000

boids = new SimulationBoid[boidCount]
finder.update(boids)
animationData = new AnimationData(frames)

for f from 0 to frames:
  for boid in boids
        neighbours = finder.findInRadius(boid, boid.radius)
        boid.update(neighbours)

  animationData.addFrame(boids)
  finder.update(boids)

showAnimation(animationData)
```

Figure 9 shows a screenshot of a visualisation of a generated animation:

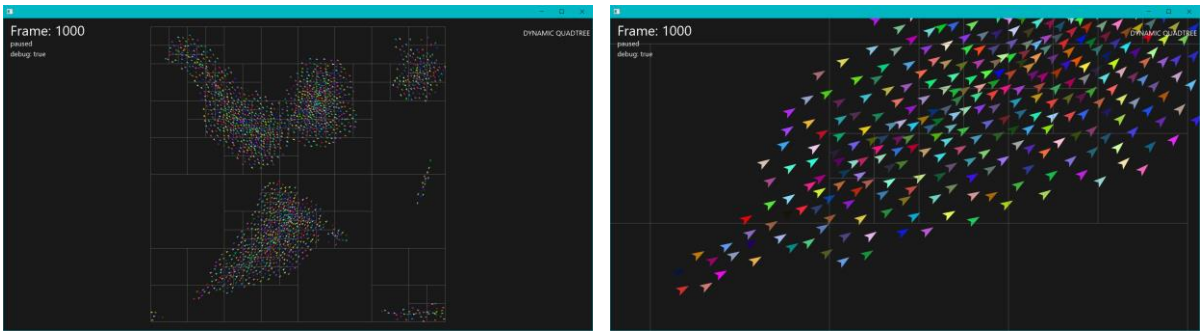

Figure 9: Two screenshots of an animation of 1200 boids. Background lines visualise the structure of a quadtree.
Source: self-made

5 Method

5.1 Experiments

To compare the performances of different techniques, an experiment will be conducted. The experiment will include measuring the time it takes to simulate 500 frames of animation using a given technique. Only simulation time will be measured and the time of displaying the simulation will not influence the results.

To conduct this experiment, some parameters of data structures need to be set. These parameters include the number of subdivisions for the bins algorithm and the capacity of a node for the quadtree algorithm. To decide on these values, one test for each finder needing a parameter will be conducted. The test will determine the time taken to generate an animation against different values of the tested parameter and different boid counts. The value of the parameter for which the algorithm performs the fastest will be considered the best and used in the main experiment.

Additionally, one validation testing if the data structures get degraded over time will be performed. This will be done by examining the time needed to generate an animation of increasing length. If this time is increasing faster than linearly, that means that the data structure takes more time to respond the longer it is used. It should not happen for static data structures, as they are rebuilt every frame, however, the dynamic data structures can become less performant if implemented incorrectly.

5.2 Variables

In all the experiments the dependent variable is the time taken to generate an animation. All the experiments are conducted on the same computer with Intel i7 9750H CPU, with no other programs running. All experiments also share controlled variables like the boids' speed, search radius and scaling factors of interactions multipliers.

5.2.1 Tests of hyperparameter values

In the tests that identify the best parameters for bins and quadtree, the independent variable is the value of the hyperparameter. The experiment is repeated for different

boid counts, to decide if the choice of the parameter should be different for different numbers of boids. The duration of the simulation is controlled and equal to 200 frames.

The value of the independent variable for the bins test will range from 2 to 40 subdivisions. 2 is chosen as the minimum, because 1×1 bins structure is equivalent to the naïve algorithm, so there is no point in testing it. 40 is chosen as the maximum value, because with this subdivision count the search radius spans over slightly under 20 bins, and the time of execution is not expected to be improved by further increasing the parameter.

The capacity of a quadtree node will be tested from 1 to 50. The larger this value, the less the quadtree will subdivide and the more it will resemble the naïve approach. 50 was chosen as the maximum because it is not expected that the performance will improve above it.

5.2.2 Validation of data structures

In this experiment, the independent variable is the number of simulated frames. For each duration, there is a simulation conducted with each querying technique. The boid count is controlled, and equal to 500 boids. The value of the independent variable will be changed from 0 to 4000 frames, which is equivalent to a range from 0 seconds to 1 minute and 20 seconds. This maximum value was chosen because it is considered long enough for any degradation to influence the outcomes.

5.2.3 Experiment testing the performance of different techniques

This experiment has the number of boids as the independent variable. For each value of the independent variable, each technique is evaluated by conducting a simulation.

The simulation length is a controlled variable, equal to 200 frames. Boid counts from 0 to 4000 will be tested, in 200 boids increments.

The length of 500 frames was chosen, as it translates to 10 seconds of animation, which allows all the algorithms needed to simulate boids to influence the outcome in realistic proportions. Creation, maintenance and using a data structure will all influence the time of execution. The maximum boid count of 4000 was chosen because this high number covers the majority of use cases[14].

5.3 Measurements

