UNIVERSITY OF WATERLOO

Software Engineering

A Comparative Analysis of Data Structures for Use in Two Dimensional Collision Detection

Prepared By

Ajay Sreenivasan

Student ID: 20391951

User ID: asreeniv

3B Software Engineering

September 4, 2013

Ajay Sreenivasan
1274 York Mills Road
Apartment 701
Toronto, ON
M3A 1Z1

September 4, 2013

Dr. Andrew Morton, Director
Software Engineering
University of Waterloo
Waterloo, ON
N2L 3G1

Dear Sir:

This report entitled *A Comparative Analysis of Data Structures for Use in Two Dimensional Collision Detection* is my second work term report. This report is written in fulfillment of the course WKRPT 300 and is based on a personal project that is currently in development, which I began during my 3A work term.

The project involved the development of a two-dimensional scrolling shooter (Starfight) in which the player's primary objective is to successfully pilot a starship through a barrage of incoming obstacles such as meteors and comets, as well as eliminate any enemy ships while avoiding their fire and eventually reach the end of the level. As with any game of this genre, it is required that there be some method of collision detection implemented without which there would be no way of determining if the player had performed an action which involves any sort interaction with other elements in the game.

One of the major decisions that had to be made when creating this game was the manner in which collision detection would function. Before any checks can be performed for whether or not a collision has occurred, the game's data has to be retrieved for analysis. Even with the most efficient collision detection algorithm, if the data is fetched slowly, then there will always be a bottleneck in terms of the game's overall performance. This report examines the efficiency of using various data structures to for data storage and retrieval for use in the collision detection algorithm.

I hereby confirm that I have received no further help other than what is mentioned above in writing this report. I also confirm that this report has not been previously submitted for academic credit at this or any other academic institution.

Sincerely,

S. Ajay

Ajay Sreenivasan
Student ID: 20391951

Executive Summary

Starfight is a two dimensional scrolling shooter (currently in development) in which the player's main objective is to navigate through the level while avoiding incoming obstacles such as meteors and comets, as well as to eliminate any enemy ships while dodging their attacks. Implementing this game properly requires some form of collision detection, the algorithm of which needs to check the area around the player's starship in order to determine whether the player has collided with an in game element.

The problem explored within this report is the most efficient structure for storage and retrieval of data so that the game's collision detection algorithm will not be bottlenecked by slow data retrieval.

In order to develop a solution, four data structures are chosen for testing. The chosen alternative are: a linked list, a two-dimensional array, a range tree, and a quad tree. This report assesses the chosen alternatives using a set of three criteria: efficiency, reusability and flexibility, and ease of implementation.

After analysing and evaluating all four structures against the chosen criteria, it is concluded that the range tree is the best choice. This is because the range tree possess the fastest lookup time in comparison to the other alternatives. It also has the added functionality of being able to efficiently retrieve data within a certain range of values. Therefore, I recommend the use of the range tree as the best solution out of the suggested alternatives.

Table of Contents

List of Figures

List of Tables

1. Introduction

When developing a scrolling shooter, it is required that there be some method of collision detection implemented. This is so that the game can recognize when the player performs an action that interacts with the game's environment. Whether this be running into a wall or eliminating an enemy, all these actions require that the game knows which in game elements the player collided with in order to provide the appropriate reaction (e.g. running into a wall causes the player to stop moving as there is nowhere further to go in that direction). Specifically in this game, it is required that the game understands when the player pilots his starship into an obstacle – for example, a meteor – so that the game can react accordingly and in this case display the death animation and subsequently lower the player's remaining lives.

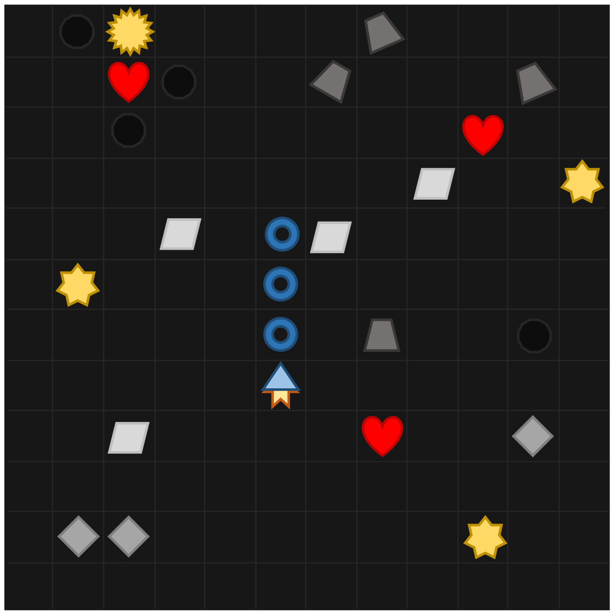
Figure 1-1: A frame during Starfight's gameplay

Starfight's environment is designed in such a way that every element in the game is placed into an individual cell within an invisible grid. Generally, each cell contains exactly one game element which is either harmful or beneficial to the player. Harmful elements include enemy craft and their projectiles, obstacles such as meteors which can be destroyed by firing at them, or black holes which cause the player to immediately lose a life. Beneficial elements include stars which give the player bonus points, as well as hearts which restore player health. It is important to be mindful of the fact that this is only a preliminary version of the game. Therefore, this simplistic method of object positioning will suffice until later iterations are produced.

Given the game environment, it is possible to implement a simple collision detection method in which the game checks for the simultaneous existence of two objects in the same cell. The problem now is to find a computationally efficient method of data storage and retrieval.