Because the software is written in Java, a language which uses a garbage collector (GC), some noise is expected in the measurements. The GC can be called by the Java Virtual Machine during the simulation, which will slow down the execution of the program for some period[15]. This is unwanted, as the measurements should only represent the time needed to generate an animation.

To reduce the noise, instead of invoking one simulation for a measurement, ten successive calls of the `generateAnimation()` method are measured, and the median of the durations is calculated. The median ignores extreme values (in this case the fastest and the slowest four executions), so the GC should have no impact on the results unless it has influenced the time of execution of at least 5 simulations out of ten.

[14] (Prudence M Mavhemwa, 2018) uses up to 1290, (L Bajec, 2007) uses 100, (Jakob Kratz, 2021) uses 1000
[15] (Oracle, 2012)

Some background processes, like the work of the operating system, can make the software run slower. To reduce the impact of such events, the measurements of performances of different techniques are interlaced. That means that instead of making all the measurements for one technique at once, then for another, etc., the program makes the measurements by turns.

6 Results and Analysis

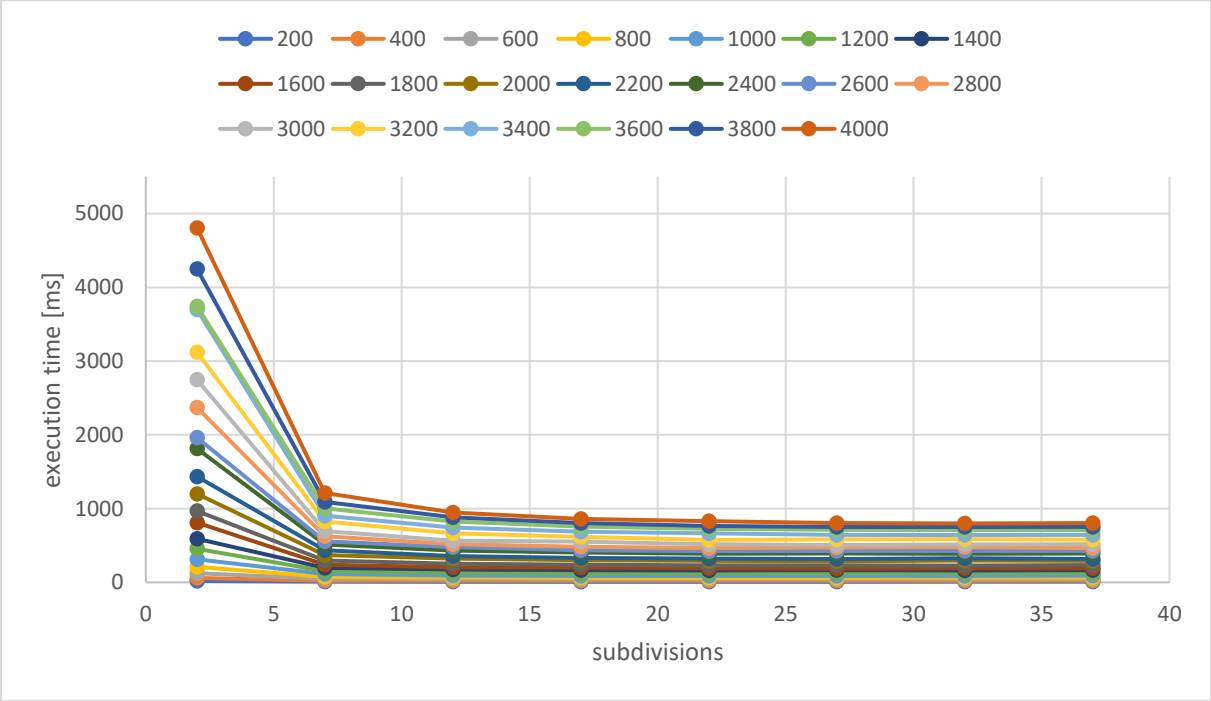

Figure 10: The execution time against the subdivision count of the static bins neighbour finder for different boid counts.
Source: self-made

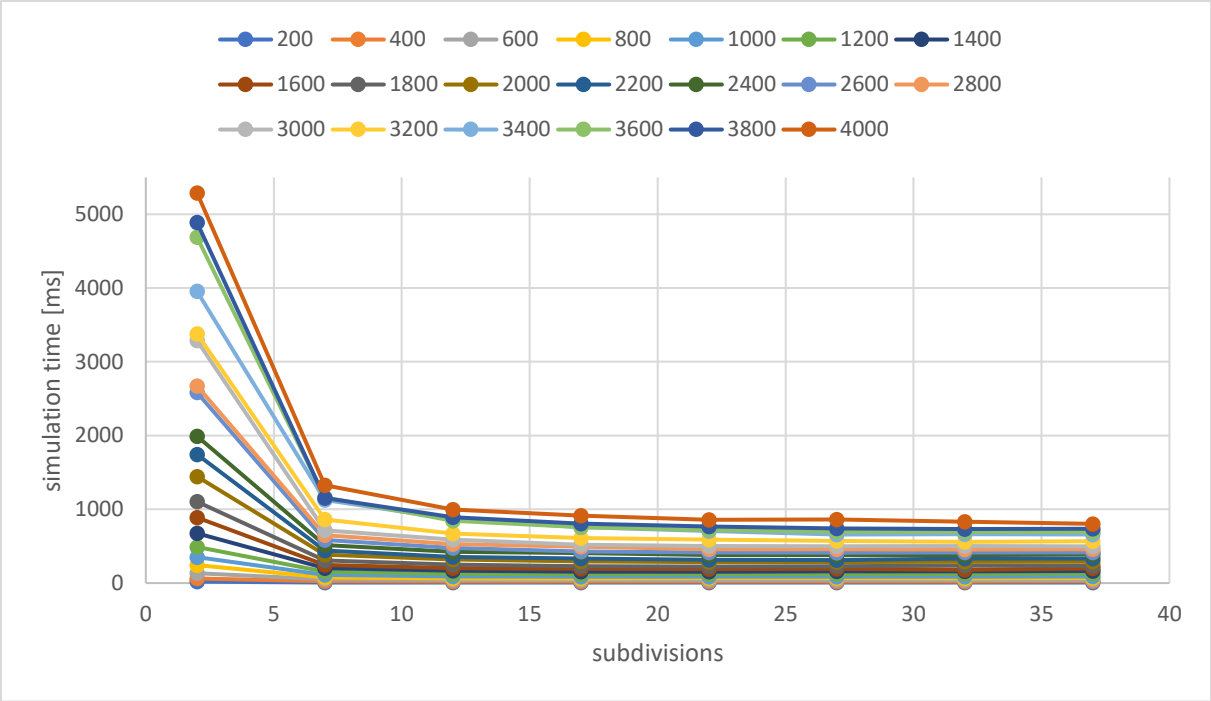

Figure 11: The execution time against the subdivision count of the dynamic bins neighbour finder for different boid counts.
Source: self-made

Figures 10 and 11 prove that both dynamic and static bins finders behave similarly under the same subdivision count. It is shown that the bin neighbour finders whose data structure was subdivided into less than 10x10 bins performed worse than those with more than 10 subdivisions. It seems that, for any number of boids, the value of the hyperparameter does not influence the performance, as long as it is greater than 10. It is caused by the fact that for lower subdivisions the finder has to consider a larger area to find the neighbours of a boid. Based on these observations, the subdivision count of 20 was chosen for both static and dynamic bins finders for further experiments.

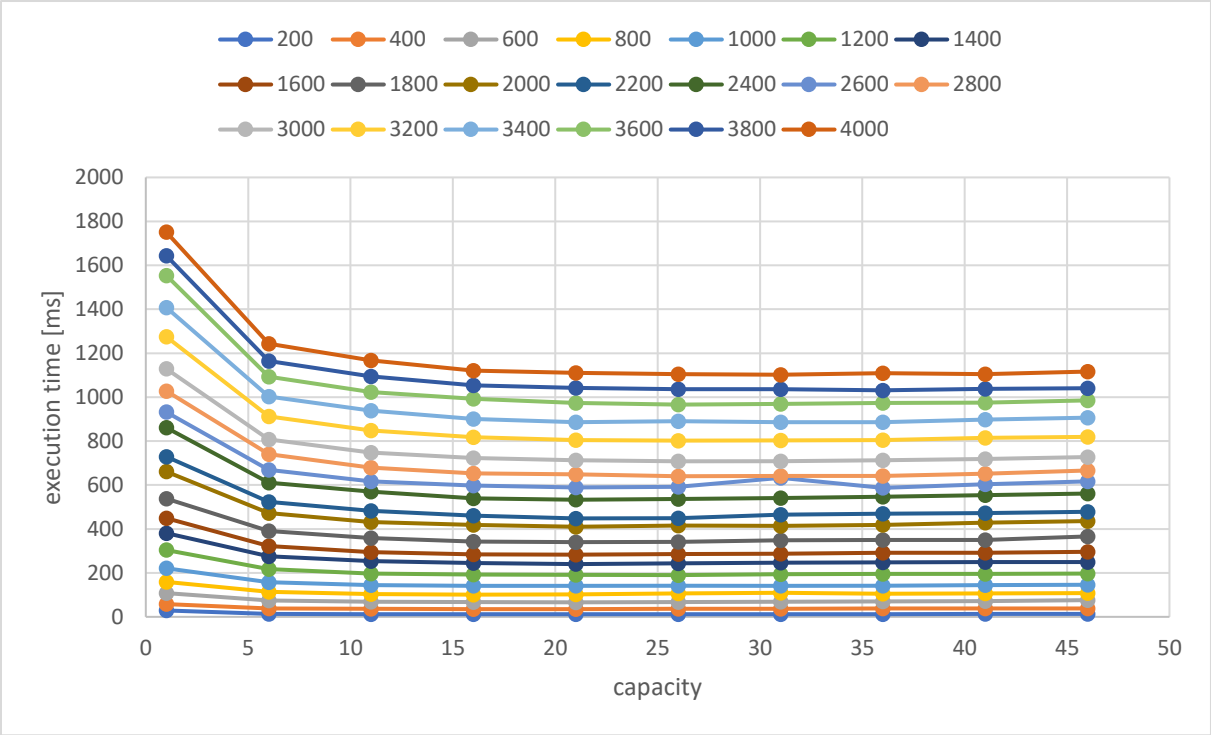

Figure 12: The execution time against the static quadtree's node's capacity for different boid counts.
Source: self-made

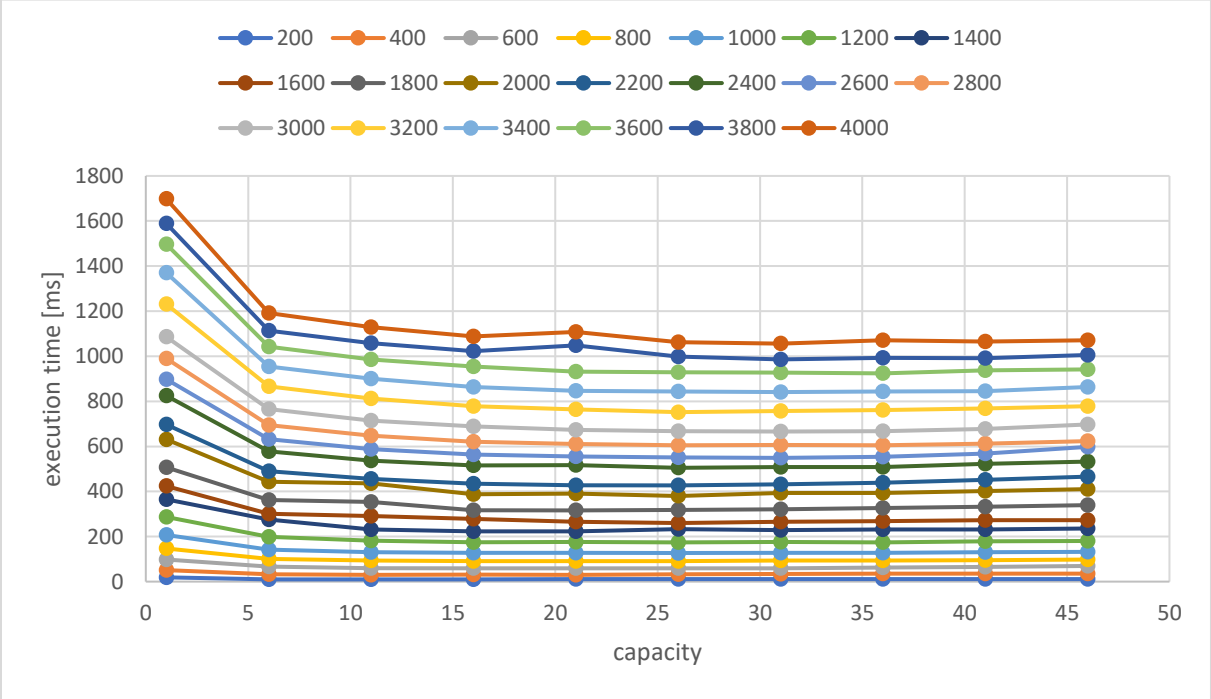

Figure 13: The execution time against the dynamic quadtree's node's capacity for different boid counts.
Source: self-made

Figures 12 and 13 show similar behaviour as figures 10 and 11, however, it seems that the capacity of a quadtree node has less influence on the simulation's performance than the bins subdivisions. The downward curve of the graph is caused by the fact that the lower the capacity of a node, the sooner it will have to subdivide and consequently the tree will be deeper. This means that to retrieve a list of neighbours of a boid the finder needs to traverse more nodes, which takes more time. It is once again noted that the behaviour of dynamic and static data structures is similar. In this case, 25 was chosen as the capacity of a node for comparative experiments.

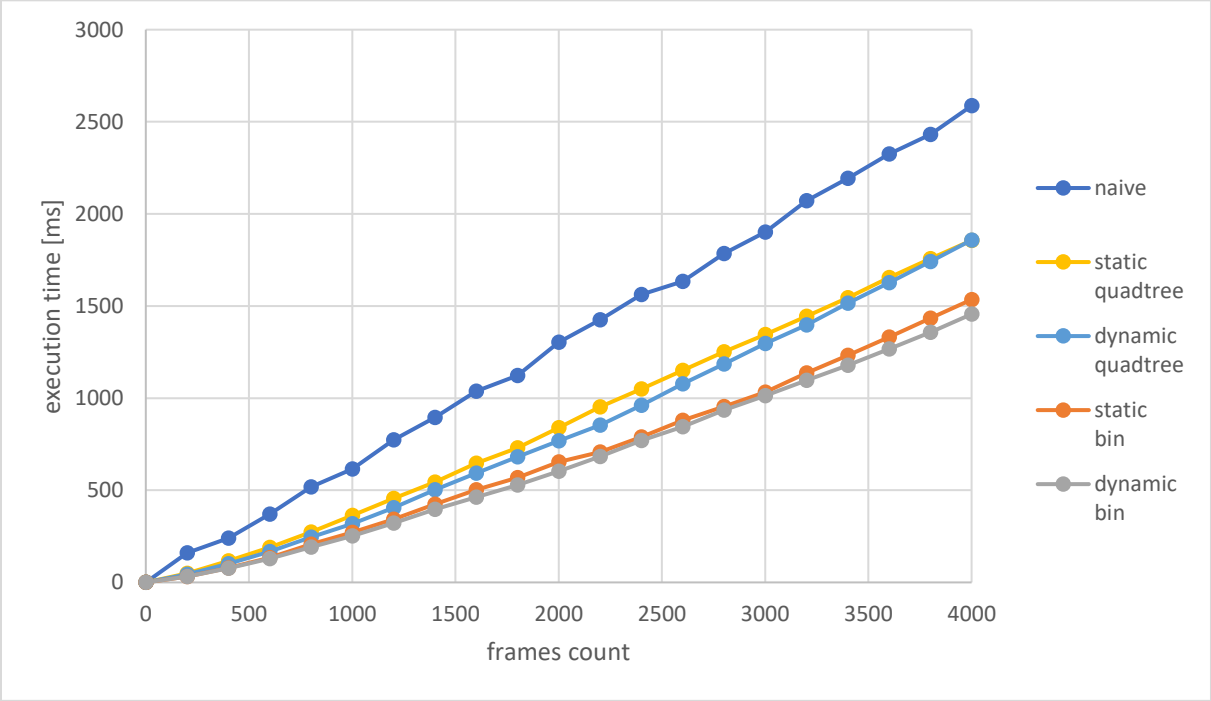

Figure 14: The execution time of all techniques against the simulation duration. Source: self-made.

Figure 14 shows that the time needed to generate an animation is linearly dependent on the length of the animation. This indeed shows that the data structures do not get degraded as the simulation progresses, as they need approximately the same time to process each frame under the same circumstances.

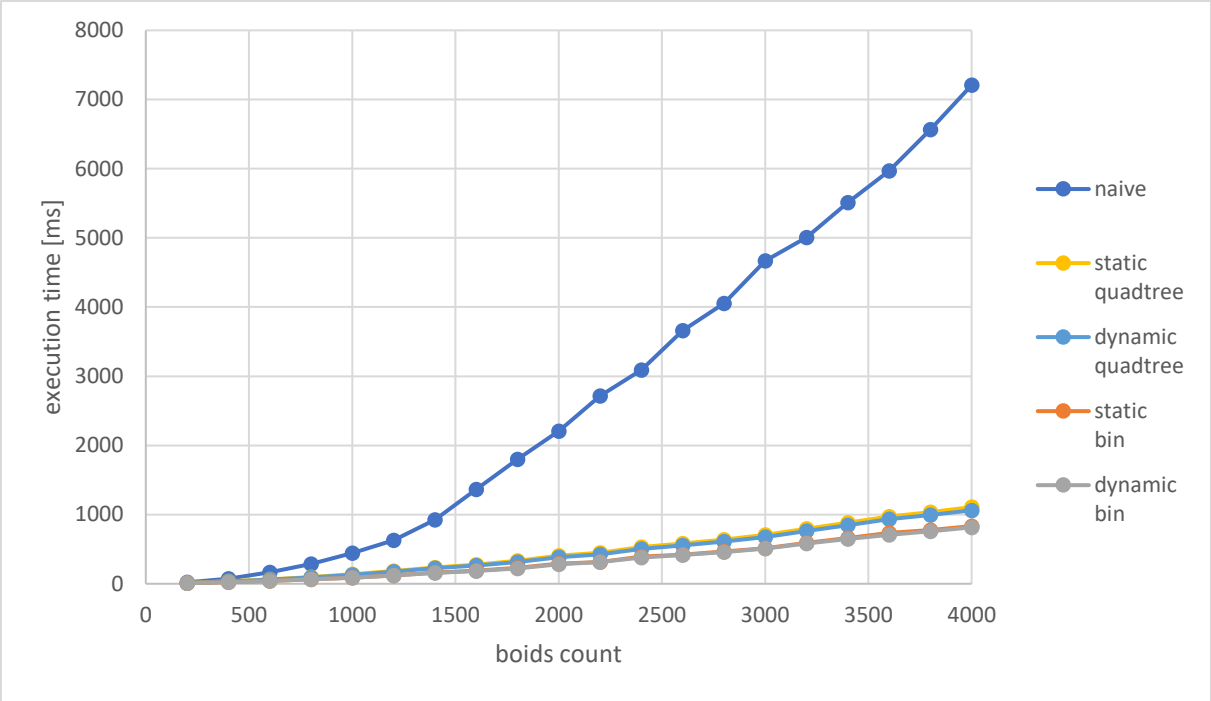

Figure 15: The execution time of all techniques against the boids number. Source: self-made.

On figure 15 it is seen that the naïve approach is overwhelmingly slower than the optimized methods, especially for large boid counts. This observation confirms the hypothesis about the naïve approach being the least performant. The shape of the graph confirms Reynold's statement about the quadratic time complexity of this method[16]. To allow for inspecting the patterns of the four optimized techniques, figure 16 shows the same graph, but magnified:

[16] (Reynolds C. W., Flocks, Herds, and Schools: A Distributed Behavioral Model, 1987)

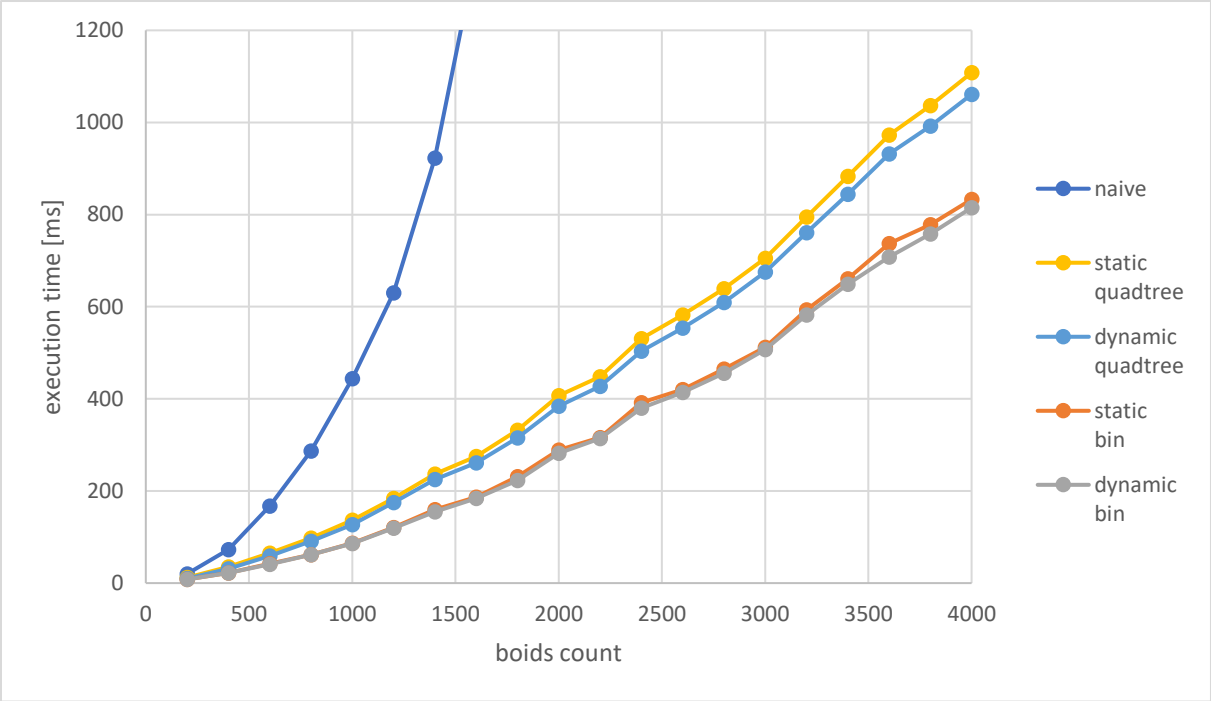

Figure 16: The execution time of all techniques against the boids number, magnified. Source: self-made

Figure 16 shows that the bins optimization is more performant than the quadtree optimization, confirming the hypothesis based on time complexities. In both cases, dynamic data structures are faster than static ones. Despite it being a very small improvement, the difference becomes larger for greater boid numbers.

The lines on the graph do not intersect, which means that in every tested situation the naïve method is the worst, and the bins optimization is the best.

The null hypothesis is rejected, as there is a clear difference in the techniques' performances.

7 Conclusion

The investigation met its aims, and successfully evaluated the postulated hypotheses. The outcomes of the research are useful, as they provide empirical guidelines on the

choice of a fitting spatial partitioning technique for simulations requiring multi-object interactions based on distance.

The naïve technique should never be used in performance-critical cases. The bins optimization is more performant than the quadtree optimization in all checked cases, so it should be always used. The bins optimization is also much easier to implement than the quadtree, as there is no recursion involved in the data structure or any algorithm. It is also reflected on figure 8 – the implementations of bins are about two times shorter than the respective implementations of quadtrees. The code is also less complex, as it does not require any extra class definitions, whereas the quadtree finder needed to define a quadtree node, which took the majority of lines of code.

Although using dynamic data structures improved the program's performance in comparison to using static ones, the change was minute, so it could not always be worth the effort of implementing the logic. As shown on figure 8, in the case of the quadtree, this change introduced 58 new lines of code, whereas in the case of the bins – only 13. It should be also noted that the 13 lines introduced for dynamic bins contained a nested loop and a simple check, but in the case of the quadtree, the 58 lines contained multiple recursive methods. It is concluded that the implementation of a dynamic data structure is beneficial for bins, and not worth the effort for a quadtree.

The answer to the research question is that both bins and quadtree optimizations improve the performance of the boids simulation to great extent, compared to the naïve method, and that bins are faster than a quadtree.

8 Bibliography

Alessandro Ribeiro da Silva, W. S. (2009, December). Boids that See: Using Self-Occlusion for Simulating Large Groups on GPUs. *ACM Computers in Entertainment*.

Devlin, C. (2016, August 22). An Investigation into an Assortment of Flocking Algorithms.

Jakob Kratz, V. L. (2021, June). Comparison of spatial partitioning data structures in crowd simulations.

javidx9. (2022, May). *Quirky Quad Trees Part2: Dynamic Objects In Trees*. Retrieved from YouTube: https://www.youtube.com/watch?v=wXF3HIhnUOg

L Bajec, N. Z. (2007). The computational beauty of flocking: boids revisited. *Mathematical and Computer Modelling of Dynamical Systems: Methods, Tools and Applications in Engineering and Related Sciences*.

Oracle. (2012). *Java Garbage Collection Basics*. Retrieved from Oracle: https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

Prudence M Mavhemwa, I. N. (2018). Uniform spatial subdivision to improve Boids Algorithm in a gaming environment. *International Journal of Advance Research and Development, 3*.

R. A. Finkel, J. L. (1974). Quad Trees - A Data Structure for Retrieval on Composite Keys. *Acta Informatica*, p. 4.

Reynolds, C. (2000). Interaction with Groups of Autonomous Characters. *Game developers conference*.

Reynolds, C. W. (1987, July). Flocks, Herds, and Schools: A Distributed Behavioral Model. *Computer Graphics*.

Reynolds, C. W. (1995, June 29). *Boids - Background and Update*. Retrieved from http://www.red3d.com/cwr/boids/

user4842163. (2018, January 18). *Efficient (and well explained) implementation of a Quadtree for 2D collision detection*. Retrieved from StackOverflow: https://stackoverflow.com/questions/41946007/efficient-and-well-explained-implementation-of-a-quadtree-for-2d-collision-det

9 List of Figures

10 Appendix

10.1 Raw data

Table 1: Data obtained from the execution time against subdivision count experiment using the static bin finder

boids	subdivisions							
	2	**7**	**12**	**17**	**22**	**27**	**32**	**37**
200	19	9	8	8	8	8	9	9
400	60	25	22	22	23	22	22	23
600	124	50	43	42	43	45	45	44
800	210	75	67	63	64	64	66	66
1000	313	112	91	88	88	89	90	92
1200	455	152	131	124	122	125	125	129
1400	591	197	174	164	162	164	163	166
1600	810	240	206	200	190	187	194	194
1800	968	295	254	238	233	231	228	237
2000	1201	370	321	299	293	286	295	291
2200	1435	436	356	330	321	317	325	320
2400	1814	514	431	406	388	390	389	392