This report provides an analysis and evaluation of various methods of storing and retrieving data in order to use the fetched data for collision detection. To begin, some background information on the requirements of the ideal structure as well as the design constraints established by examining Starfight's needs are presented. Next, a set of evaluation criteria is defined and the possible alternatives are provided along with brief overviews. This report then proceeds to evaluate each alternative against each criterion. Finally, a conclusion is drawn from the performed analysis and a recommendation is made. In order to fully understand this report, some knowledge of programming, as well as some technical knowledge of data structures, and runtime analysis will be required.

2. Background

2.1 Overview

Presently, Starfight's game screen is divided into a grid (as seen in Figure 2-1) in which each game element occupies exactly one cell in the grid. In future, this may change in order to allow for larger sprites. The reasoning behind using this grid system to position the game's elements was to simplify the check for collision detection. Every time the game updates the screen, it simply has to check whether the player's starship (whose position is known) is occupying the same cell as another object in the game. Depending on whether the object in question is helpful or harmful to the player, the game can then react accordingly. The issue that now requires addressing is the manner in which the objects in the game's environment are stored. Different methods of storage and retrieval have both their advantages and limitations. Some techniques are very space efficient at the cost of run-time efficiency while some are the opposite, and others possess their own unique functionalities. It is important to find and implement a solution that has a good trade-off between build-time, run-time and space efficiency, as well as functionality.

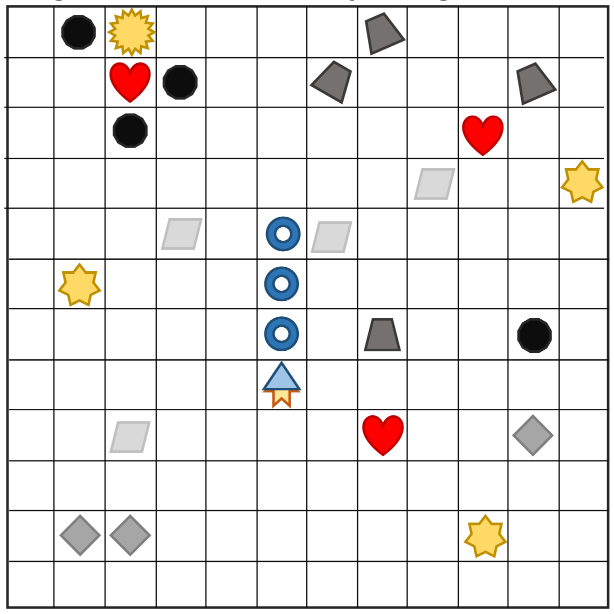
Figure 2-1: Starfight's invisible positioning grid revealed.

2.2 Design Constraints

The design constraints for the ideal data structure were quite general. The primary concern when developing a solution was that it should be efficient in terms of run-time. Because the program will have to perform collision detection checks every time the game screen updates, it is important that the retrieval of data related to the game objects' positioning be as efficient as possible. As the screen will refresh itself at least 30 times per second, a slow computation will quickly compound into a lag prone frame rate. To help restrict the possible designs, the following constraints were established:

- The ideal data structure needs to correctly find all elements within a specified area (one cell) or ideally even from a specified range of cells within a reasonable amount of time.

  - Given that the game must refresh its screen at around 30 frames per second in order for gameplay to appear smooth; this puts the data retrieval time limit in the area of milliseconds.
- The implemented data structure should be reusable in future for other sections within this project or even other projects entirely.
  - Programmer time is a valuable quantity. If a code fragment can be reused multiple times, then it is something worth investing time in, as it will save time and effort in the end.
- The chosen solution should be implementable within a relatively short period.
  - As there is only one coder (myself) working on this project, it is important to not commit to a solution which requires a large amount of work as then the project's completion will inevitably be significantly delayed.
  - In addition, the fastest implementation is usually the simplest implementation. A simple implementation will take less time to implement and is usually easiest to debug if in case something does go wrong.

2.3 Evaluation Criteria

In order to have a set of standards by which to analyze the alternatives to find the one which best suited the design constraints, the following evaluation criteria were established.

2.3.1 Efficiency

This is the primary factor in the final decision of which data structure will be selected for use in Starfight. A run-time inefficient structure, which has a slow build-time or run-time, will negatively affect game performance. A space inefficient structure, which occupies a lot of space in memory, will result in potential game crashes due to stack overflows. There is usually a trade-off between run-time efficiency and space efficiency, as it is not possible to find a structure which does both exceedingly well.

2.3.1.1 Run-time Efficiency

The run-time efficiency is determined by the speed (indirectly measured by observing the execution time) with which the structure can be constructed as well as the speed with which the data can be accessed from within the structure. In this case, it involves constructing and populating the structure with a set of coordinates of game objects and then searching for a given point within the structure. The program should be able to find the point if it exists or realize it does not exist as quickly as possible. There are two possible distributions of data within the game. It can be either dispersed evenly (objects on the map are scattered uniformly), or it can be clustered in certain areas (objects on the map are positioned close to each other). Depending on the nature of the data's distribution, the run-time of certain data structures will be affected.

2.3.1.2 Space Efficiency

All the suggested solutions will require memory allocated to it. Space efficiency is a measure of the amount of space in memory a data structure will occupy. This is related to the possibility of differing data distributions. Certain distributions of data significantly affect the space some data structures require for construction.