2600	1961	556	479	436	424	426	428	425
2800	2371	625	516	484	467	474	477	464
3000	2747	694	567	552	516	507	512	518
3200	3120	826	667	617	576	584	585	578
3400	3698	906	745	687	666	645	642	645
3600	3744	1002	825	757	726	715	709	708
3800	4251	1090	884	805	766	751	751	754
4000	4807	1211	949	862	830	802	799	805

Table 2: Data obtained from the execution time against the subdivision count experiment using the dynamic bin finder

boids	subdivisions							
	2	**7**	**12**	**17**	**22**	**27**	**32**	**37**
200	20	9	8	8	8	8	8	8
400	62	25	22	22	22	22	22	23
600	134	49	42	42	42	43	42	44
800	244	75	65	62	67	67	66	70
1000	352	111	90	87	87	87	87	91
1200	490	154	124	121	121	122	124	128
1400	675	203	167	157	154	158	161	164
1600	889	247	204	193	191	192	181	200
1800	1104	308	249	230	224	227	239	232
2000	1446	383	318	296	284	287	288	288
2200	1745	443	353	331	315	312	335	334
2400	1992	517	423	408	382	379	379	389
2600	2584	585	477	427	416	413	414	416
2800	2674	650	529	493	461	461	455	459
3000	3288	717	588	521	503	499	501	500
3200	3380	862	673	610	587	570	559	567
3400	3956	1126	866	757	706	660	663	659
3600	4690	1162	843	758	720	696	693	695
3800	4889	1153	893	808	768	742	733	737
4000	5291	1326	999	913	857	863	831	803

Table 3: Data obtained from the execution time against node capacity experiment using the static quadtree finder

boids	node capacity									
	1	**6**	**11**	**16**	**21**	**26**	**31**	**36**	**41**	**46**
200	29	13	12	12	12	12	12	12	13	13
400	58	38	36	35	35	36	37	38	38	38
600	108	75	69	67	66	67	68	70	72	76
800	159	114	104	101	102	107	110	105	107	108
1000	221	158	144	141	141	142	141	142	144	146
1200	304	218	197	192	191	190	194	195	196	197
1400	381	276	253	245	240	243	246	248	249	249
1600	449	322	295	284	283	286	287	291	291	296
1800	538	391	359	343	340	341	348	350	350	366
2000	661	472	431	418	410	415	414	418	428	436
2200	729	523	482	461	448	449	465	469	472	478
2400	861	610	570	540	533	536	541	546	554	561
2600	932	669	616	598	589	592	633	587	603	617
2800	1028	741	679	653	648	640	641	642	652	666
3000	1129	807	748	723	712	708	709	712	718	727
3200	1275	913	848	817	804	802	803	805	814	819
3400	1408	1003	939	900	886	891	886	886	897	906
3600	1554	1093	1023	992	974	966	969	973	975	985
3800	1644	1165	1095	1054	1042	1036	1036	1031	1038	1040
4000	1751	1243	1167	1121	1110	1104	1102	1109	1104	1116

Table 4: Data obtained from the execution time against node capacity experiment using the dynamic quadtree finder

boids	node capacity									
	1	**6**	**11**	**16**	**21**	**26**	**31**	**36**	**41**	**46**
200	19	10	10	10	11	11	11	11	11	11
400	51	33	30	31	30	32	34	35	35	35
600	98	66	60	59	59	59	59	63	65	69
800	147	101	93	91	91	91	93	94	95	98
1000	207	142	130	128	128	127	128	128	130	132
1200	287	199	182	175	176	174	176	174	179	180
1400	365	275	232	223	223	233	229	231	231	235
1600	424	301	291	278	265	260	266	268	272	273
1800	507	362	353	317	316	318	321	327	332	339
2000	631	443	436	388	391	380	394	393	402	411
2200	697	490	456	435	428	427	432	439	452	466
2400	825	578	537	516	517	505	509	509	523	533
2600	898	632	588	564	556	551	549	554	568	598
2800	990	695	647	620	610	605	606	605	612	623
3000	1086	765	715	689	673	668	666	667	677	698
3200	1231	867	813	778	764	752	757	761	769	779
3400	1370	954	900	863	846	844	841	843	845	864
3600	1497	1043	985	955	932	929	928	924	937	941
3800	1589	1113	1058	1023	1048	999	986	993	991	1005
4000	1698	1192	1129	1088	1108	1063	1056	1071	1065	1071

Table 5: Data obtained from the execution time against frame count experiment

frames	naive	static bin	dynamic bin	static quadtree	dynamic quadtree
0	0	0	0	0	0
200	161	32	31	49	43
400	241	79	76	117	101
600	371	134	129	190	168
800	519	207	191	273	245
1000	616	271	252	363	318
1200	774	343	321	455	406
1400	896	424	397	544	503
1600	1037	503	462	646	593
1800	1122	568	528	730	681
2000	1303	654	603	839	769
2200	1426	707	684	953	854
2400	1563	790	770	1049	961
2600	1633	880	845	1151	1077
2800	1785	954	936	1251	1186
3000	1901	1033	1014	1345	1297
3200	2071	1137	1096	1444	1398
3400	2193	1232	1178	1545	1515
3600	2325	1331	1267	1655	1627
3800	2431	1433	1358	1757	1741
4000	2587	1535	1456	1856	1858

Table 6: Data obtained from the execution time against the boid count experiment

boids	naive	static bin	dynamic bin	static quadtree	dynamic quadtree
200	20	8	8	12	10
400	73	22	22	35	31
600	167	42	41	65	59
800	287	62	62	98	91
1000	444	87	86	137	127
1200	630	121	119	184	175
1400	923	160	155	237	225
1600	1365	187	184	275	261
1800	1801	231	223	332	315
2000	2206	289	282	407	384
2200	2715	316	314	448	427
2400	3090	392	380	531	504
2600	3659	420	414	582	554
2800	4053	465	456	639	609
3000	4668	512	507	705	675
3200	5006	593	582	795	761
3400	5511	661	649	883	844
3600	5967	737	708	973	932
3800	6568	778	758	1037	992
4000	7207	833	815	1108	1061

10.2 Code

main.Main.java:

```
package main;

import finder.DynamicQuadtreeFinder;
import javafx.application.Application;
import visualisation.AnimationData;
import visualisation.AnimationViewer;

public class Main {

    public static void main(String[] args) {
        // Experiments.commonExperimentNum();

        // use the scroll wheel to zoom, drag to pan, space to pause/play,
arrows to skip one frame, shift+arrows to skip 100 frames
        showAnimation(
                BoidsSimulator.generateAnimation(
                        0,
                        true,
                        1200,
                        2000,
                        0.2,
                        0.05,
                        2,
                        0.005,
                        12,
                        new DynamicQuadtreeFinder(25)
                )
        );

    }


    private static void showAnimation(AnimationData data) {
        AnimationViewer.animationData = data;
        Application.launch(AnimationViewer.class);
    }
}
```

main.SimulationBoid.java

```java
package main;

// A class to hold and update boid data
public class SimulationBoid {
    public double x, y, vX, vY, aX, aY, speed;

    public SimulationBoid(double x, double y, double angle, double speed) {
        this.x = x;
        this.y = y;
        this.vX = Math.cos(angle) * speed;
        this.vY = Math.sin(angle) * speed;
        this.speed = speed;

        aX = 0;
        aY = 0;
    }

    public void step(double dt) {
        vX += aX*dt;
        vY += aY*dt;

        aX = 0;
        aY = 0;

        double length = Math.sqrt(vX*vX + vY*vY);
        vX *= speed / length;
        vY *= speed / length;

        x += vX*dt;
        y += vY*dt;
    }
}
```

main.BoidsSimulator.java

```
package main;

import finder.NeighbourFinder;
import visualisation.AnimatedBoid;
import visualisation.AnimationData;
import visualisation.Line;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class BoidsSimulator {
    public static AnimationData generateAnimation(long seed, boolean debug,
int frames, int boids, double speed, double radius, double cohesion, double
separation, double alignment, NeighbourFinder finder) {

        // hyperparameter
        double dt = 0.02;

        // seed rng
        Random rand = new Random(seed);

        // allocate arrays
        SimulationBoid[] simulationBoids = new SimulationBoid[boids];
        AnimatedBoid[] animatedBoids = new AnimatedBoid[boids];

        String[] debugData = new String[frames];
        ArrayList<Line>[] lines = new ArrayList[frames];

        // initialize boids
        for (int i = 0; i < boids; i++) {
            simulationBoids[i] = new SimulationBoid(rand.nextDouble(),
rand.nextDouble(), rand.nextDouble() * Math.PI*2, speed);
                // radius*0.1 - the size of a boid is ten times smaller
than it's neighbourhood
            animatedBoids[i] = new AnimatedBoid(frames, radius*0.1,
rand.nextDouble(), rand.nextDouble(), rand.nextDouble());
        }

        // simulation's main loop
        for (int f = 0; f < frames; f++) {

            // give the finder all boids to search later
            finder.update(simulationBoids);

            // for each boid...
            for(SimulationBoid b : simulationBoids) {
                // use the finder to get the neighbours
                List<SimulationBoid> neighbours = finder.findInRadius(b,
radius);

                if(neighbours.size() == 0) continue;

                // boids will be boids
                cohere(b, neighbours, cohesion);
                separate(b, neighbours, separation);
```

```
                align(b, neighbours, alignment);
            }

            // that's a separate loop, because we need to calculate each
boid's new speed before moving it, which would influence the behaviour of
other boids
            for(SimulationBoid b : simulationBoids) {
                // move the boid
                b.step(dt);

                // teleport it around so it stays in the square
                if(b.x <  0) b.x += 1;
                if(b.x >= 1) b.x -= 1;
                if(b.y <  0) b.y += 1;
                if(b.y >= 1) b.y -= 1;
            }

            // save the SimulationBoid data to the AnimationBoid
            for (int i = 0; i < boids; i++) {
                animatedBoids[i].x[f] = simulationBoids[i].x;
                animatedBoids[i].y[f] = simulationBoids[i].y;
                animatedBoids[i].angle[f] =
Math.atan2(simulationBoids[i].vY, simulationBoids[i].vX);
            }

            // save debug data
            if(debug) {
                debugData[f] = finder.getDebugInfo();
                lines[f] = finder.getDebugLines();
            }
        }

        // wrap the collected data and return it
        AnimationData data = new AnimationData();
        data.boids = animatedBoids;
        data.dt = dt;
        data.frames = frames;
        data.data = debugData;
        data.lines = lines;
        data.debug = debug;

        return data;
    }

    private static void cohere(SimulationBoid boid, List<SimulationBoid>
neighbours, double factor) {
        // steer towards the average position of all neighbours
        double avX = 0;
        double avY = 0;
        for(SimulationBoid other : neighbours) {
            avX += other.x;
            avY += other.y;
        }
        avX /= neighbours.size();
        avY /= neighbours.size();

        double cohesionX = avX - boid.x;
        double cohesionY = avY - boid.y;
        double cohesionLength = Math.sqrt(cohesionX*cohesionX +
```

```
cohesionY*cohesionY);
        cohesionX *= factor / cohesionLength;
        cohesionY *= factor / cohesionLength;

        boid.aX += cohesionX;
        boid.aY += cohesionY;
    }

    private static void separate(SimulationBoid boid, List<SimulationBoid>
neighbours, double factor) {
        // steer strongly away from close neighbours
        double separationX = 0, separationY = 0;
        for(SimulationBoid other : neighbours) {
            double dx = other.x - boid.x;
            double dy = other.y - boid.y;
            double distSq = dx*dx + dy*dy;

            separationX -= dx / distSq;
            separationY -= dy / distSq;
        }
        separationX *= factor;
        separationY *= factor;

        boid.aX += separationX;
        boid.aY += separationY;
    }

    private static void align(SimulationBoid boid, List<SimulationBoid>
neighbours, double factor) {
        double alignmentX = 0, alignmentY = 0;
        for(SimulationBoid other : neighbours) {
            alignmentX += other.vX;
            alignmentY += other.vY;
        }
        alignmentX /= neighbours.size();
        alignmentY /= neighbours.size();

        alignmentX -= boid.vX;
        alignmentY -= boid.vY;

        alignmentX *= factor;
        alignmentY *= factor;

        boid.aX += alignmentX;
        boid.aY += alignmentY;
    }
}
```

main.Experiments.java

```
package main;

import finder.*;

import java.util.Arrays;

public class Experiments {

    // what subdivision number is the most performant for the bins
optimization?
    // for every boid number and subdivision amount simulates 200 frames of
animation many times, takes the median, and prints a csv (columns -
subdivisions, rows - boids)
    public static void binsSubdivisionsExperiment() {

        for (int boids = 200; boids <= 4000; boids += 200) {
            for (int regions = 2; regions <= 40; regions+=5) {
                int finalRegions = regions;
                measureAndPrint(() -> new StaticBinFinder(finalRegions), 5,
boids, 200, 10);
            }
            System.out.println();
        }
    }

    // what capacity is the most performant for the quadtree optimization?
    // for every boid number and subdivision amount simulates 200 frames of
animation many times, takes the median, and prints a csv (columns -
capacities, rows - boids)
    public static void quadtreeCapacityExperiment() {

        for (int boids = 200; boids <= 4000; boids += 200) {
            for (int capacity = 1; capacity <= 50; capacity+=5) {
                int finalCapacity = capacity;
                measureAndPrint(() -> new
StaticQuadtreeFinder(finalCapacity), 5, boids, 200, 10);
            }
            System.out.println();
        }
    }

    // sorts and returns the median
    private static int getMedian(int[] nums) {
        Arrays.sort(nums);
        return nums.length % 2 == 0 ? ( (nums[nums.length/2] +
nums[nums.length/2-1])/2 ) : ( nums[nums.length/2] );
    }

    // tries all the techniques with different boids numbers, prints the
csv
    public static void commonExperimentNum() {
        for (int boids = 200; boids <= 4000; boids += 200) {

            measureAndPrint(() -> new NaiveFinder(),                    5,
boids, 200, 10);
            measureAndPrint(() -> new StaticBinFinder(20),          5,
```