2.3.2 Reusability and Flexibility

Any of the possible implementations will take a measurable amount of time to develop and incorporate into the project. If the structure which is developed is modular enough, it is possible that it can be reused for other areas on this project, or even other completely irrelevant projects the future. Modularity also ensures that if in future a more efficient method of solving the problem is developed, the existing solution can be swapped out and replaced with the better method with minimal effort. It is also important that the chosen structure have a certain amount of functionality. For example, the structure should be able to locate the ten nearest objects in relation to the player within a reasonable amount of time.

2.3.3 Ease of Implementation

As this is a project with only myself as the sole programmer, if it is to be completed within any reasonable amount of time, a method that is relatively quick and straightforward to implement should be used. Implementation difficulty can be evaluated by the complexity of creating the structure in question and integrating it into Starfight. Certain structures are prepackaged into the standard library, while some have to be created from the ground up. However, as there is no urgent deadline, this criterion is of lowest priority of the three present. It is more important that an efficient and flexible solution be found.

2.4 Alternatives

A preliminary search into possible solutions generated a rather large list of possibilities for a potential solution. However, the choices were subsequently narrowed down into the following four: a linked list, a two dimensional array, a range tree, and a quad tree.

2.4.1 A Linked List

This solution involves storing every element currently present in the game in a linked list. As elements enter the game screen, they are added to the end of the linked list. Similarly, as objects move off screen or are destroyed, they are removed from within the list. [1] This implementation will also be referred to as the "naïve method" because of the suggested alternatives, it is the most simplistic and straightforward solution. If this method is integrated into Starfight, new elements will be entered into the list as they appear on the game screen and deleted from the list as they are destroyed or leave the visible area.

2.4.2 A Two-Dimensional (2D) Array

This data structure presents a mapping of the entire game grid to a two-dimensional array in memory. Each element in the array would represent its corresponding cell on the game screen. If the cell were empty, it would contain a null value. If the cell were currently occupied, it would contain a pointer to the game object currently in that space.

2.4.3 A Range Tree

A range tree is a data structure in which all the objects in the trees possess values which indicate their location in an n-dimensional coordinate system. For example, in Figure 2-2, each node has a sub tree in which all of the points have the same x-value. In the diagram, the circular nodes represent the possible x-values in the tree while the grey triangles represent the sub trees with all objects that possess the same x-value. However, in this sub tree, the objects are sorted according to their y-value. This structure can be extended to n-dimensions of complexity. [2] However, for the purposes of this

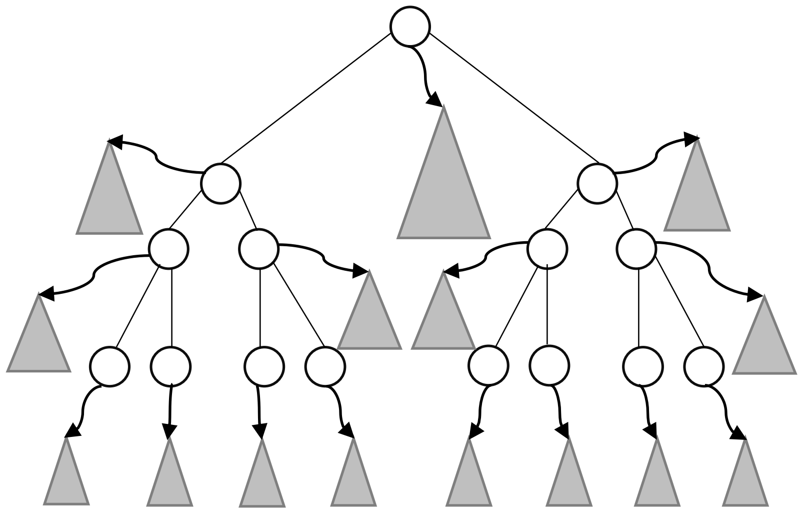
Figure 2-2: A sample two-dimensional range tree.

game, only a two dimensional range tree is required. In Starfight, these objects in the tree would be the in game elements which possess knowledge of which cell they are currently located in. When inserting into this range tree, the program would first locate the node with the correct x-coordinate, and then insert the object into the corresponding y-coordinate sub tree.

2.4.4 A Quad Tree

A quad tree is a tree style data structure in which every node has exactly four children, and

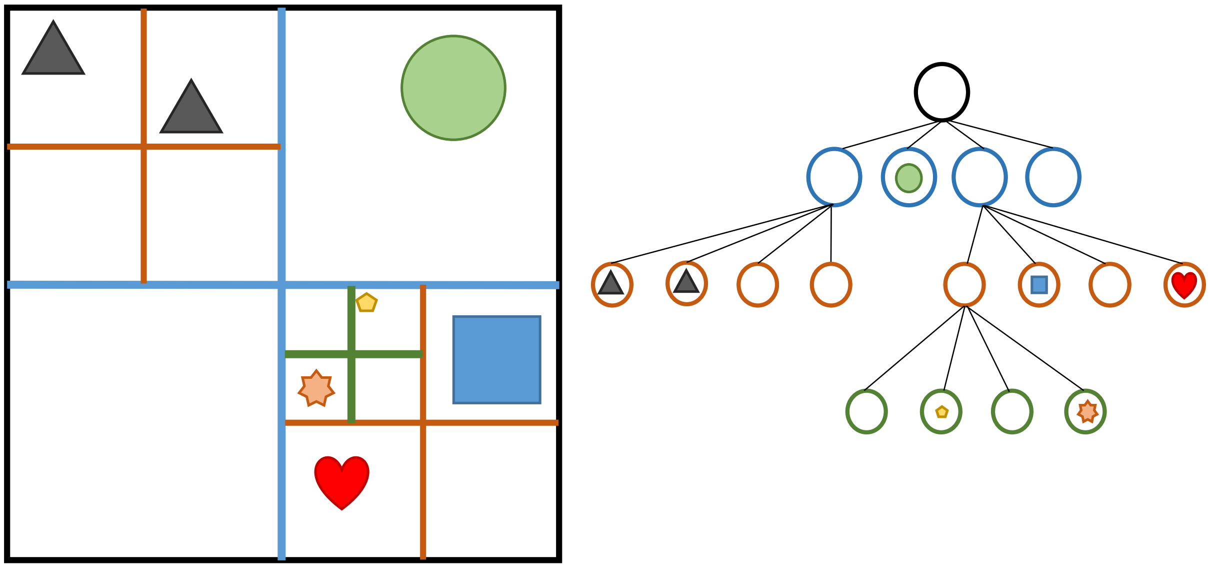

Figure 2-4: A balanced quad tree.

each node contains a certain number (usually one) of objects. Quad trees are able to partition an area into quadrants by subdividing recursively every time a quadrant contains more than the given number of objects. [3] A balanced quad tree is illustrated in Figure 2-3. In this tree, the objects are

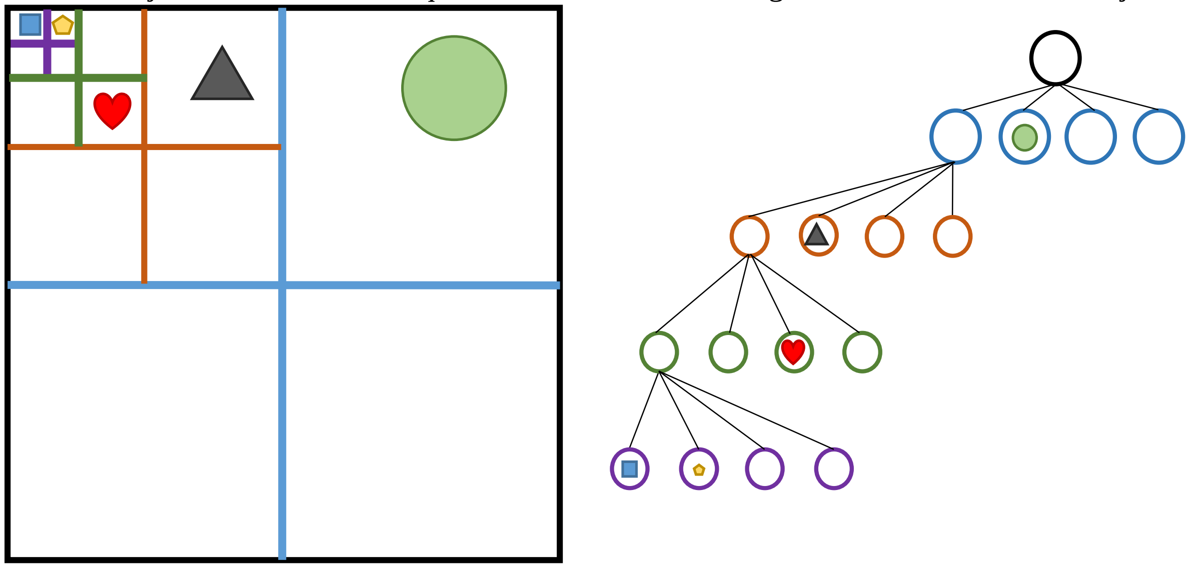

Figure 2-3: A skewed quad tree.

more or less evenly distributed across the given area; therefore, the resulting quad tree is relatively balanced. However, a quad tree can become quite skewed and unbalanced if the points it attempts to represent are clustered together as seen in Figure 2-4. If a quad tree is used for Starfight, precautions will have to be taken in order to ensure that objects on screen are not concentrated on a small area.

2.4.5 Alternatives Not Considered

Cursory research into possible solutions also turned up various other data structures such as the kd-tree, the AVL tree, and the binary heap. However, of the possible structures found, the four chosen were the simplest in terms of implementation that provided at least the basic required functionality. From previous experience, attempting to implement the aforementioned structures would be an extremely tedious and error prone endeavour.

3. Evaluation

3.1 Efficiency

In order to analyze efficiency, the four suggested data structures were first analyzed theoretically, after which then they were implemented and test data was obtained by executing the program with input of different sizes and distribution. The following graphs were obtained:

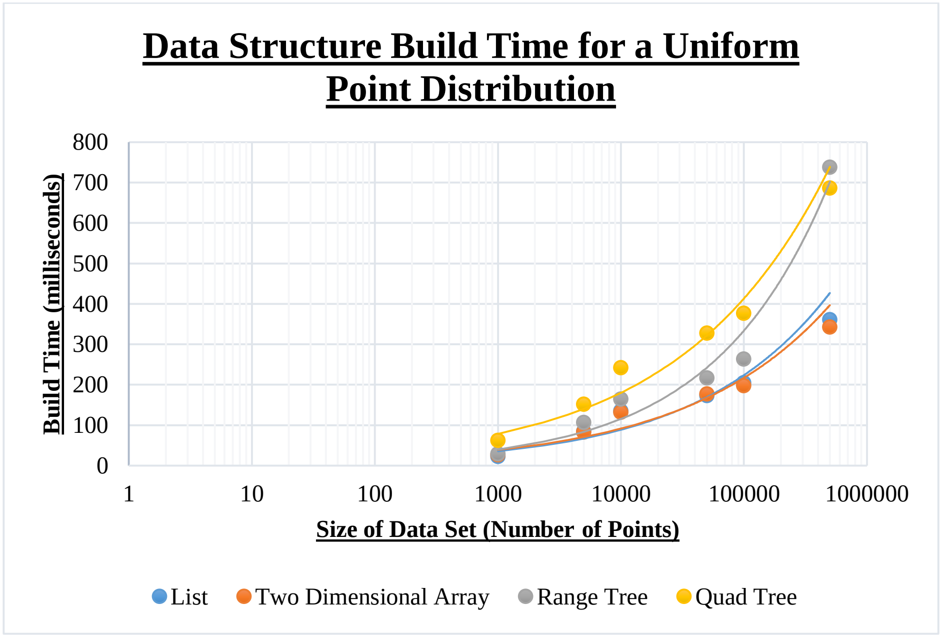

Figure 3-1: Graph of Structural Build-Times for Uniform Point Distribution

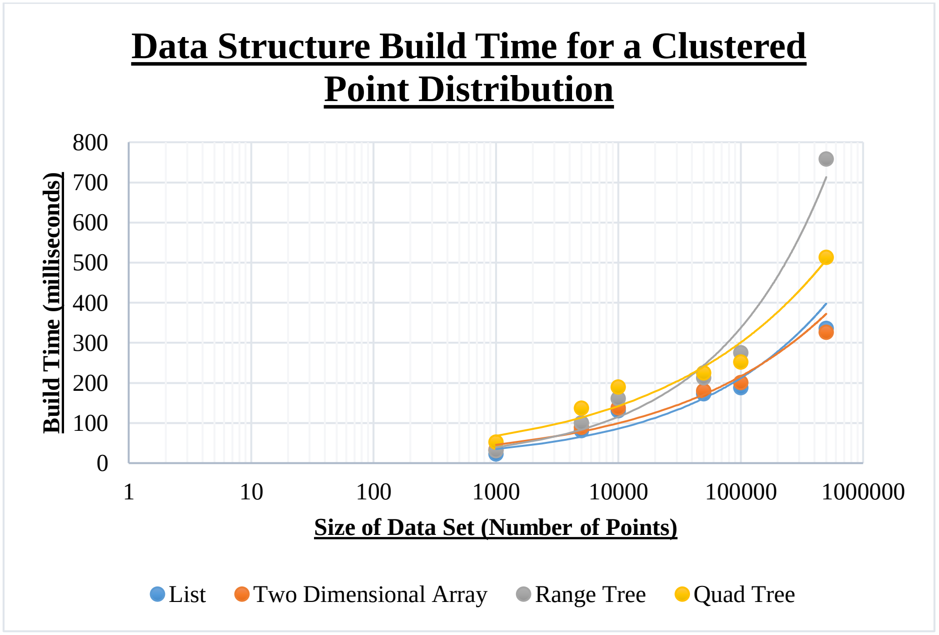

Figure 3-2: Graph of Structural Build-Times for Clustered Point Distribution

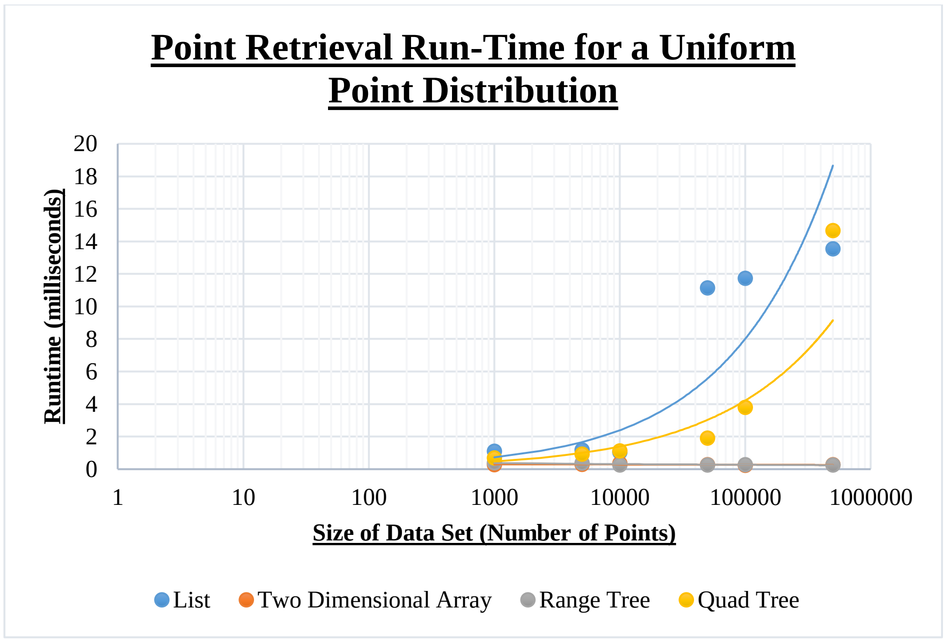

Figure 3-3: Graph of Point Retrieval Run-Times for Uniform Point Distribution

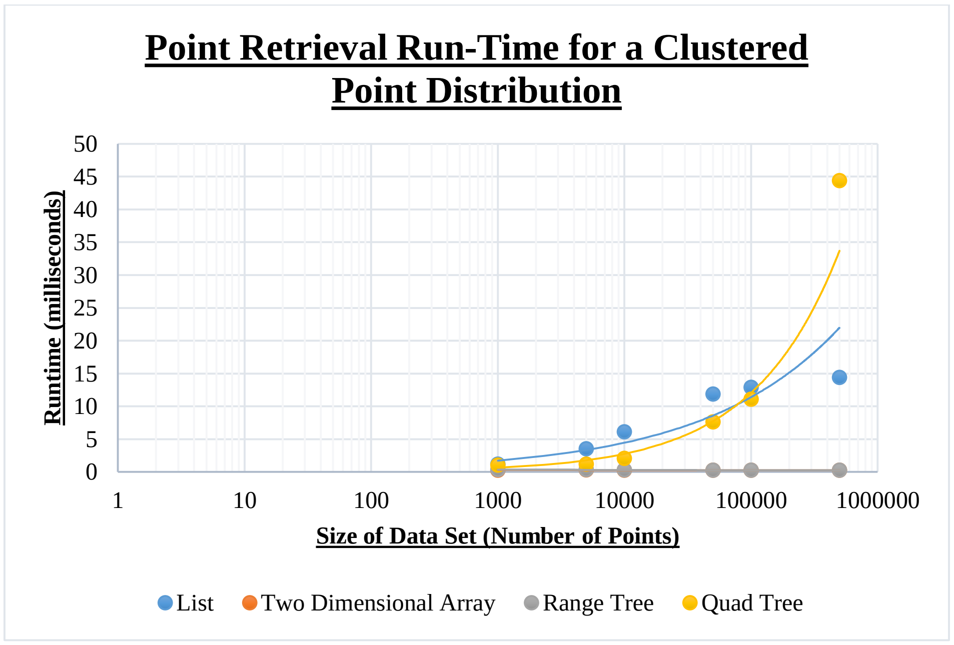

Figure 3-4: Graph of Point Retrieval Run-Times for Clustered Point Distribution

3.1.1 Run-time Efficiency

In terms of build-time efficiency, the linked list and two-dimensional array have the fastest construction for both possible distributions of data. Unfortunately, both the range tree and quad tree appear to perform increasingly poorly as the size of the data set increases. This is as expected because both the range tree and quad tree are comparatively complex structures when compared with the linked list and range tree. Constructing a linked list involves nothing more than adding the newest element to the end of the list. However, constructing the range tree involves traversing two trees until the appropriate location for insertion can be found. In particular, the difference in construction time for the two distributions of data is clearly visible in the range tree's build-time. The build-time for concentrated data points is significantly higher as the sub trees within the range tree grow increasingly unbalanced and take progressively longer to traverse.

When considering the run-time for retrieval of a single point, it is seen that both the range tree and two-dimensional array have near instantaneous response times for both possible point distributions. In fact, the run time actually appears to decrease as the size of the data set increases. This is because of the functionality of a range tree in that it only checks a small subset of all the values present in the whole structure. Similarly, the array only has to check the value for one index and has the advantage of not having to traverse any tree structure. It is also observed that for a clustered distribution of points, the quad tree's performance degrades significantly as the cluster's density increases. The linked list's performance appears to be quite sporadic throughout the tests. This is actually caused because of the way in which the structures were tested. The points for the input data were randomly generated and a single point was selected for testing. It just coincidentally happens to be that in some data sets, the point that was being searched for was located near the start of the list which resulted in an unexpectedly low run time. However, this is unreliable as there is no way to reliably predict where a point will be found within an unsorted list.

In this criterion, it is clear that the best overall performance is from the two dimensional array with a very fast build-time and retrieval run-time.

3.1.2 Space Efficiency