```
boids, 200, 10);
            measureAndPrint(() -> new DynamicBinFinder(20),         5,
boids, 200, 10);
            measureAndPrint(() -> new StaticQuadtreeFinder(25),   5, boids,
200, 10);
            measureAndPrint(() -> new DynamicQuadtreeFinder(25),  5, boids,
200, 10);

            System.out.println();
        }
    }

    // tries all the techniques with different frame numbers, prints the
csv
    public static void commonExperimentFrames() {
        for (int frames = 0; frames <= 4000; frames += 200) {

            measureAndPrint(() -> new NaiveFinder(),                      5,
500, frames, 10);
            measureAndPrint(() -> new StaticBinFinder(20),          5, 500,
frames, 10);
            measureAndPrint(() -> new DynamicBinFinder(20),         5, 500,
frames, 10);
            measureAndPrint(() -> new StaticQuadtreeFinder(25),   5, 500,
frames, 10);
            measureAndPrint(() -> new DynamicQuadtreeFinder(25),  5, 500,
frames, 10);

            System.out.println();
        }
    }


    private interface FinderSpawner {
        NeighbourFinder spawn();
    }

    // makes {tries} experiments, prints the median
    private static void measureAndPrint(FinderSpawner spawner, long seed,
int boids, int frames, int tries) {
        long start, end;
        int[] outcomes = new int[tries];

        for (int i = 0; i < tries; i++) {

            start = System.currentTimeMillis();
            BoidsSimulator.generateAnimation(seed, false, frames, boids,
0.05, 0.03, 2, 0.005, 12,
                    spawner.spawn());
            end = System.currentTimeMillis();

            outcomes[i] = (int)(end - start);
        }
        System.out.print(getMedian(outcomes) + ",");
    }

}
```

finder.NeighbourFinder.java

```java
package finder;

import visualisation.Line;
import main.SimulationBoid;

import java.util.ArrayList;
import java.util.List;

// an interface for handling queries about boids in a certain area. Each
implementation should take an array of boids as a parameter to the
constructor and store it.
public interface NeighbourFinder {

    List<SimulationBoid> findInRadius(SimulationBoid boid, double r);
    void update(SimulationBoid[] boids);

    ArrayList<Line> getDebugLines();
    String getDebugInfo();
}
```

finder.StaticBinFinder.java

```
package finder;

import visualisation.Line;
import main.SimulationBoid;

import java.util.ArrayList;
import java.util.List;

// A NeighbourFinder implementing the Bins optimization
public class StaticBinFinder implements NeighbourFinder {
    ArrayList<SimulationBoid>[][] bins;
    int size;

    public StaticBinFinder(int size) {
        this.size = size;
        bins = new ArrayList[size][size];

        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                bins[i][j] = new ArrayList<>();
            }
        }
    }

    private int worldToBin(double c) {
        return (int)(c * size);
    }

    @Override
    public List<SimulationBoid> findInRadius(SimulationBoid boid, double r)
{
        List<SimulationBoid> out = new ArrayList<>();

        int minX = Math.max(worldToBin(boid.x - r), 0);
        int maxX = Math.min(worldToBin(boid.x + r), size-1);
        int minY = Math.max(worldToBin(boid.y - r), 0);
        int maxY = Math.min(worldToBin(boid.y + r), size-1);

        for (int i = minY; i <= maxY; i++) {
            for (int j = minX; j <= maxX; j++) {

                for(SimulationBoid b : bins[i][j]) {
                    if(b != boid) {
                        double dx = b.x - boid.x;
                        double dy = b.y - boid.y;

                        if(dx*dx + dy*dy < r*r) {
                            out.add(b);
                        }
                    }
                }

            }
        }

        return out;
```

```
    }

    @Override
    public void update(SimulationBoid[] boids) {
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                bins[i][j].clear();
            }
        }

        for(SimulationBoid b : boids) {
            int i = (int)(b.y * size);
            int j = (int)(b.x * size);

            bins[i][j].add(b);
        }
    }

    @Override
    public ArrayList<Line> getDebugLines() {
        ArrayList<Line> out = new ArrayList<>();
        for (int i = 0; i < size; i++) {
            double t = (double)i/size;
            out.add(new Line(0, t, 1, t));
            out.add(new Line(t, 0, t, 1));
        }
        return out;
    }

    @Override
    public String getDebugInfo() {
        return  "STATIC BINS\n" +
                "size: " + size + "x" + size;
    }
}
```

finder.DynamicBinFinder.java

```
package finder;

import visualisation.Line;
import main.SimulationBoid;

import java.util.ArrayList;
import java.util.List;

// A NeighbourFinder implementing the Bins optimization
public class DynamicBinFinder implements NeighbourFinder {
    ArrayList<SimulationBoid>[][] bins = null;
    int size;

    public DynamicBinFinder(int size) {
        this.size = size;
    }

    private int worldToBin(double c) {
        return (int)(c * size);
    }

    @Override
    public List<SimulationBoid> findInRadius(SimulationBoid boid, double r)
{
        List<SimulationBoid> out = new ArrayList<>();

        int minX = Math.max(worldToBin(boid.x - r), 0);
        int maxX = Math.min(worldToBin(boid.x + r), size-1);
        int minY = Math.max(worldToBin(boid.y - r), 0);
        int maxY = Math.min(worldToBin(boid.y + r), size-1);

        for (int i = minY; i <= maxY; i++) {
            for (int j = minX; j <= maxX; j++) {

                for(SimulationBoid b : bins[i][j]) {
                    if(b != boid) {
                        double dx = b.x - boid.x;
                        double dy = b.y - boid.y;

                        if(dx*dx + dy*dy < r*r) {
                            out.add(b);
                        }
                    }
                }

            }
        }

        return out;
    }

    @Override
    public void update(SimulationBoid[] boids) {
        if(bins == null) {
            bins = new ArrayList[size][size];
```

```
            for (int i = 0; i < size; i++) {
                for (int j = 0; j < size; j++) {
                    bins[i][j] = new ArrayList<>();
                }
            }

            for(SimulationBoid b : boids) {
                int i = (int)(b.y * size);
                int j = (int)(b.x * size);

                bins[i][j].add(b);
            }
        }

        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                for (int k = 0; k < bins[i][j].size();) {
                    SimulationBoid b = bins[i][j].get(k);
                    int boidI = worldToBin(b.y);
                    int boidJ = worldToBin(b.x);
                    if(boidI != i || boidJ != j) {
                        bins[i][j].remove(b);
                        bins[boidI][boidJ].add(b);
                    }
                    else {
                        k++;
                    }
                }
            }
        }
    }

    @Override
    public ArrayList<Line> getDebugLines() {
        ArrayList<Line> out = new ArrayList<>();
        for (int i = 0; i < size; i++) {
            double t = (double)i/size;
            out.add(new Line(0, t, 1, t));
            out.add(new Line(t, 0, t, 1));
        }
        return out;
    }

    @Override
    public String getDebugInfo() {
        return  "DYNAMIC BINS\n" +
                "size: " + size + "x" + size;
    }
}
```

finder.StaticQuadtreeFinder.java

```
package finder;

import visualisation.Line;
import main.SimulationBoid;

import java.util.ArrayList;
import java.util.List;

// A NeighbourFinder implementing the static Quadtree optimization
public class StaticQuadtreeFinder implements NeighbourFinder {
    static class Quadtree {

        double minX, minY, maxX, maxY;
        double width, height;

        Quadtree nw = null;
        Quadtree ne = null;
        Quadtree sw = null;
        Quadtree se = null;

        SimulationBoid[] boids;
        int boidsCount = 0;
        int capacity;
        boolean isLeaf = true;


        public Quadtree(double minX, double minY, double maxX, double maxY,
int capacity) {

            this.minX = minX;
            this.minY = minY;
            this.maxX = maxX;
            this.maxY = maxY;

            width = maxX - minX;
            height = maxY - minY;

            boids = new SimulationBoid[capacity];
            this.capacity = capacity;
        }

        public boolean contains(SimulationBoid b){
            return contains(b.x, b.y);
        }

        public boolean contains(double x, double y) {
            return minX <= x && maxX > x && minY <= y && maxY > y;
        }

        public void insert(SimulationBoid boid){
            if(isLeaf && boidsCount < capacity) {
                boids[boidsCount++] = boid;
            }
            else if(isLeaf) {

                isLeaf = false;
```

```
                nw = new Quadtree(minX, minY, maxX - width / 2, maxY -
height / 2, capacity);
                ne = new Quadtree(minX + width / 2, minY, maxX, maxY -
height / 2, capacity);
                sw = new Quadtree(minX, minY + height / 2, maxX - width /
2, maxY, capacity);
                se = new Quadtree(minX + width / 2, minY + height / 2,
maxX, maxY, capacity);

                for(SimulationBoid b : boids) {
                    if      (nw.contains(b)) nw.insert(b);
                    else if(ne.contains(b)) ne.insert(b);
                    else if(sw.contains(b)) sw.insert(b);
                    else if(se.contains(b)) se.insert(b);
                    else System.err.println("Error: Boid (" + boid.x + ", "
+ boid.y + ") lost in quadtree insertion.");
                }
                if      (nw.contains(boid)) nw.insert(boid);
                else if(ne.contains(boid)) ne.insert(boid);
                else if(sw.contains(boid)) sw.insert(boid);
                else if(se.contains(boid)) se.insert(boid);
                else System.err.println("Error: Boid (" + boid.x + ", " +
boid.y + ") lost in quadtree insertion.");