In terms of space requirements, a linked list structure will require as many blocks of memory as there are elements to be represented (assuming that one object occupies one block of memory). Therefore, in an instance where there are k elements on the grid, k blocks of memory will be occupied. The two-dimensional array attempts to represent n^2 elements as it is trying to provide a one to one mapping with the game grid. Each cell will contain space for one object and on a grid that has dimensions with each side is n cells long; the array will have to represent at most n^2 objects in memory. The range tree's performance is similar to that of the linked list in that it only occupies the space that it needs. If only k objects are in existence, then only k blocks of memory will be used. The quad tree has an extremely high space usage, especially when the tree is severely unbalanced. If for example two elements are located right next to each other, then the quad tree will have to subdivide until it can successfully place one element per quadrant. This will result in a massive amount of wasted space in memory.

Of the alternatives, the range tree and the list are most efficient in terms of memory usage.

3.2 Reusability and Flexibility

There is not much value in evaluating the reusability of the linked list as it is found in the standard Java library. Similarly, the array is also a data structure which is found in every programming language. However, arrays are tightly coupled in with the code once written. In order to change the implementation, a lot more time would need to be spent in order to switch in a new structure. Range trees and quad trees are also modular as they both can be easily switched out for another structure. Both structures also have the added benefit of being able to efficiently retrieve nearby points. For example, in Starfight, the range tree can quickly determine all objects present within a set distance of the player's ship.

The quad tree and range tree best satisfy this criterion.

3.3 Ease of Implementation

As a prototype version of each data structure was implemented for the purpose of this report, there is a general idea of the time required to develop and integrate a fully optimized version of these structures would require. The easiest and fastest structure to implement is without a doubt the linked list. In addition to this, the list structure is a part of the Java standard library. There are many functions (such as sorting, and equality comparison) which have been heavily optimized and are already packaged in with the library which means that the programmers of Starfight are not required to create their own basic functions. The two-dimensional array is also a very simple structure as it is just essentially a table of objects. Every programming language is capable of creating arrays. The range tree while slightly tricky at first to implement, can be extended very easily to higher dimensions once the base file is created. The most difficult implementation of the alternatives is the quad tree. It is very easy to make a mistake in the recursive calls while building the structure and cause stack overflows or out of memory exceptions which can prove quite difficult to troubleshoot and debug.

In terms of ease of implementation, either the list or two dimensional array implementations are preferable over the range tree and quad tree.

4. Conclusion

From the analysis, it can be seen that in terms of overall run-time efficiency, there is only one clear choice; the two-dimensional array far outperforms the other alternatives in both build and retrieval run-times. However, it is important to notice that the list also has one of the slowest growing build-times, as well as notice that the range tree is extremely quick when retrieving a given point. In terms of space efficiency, the linked list and range tree are superior as they occupy only the minimum amount of space required. When it comes to reusability and flexibility, the range and quad trees are both clearly the best. They both have the intrinsic ability to retrieve a range of data efficiently. In addition, they are both modular and can be replaced without much effort when an improved solution is developed. Finally, in terms of ease of implementation, the linked list is the clear winner as it is prebuilt into the standard library and comes with an assortment of default functions.

All the data structures have their strengths and weaknesses. However, when taking into consideration the context of the problem, it becomes clear that the range tree has the overall benefits. While it does have one of the highest build-times, it is important to keep in mind that the time is measured on the order of milliseconds. This high build-time is also mitigated in part by the speed at which it can retrieve data once construction is complete. In addition, the range tree also has extra functionality in being able to efficiently retrieve data from within a given range that the linked list and two-dimensional array simply do not possess.

In conclusion, the range tree’s speedy data retrieval time along with its ability to retrieve data from a range of values and its relative ease of implementation make it the most appropriate choice for use in Starfight.

5. Recommendations

Based on the conclusion, there seem to be two choices for a data structure that stand out over the rest in terms of build and run-time efficiency. The two dimensional array is quite clearly the fastest in both build and retrieval time for large amounts of data. However, it is also important to observe that the range tree, while it has a much higher build-time, has functionality that the array simply does not. It is possible to quickly obtain all game elements within a given area using the range tree. If the array is used, a similar operation requires the traversal of every cell in the area for whether or not an element exists. While this operation is acceptable for small ranges, when the area to be searched increases in size, it is more efficient to simply look for what exists than to check every single cell.

I recommend the range tree structure as the best structure for use in conjunction with Starfight's collision detection algorithm. While it does have a larger build-time than some other structures (namely the linked list and the two dimensional array), it does provide additional functionality that those two simply do not. In addition, if the results are examined from a different perspective, it can be argued that the cost of building the range tree is quickly amortized by the extremely fast lookup speed. In Starfight, it is likely that for every time the tree is constructed, several searches will be performed on the tree before it is required to be modified to display the proceeding frame. Compared to the other three alternatives, the range tree is the superior choice as its only flaw is the larger build time. However, it is important to mind the time scale. All other solutions have a lower build time, but they (save for the two dimensional array) however do have a much larger run time when searching for a point. While the range tree has a higher build time than the array, it can also execute a range search for a large interval much faster than the array.

Time permitting, I would also recommend that some more research be done into finding and developing a more optimized version of the range tree in order to improve the build-time. In addition, I suggest that further tests be conducted in order to find which structure can most efficiently perform a range search for all values between a given lower and upper bound.

Glossary

Array – a collection of items which can be selected by navigating to the appropriate index

Build-Time – the time to construct and populate the data structure in question

Run-Time – the time during the program’s execution

Range Search – a search performed on a data structure that returns a range of values from specified bounds

Stack Overflow – an unwanted situation that occurs when the computer attempts to use memory space, which is unavailable for use

References

[1] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. "Elementary Data Structures." *Introduction to Algorithms*. 3rd ed. N.p.: McGraw-Hill, 2001. 204-08. Print.

[2] De Berg, Mark, Dr., Marc Van Kreveld, Dr., Mark Overmars, Dr., and Otfried Cheong Schwarzkopf, Dr. "Orthogonal Range Searching" *Computational Geometry: Algorithms and Applications*. N.p.: Springer, 1997. 94-112. Print.

[3] De Berg, Mark, Dr., Marc Van Kreveld, Dr., Mark Overmars, Dr., and Otfried Cheong Schwarzkopf, Dr. "Quadtrees." *Computational Geometry: Algorithms and Applications*. N.p.: Springer Berlin Heidelberg, 2008. 309-14. Print.

[4] Lambert, Steven. "Quick Tip: Use Quadtrees to Detect Likely Collisions in 2D Space."*Gamedevtuts RSS*. N.p., 3 Sept. 2012. Web. 5 Sept. 2013. <http://gamedev.tutsplus.com/tutorials/implementation/quick-tip-use-quadtrees-to-detect-likely-collisions-in-2d-space/>

Appendix A: Testing Results

These results are obtained by executing the testing code for each structure's implementation using the same data set. These data sets were generated by the code seen in appendix B.1: Test Generation. The time measurements were obtained by recording the system's time before and after execution. There is of course some variation in the times obtained as these times were measured on a scale of nanoseconds. In addition, there was no accounting performed for what other processes and threads that may be using computation time which could potentially skew the actual time measurements. This is why the average of five trials was taken as the approximate value.

A.1: Linked List (Naïve Method) Test Results

The values for these are not quite accurate as the point for which the test harness was searching was placed randomly in the list. As the test iterates sequentially through the list, it is entirely possible that the first item retrieved from the list is the one that is being searched for, but also equally likely that the item in question is last in the list. This is why the run-time values are so randomly dispersed.