                boidsCount = 0;
            }
            else {

                if      (nw.contains(boid)) nw.insert(boid);
                else if(ne.contains(boid)) ne.insert(boid);
                else if(sw.contains(boid)) sw.insert(boid);
                else if(se.contains(boid)) se.insert(boid);
                else System.err.println("Error: Boid (" + boid.x + ", " +
boid.y + ") lost in quadtree insertion.");
                }
        }

        public ArrayList<SimulationBoid> getFromRegion(double minX, double
minY, double maxX, double maxY) {
            return getFromRegion(minX, minY, maxX, maxY, new
ArrayList<>());
        }
        private ArrayList<SimulationBoid> getFromRegion(double minX, double
minY, double maxX, double maxY, ArrayList<SimulationBoid> list) {
            if(isLeaf) {
                for (int i = 0; i < boidsCount; i++) {
                    SimulationBoid b = boids[i];
                    if(minX <= b.x && maxX > b.x && minY <= b.y && maxY >
b.y) list.add(b);
                }
            }
            else {
                if(doRegionsIntersect(nw.minX, nw.maxX, nw.minY, nw.maxY,
minX, maxX, minY, maxY))
                    nw.getFromRegion(minX, minY, maxX, maxY, list);
                if(doRegionsIntersect(ne.minX, ne.maxX, ne.minY, ne.maxY,
minX, maxX, minY, maxY))
                    ne.getFromRegion(minX, minY, maxX, maxY, list);
                if(doRegionsIntersect(sw.minX, sw.maxX, sw.minY, sw.maxY,
```

```
minX, maxX, minY, maxY))
                    sw.getFromRegion(minX, minY, maxX, maxY, list);
                if(doRegionsIntersect(se.minX, se.maxX, se.minY, se.maxY,
minX, maxX, minY, maxY))
                    se.getFromRegion(minX, minY, maxX, maxY, list);
            }

            return list;
        }

        private static boolean doRegionsIntersect(double lx1, double hx1,
double ly1, double hy1, double lx2, double hx2, double ly2, double hy2) {
            return  lx1 <= hx2 &&
                    hx1 >= lx2 &&
                    ly1 <= hy2 &&
                    hy1 >= ly2;
        }

        private ArrayList<Line> getLines(ArrayList<Line> list) {

            if(!isLeaf) {
                list.add(new Line((minX + maxX)/2, minY, (minX + maxX)/2,
maxY));
                list.add(new Line(minX, (minY + maxY)/2, maxX, (minY +
maxY)/2));
                nw.getLines(list);
                ne.getLines(list);
                sw.getLines(list);
                se.getLines(list);
            }

            return list;
        }

        public ArrayList<Line> getLines() {
            return getLines(new ArrayList<>());
        }
    }

    Quadtree tree;

    public StaticQuadtreeFinder(int capacity) {
        tree = new Quadtree(0, 0, 1, 1, capacity);
    }

    @Override
    public List<SimulationBoid> findInRadius(SimulationBoid boid, double r)
{
        List<SimulationBoid> neighbours = tree.getFromRegion(boid.x-r,
boid.y-r, boid.x+r, boid.y+r);

        neighbours.removeIf(b -> b==boid || (b.x - boid.x)*(b.x - boid.x) +
(b.y - boid.y)*(b.y - boid.y) > r*r);

        return neighbours;
    }

    @Override
    public void update(SimulationBoid[] boids) {
```

```
        tree = new Quadtree(0, 0, 1, 1, tree.capacity);

        for(SimulationBoid b : boids) {
            tree.insert(b);
        }
    }

    @Override
    public ArrayList<Line> getDebugLines() {
        return tree.getLines();
    }

    @Override
    public String getDebugInfo() {
        return "STATIC QUADTREE";
    }
}
```

finder.DynamicQuadtreeFinder.java

```
package finder;

import visualisation.Line;
import main.SimulationBoid;

import java.util.ArrayList;
import java.util.List;

// A NeighbourFinder implementing the dynamic Quadtree optimization
public class DynamicQuadtreeFinder implements NeighbourFinder {
    static class Quadtree {

        double minX, minY, maxX, maxY;
        double width, height;

        Quadtree parent;

        Quadtree nw = null;
        Quadtree ne = null;
        Quadtree sw = null;
        Quadtree se = null;

        SimulationBoid[] boids;
        int boidsCount = 0;
        final int capacity;
        boolean isLeaf = true;


        public Quadtree(Quadtree parent, double minX, double minY, double
maxX, double maxY, int capacity) {

            this.parent = parent;

            this.minX = minX;
            this.minY = minY;
            this.maxX = maxX;
            this.maxY = maxY;

            width = maxX - minX;
            height = maxY - minY;

            boids = new SimulationBoid[capacity];
            this.capacity = capacity;
        }

        public boolean contains(SimulationBoid b){
            return contains(b.x, b.y);
        }

        public boolean contains(double x, double y) {
            return minX <= x && maxX > x && minY <= y && maxY > y;
        }

        public void insert(SimulationBoid boid){
            if(isLeaf && boidsCount < capacity) {
                boids[boidsCount++] = boid;
```

```
            }
            else if(isLeaf) {

                isLeaf = false;
                nw = new Quadtree(this, minX, minY, maxX - width / 2, maxY
- height / 2, capacity);
                ne = new Quadtree(this, minX + width / 2, minY, maxX, maxY
- height / 2, capacity);
                sw = new Quadtree(this, minX, minY + height / 2, maxX -
width / 2, maxY, capacity);
                se = new Quadtree(this, minX + width / 2, minY + height /
2, maxX, maxY, capacity);

                for(SimulationBoid b : boids) {
                    if     (!contains(b)) parent.reallocate(b);
                    else if(nw.contains(b)) nw.insert(b);
                    else if(ne.contains(b)) ne.insert(b);
                    else if(sw.contains(b)) sw.insert(b);
                    else if(se.contains(b)) se.insert(b);
                    else System.err.println("A. Error: Boid (" + boid.x +
", " + boid.y + ") lost in quadtree insertion.");
                }
                if     (nw.contains(boid)) nw.insert(boid);
                else if(ne.contains(boid)) ne.insert(boid);
                else if(sw.contains(boid)) sw.insert(boid);
                else if(se.contains(boid)) se.insert(boid);
                else System.err.println("B. Error: Boid (" + boid.x + ", "
+ boid.y + ") lost in quadtree insertion.");

                boidsCount = 0;
            }
            else {

                if     (nw.contains(boid)) nw.insert(boid);
                else if(ne.contains(boid)) ne.insert(boid);
                else if(sw.contains(boid)) sw.insert(boid);
                else if(se.contains(boid)) se.insert(boid);
                else System.err.println("C. Error: Boid (" + boid.x + ", "
+ boid.y + ") lost in quadtree insertion.");
            }
        }

        // a function to check if the boids are still in the proper node,
and if not, reallocate them, returns true if it removed a boid
        public boolean update() {
            if(!isLeaf) {
                boolean changed;
                changed  = nw.update();
                changed |= ne.update();
                changed |= sw.update();
                changed |= se.update();

                if(changed) {
                    return attemptMerging();
                }

                return false;
            }
            else {
```

```
                boolean changed = false;
                for (int i = 0; i < boidsCount; ) {
                    SimulationBoid boid = boids[i];
                    if(contains(boid)) {
                        i++;
                        continue;
                    }

                    // delete the reference to the deleted boid by
overwriting it with the last one
                    boids[i] = boids[--boidsCount];

                    parent.reallocate(boid);

                    changed = true;
                }

                return changed;
            }
        }

        // the node checks if its children can be deleted and if so,
deletes them and takes all the boids. Returns true if it merged
        public boolean attemptMerging() {
            int sum = nw.boidsCount + ne.boidsCount + sw.boidsCount +
se.boidsCount;

            if(sum <= capacity && nw.isLeaf && ne.isLeaf && sw.isLeaf &&
se.isLeaf) {
                for (int i = 0; i < nw.boidsCount; i++) boids[boidsCount++]
= nw.boids[i];
                for (int i = 0; i < ne.boidsCount; i++) boids[boidsCount++]
= ne.boids[i];
                for (int i = 0; i < sw.boidsCount; i++) boids[boidsCount++]
= sw.boids[i];
                for (int i = 0; i < se.boidsCount; i++) boids[boidsCount++]
= se.boids[i];

                isLeaf = true;
                nw = null;
                ne = null;
                sw = null;
                se = null;

                return true;
            }

            return false;
        }

        // called by children which pass a boid that is outside of their
bounds
        public void reallocate(SimulationBoid boid) {
            if(contains(boid)) {
                insert(boid);
            }
            else {
                parent.reallocate(boid);
            }
```

```
        }

        public ArrayList<SimulationBoid> getFromRegion(double minX, double
minY, double maxX, double maxY) {
            return getFromRegion(minX, minY, maxX, maxY, new
ArrayList<>());
        }
        private ArrayList<SimulationBoid> getFromRegion(double minX, double
minY, double maxX, double maxY, ArrayList<SimulationBoid> list) {
            if(isLeaf) {
                for (int i = 0; i < boidsCount; i++) {
                    SimulationBoid b = boids[i];
                    if(minX <= b.x && maxX > b.x && minY <= b.y && maxY >
b.y) list.add(b);
                }
            }
            else {
                if(doRegionsIntersect(nw.minX, nw.maxX, nw.minY, nw.maxY,
minX, maxX, minY, maxY))
                    nw.getFromRegion(minX, minY, maxX, maxY, list);
                if(doRegionsIntersect(ne.minX, ne.maxX, ne.minY, ne.maxY,
minX, maxX, minY, maxY))
                    ne.getFromRegion(minX, minY, maxX, maxY, list);
                if(doRegionsIntersect(sw.minX, sw.maxX, sw.minY, sw.maxY,
minX, maxX, minY, maxY))
                    sw.getFromRegion(minX, minY, maxX, maxY, list);
                if(doRegionsIntersect(se.minX, se.maxX, se.minY, se.maxY,
minX, maxX, minY, maxY))
                    se.getFromRegion(minX, minY, maxX, maxY, list);
            }

            return list;
        }

        private static boolean doRegionsIntersect(double lx1, double hx1,
double ly1, double hy1, double lx2, double hx2, double ly2, double hy2) {
            return  lx1 <= hx2 &&
                    hx1 >= lx2 &&
                    ly1 <= hy2 &&
                    hy1 >= ly2;
        }

        private ArrayList<Line> getLines(ArrayList<Line> list) {

            if(!isLeaf) {
                list.add(new Line((minX + maxX)/2, minY, (minX + maxX)/2,
maxY));
                list.add(new Line(minX, (minY + maxY)/2, maxX, (minY +
maxY)/2));
                nw.getLines(list);
                ne.getLines(list);
                sw.getLines(list);
                se.getLines(list);
            }

            return list;
        }

        public ArrayList<Line> getLines() {
```

```
            return getLines(new ArrayList<>());
        }
    }

    Quadtree tree;

    public DynamicQuadtreeFinder(int capacity) {
        tree = new Quadtree(null, 0, 0, 1, 1, capacity);
    }

    @Override
    public List<SimulationBoid> findInRadius(SimulationBoid boid, double r)
{
        List<SimulationBoid> neighbours = tree.getFromRegion(boid.x-r,
boid.y-r, boid.x+r, boid.y+r);

        neighbours.removeIf(b -> b==boid || (b.x - boid.x)*(b.x - boid.x) +
(b.y - boid.y)*(b.y - boid.y) > r*r);

        return neighbours;
    }

    @Override
    public void update(SimulationBoid[] boids) {
        if(tree.isLeaf) {
            for (SimulationBoid b : boids) {
                tree.insert(b);
            }
        }
        else {
            tree.update();
        }
    }

    @Override
    public ArrayList<Line> getDebugLines() {
        return tree.getLines();
    }

    @Override
    public String getDebugInfo() {
        return "DYNAMIC QUADTREE";
    }
}
```

visualisation.Line.java

```
package visualisation;

// a POJO to hold line data
public class Line {
    double x1, y1, x2, y2;

    public Line(double x1, double y1, double x2, double y2) {
        this.x1 = x1;
        this.y1 = y1;
        this.x2 = x2;
        this.y2 = y2;
    }
}
```

visualisation.AnimatedBoid.java

```
package visualisation;

// a class for storing data about the evolution of a boid, created during
the simulation
public class AnimatedBoid {
    public double[] x;
    public double[] y;
    public double[] angle;
    public double r;
    public double red;
    public double green;
    public double blue;

    public AnimatedBoid(int frames, double radius, double red, double
green, double blue) {
        x = new double[frames];
        y = new double[frames];
        angle = new double[frames];
        r = radius;

        this.red = red;
        this.green = green;
        this.blue = blue;
    }
}
```

visualisation.AnimationData.java

```
package visualisation;

import java.util.ArrayList;

// a class holding all the needed information for the AnimationViewer to
operate
public class AnimationData {
    public boolean debug;
    public AnimatedBoid[] boids;
    public double dt;
    public int frames;
    public String[] data;
    public ArrayList<Line>[] lines;
}
```

visualisation.AnimationViewer.java

```
package visualisation;

import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.input.KeyCode;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.stage.Stage;
import javafx.util.Duration;

// a class extending javaFx's Application, responsible for viewing the
generated simulation
public class AnimationViewer extends Application {
    double width = 1700, height = 900;

    static Canvas canvas;
    static GraphicsContext ctx;
    static StackPane root;
    static Scene scene;
    static Stage stage;

    public static AnimationData animationData;
    static int currentFrame = 0;
    static boolean isPlaying = false;

    static double viewX = 0.5, viewY = 0.5, viewSize = 2;

    static double lastDragX, lastDragY;

    @Override
    public void start(Stage primaryStage) {
        // initialize JFX stuff
        canvas = new Canvas(width, height);
        ctx = canvas.getGraphicsContext2D();
        root = new StackPane(canvas);
        scene = new Scene(root, width, height);
        stage = primaryStage;

        // canvas rescaling
        scene.widthProperty().addListener(e -> {
            canvas.setWidth(scene.getWidth());
            width = scene.getWidth();
        });
        scene.heightProperty().addListener(e -> {
            canvas.setHeight(scene.getHeight());
            height = scene.getHeight();
        });

        // drawing routine
        Timeline tl = new Timeline(new
KeyFrame(Duration.seconds(animationData.dt), e -> {
```

```
            currentFrame %= animationData.frames;
            if(currentFrame < 0) currentFrame =
animationData.boids[0].x.length-1;

            drawFrame(currentFrame);

            if(isPlaying) {
                currentFrame++;
            }
        }));
        tl.setCycleCount(-1);
        tl.play();

        // pan and zoom logic
        scene.setOnScroll(e -> {
            double mult = Math.pow(0.997, e.getDeltaY());
            viewSize *= mult;
        });
        scene.setOnMousePressed(e -> {
            lastDragX = e.getX();
            lastDragY = e.getY();
        });
        scene.setOnMouseDragged(e -> {

            double dx = e.getX() - lastDragX;
            double dy = e.getY() - lastDragY;

            lastDragX = e.getX();
            lastDragY = e.getY();

            viewX -= dx  / width * viewSize;
            viewY -= dy  / width * viewSize;

        });

        // keyboard control
        scene.setOnKeyPressed(e -> {
            if(e.getCode() == KeyCode.SPACE) isPlaying = !isPlaying;
            if(!isPlaying && e.getCode() == KeyCode.LEFT) {
                if(e.isShiftDown())
                    currentFrame -= 100;
                else
                    currentFrame--;
            }
            if(!isPlaying && e.getCode() == KeyCode.RIGHT) {
                if(e.isShiftDown())
                    currentFrame += 100;
                else
                    currentFrame++;
            }
        });

        stage.setScene(scene);
        stage.show();
    }

    private void drawFrame(int frame) {
```

```
//          viewX = animationData.boids[0].x[frame];
//          viewY = animationData.boids[0].y[frame];

        // clear
//          ctx.setFill(Color.gray(0.0, 0.05));
        ctx.setFill(Color.gray(0.1));
        ctx.fillRect(0, 0, width, height);

        // draw box
        ctx.setStroke(Color.gray(0.4));
        ctx.strokeRect(
                worldToScreenX(0),
                worldToScreenY(0),
                1 * width / viewSize,
                1 * width / viewSize);

        // draw boids
        for(AnimatedBoid b : animationData.boids) {
            ctx.setFill(new Color(b.red, b.green, b.blue, 1));

            ctx.fillPolygon(
                    new double[] {
                            worldToScreenX(b.x[frame] +
Math.cos(b.angle[frame])*b.r),
                            worldToScreenX(b.x[frame] +
Math.cos(b.angle[frame]+2.5)*b.r),
                            worldToScreenX(b.x[frame] +
Math.cos(b.angle[frame]+Math.PI)*b.r*0.4),
                            worldToScreenX(b.x[frame] +
Math.cos(b.angle[frame]-2.5)*b.r)},
                    new double[] {
                            worldToScreenY(b.y[frame] +
Math.sin(b.angle[frame])*b.r),
                            worldToScreenY(b.y[frame] +
Math.sin(b.angle[frame]+2.5)*b.r),
                            worldToScreenY(b.y[frame] +
Math.sin(b.angle[frame]+Math.PI)*b.r*0.4),
                            worldToScreenY(b.y[frame] +
Math.sin(b.angle[frame]-2.5)*b.r)},
                    4
            );
        }

        // draw debug
        if(animationData.debug) {

            for(Line l : animationData.lines[currentFrame]) {
                ctx.strokeLine(worldToScreenX(l.x1), worldToScreenY(l.y1),
                        worldToScreenX(l.x2), worldToScreenY(l.y2));
            }

            ctx.setFill(Color.WHITE);
            ctx.setFont(new Font(20));
            ctx.fillText(animationData.data[currentFrame], width-200, 50);
        }


        // draw frame data
        ctx.setFill(Color.WHITE);
```

```
        ctx.setFont(new Font(40));
        ctx.fillText("Frame: " + frame, 20, 50);
        ctx.setFont(new Font(20));
        ctx.fillText(isPlaying?"playing":"paused", 20, 80);
        ctx.fillText("debug: " + animationData.debug, 20, 110);

    }

    private double worldToScreenX(double worldX) {
        return (worldX-viewX)*width/viewSize + width/2;
    }
    private double worldToScreenY(double worldY) {
        return (worldY-viewY)*width/viewSize + height/2;
    }
}
```