Table A.1-1: Test results of Linked List – Build Time on uniformly distributed data.

Linked List Implementation (Uniform Distribution) – Build Time							
Test Size	**Trial 1 (ns)**	**Trial 2 (ns)**	**Trial 3 (ns)**	**Trial 4 (ns)**	**Trial 5 (ns)**	**Average (ns)**	**Average (ms)**
1000	20135705	21425282	19950077	29507230	23704577	22944574.2	22.9445742
5000	97482245	76264455	76245268	92810762	80757004	84711946.8	84.7119468
10000	116492581	157668926	138991470	132613193	131921998	135537633.6	135.5376336
50000	175670803	164978912	199271857	163119957	164788821	173566070	173.56607
100000	229546587	194105515	199176812	201150892	199756899	204747341	204.747341
500000	354199650	353819469	358511926	361950948	377521370	361200672.6	361.2006726

Table A.1-2: Test results of Linked List – Run Time on uniformly distributed data.

Linked List Implementation (Uniform Distribution) – Run Time							
Test Size	**Trial 1 (ns)**	**Trial 2 (ns)**	**Trial 3 (ns)**	**Trial 4 (ns)**	**Trial 5 (ns)**	**Average (ns)**	**Average (ms)**
1000	1039694	1038355	1174453	1058881	1127599	1087796.4	1.0877964
5000	1148572	692981	1045941	730909	2250290	1173738.6	1.1737386
10000	1130723	1056204	1032555	1144110	838448	1040408	1.040408
50000	10898045	10686537	10612018	12661510	10809248	11133471.6	11.1334716
100000	12606624	11424586	13015363	10574090	11028788	11729890.2	11.7298902
500000	13133611	13376801	13647211	14177767	13350474	13537172.8	13.5371728

Table A.1-3: Test results of List – Build Time on unevenly distributed data.

List Implementation (Clustered Distribution) – Build Time							
Test Size	**Trial 1 (ns)**	**Trial 2 (ns)**	**Trial 3 (ns)**	**Trial 4 (ns)**	**Trial 5 (ns)**	**Average (ns)**	**Average (ms)**
1000	22449358	26146296	20555152	24698310	21469904	23063804	23.063804
5000	67132551	82829699	81533428	78887340	95853093	81247222.2	81.2472222
10000	111634132	149975190	131147805	132250415	128659232	130733354.8	130.7333548
50000	162742008	171026986	175764509	192343390	166076168	173590612.2	173.5906122

100000	190841858	180415021	188701338	194985462	189704888	188929713.4	188.9297134
500000	329121606	345376976	335013056	320459126	350208207	336035794.2	336.0357942

Table A.1-4: Test results of List – Run Time on unevenly distributed data.

List Implementation (Clustered Distribution) – Run Time							
Test Size	**Trial 1 (ns)**	**Trial 2 (ns)**	**Trial 3 (ns)**	**Trial 4 (ns)**	**Trial 5 (ns)**	**Average (ns)**	**Average (ms)**
1000	1116444	1267267	1217290	1183823	1124922	1181949.2	1.1819492
5000	3380568	2913821	3954853	3725495	3627774	3520502.2	3.5205022
10000	6274754	5762493	6328301	6250211	6064584	6136068.6	6.1360686
50000	12111320	11894456	10713310	12290701	12217074	11845372.2	11.8453722
100000	13336641	13234903	13119332	11441096	13273278	12881050	12.88105
500000	13971167	14577135	14574457	14578919	14242915	14388918.6	14.3889186

A.2: Two-Dimensional Array Test Results

Table A.2-1: Test results of Two-Dimensional Array – Build Time on uniformly distributed data.

Two-Dimensional Array Implementation (Uniform Distribution) – Build Time							
Test Size	**Trial 1 (ns)**	**Trial 2 (ns)**	**Trial 3 (ns)**	**Trial 4 (ns)**	**Trial 5 (ns)**	**Average (ns)**	**Average (ms)**
1000	25352023	26729507	27881648	32047117	23700561	27142171.2	27.1421712
5000	76281858	83626649	82471384	85101854	88203533	83137055.6	83.1370556
10000	118625515	126383507	137050411	138126248	140712097	132179555.6	132.1795556
50000	164229707	171271068	186407763	188611647	176423578	177388752.6	177.3887526
100000	176211623	200149127	232677270	186113704	193990836	197828512	197.828512
500000	332483878	355867622	336628375	335222780	353628487	342766228.4	342.7662284

Table A.2-2: Test results of Two-Dimensional Array – Run Time on uniformly distributed data

Two-Dimensional Array Implementation (Uniform Distribution) – Run Time							
Test Size	**Trial 1 (ns)**	**Trial 2 (ns)**	**Trial 3 (ns)**	**Trial 4 (ns)**	**Trial 5 (ns)**	**Average (ns)**	**Average (ms)**
1000	358315	228018	229357	236436	257462	261917.6	0.2619176
5000	240959	348052	332434	270856	246342	287728.6	0.2877286
10000	314586	266839	340020	325295	468085	342965	0.342965
50000	209278	213294	337342	257470	298521	263181	0.263181
100000	234266	257023	235158	232035	233819	238460.2	0.2384602
500000	245421	335558	250776	236497	234712	260592.8	0.2605928

Table A.2-3: Test results of Two-Dimensional Array – Build Time on unevenly distributed data

Two-Dimensional Array Implementation (Clustered Distribution) – Build Time							
Test Size	**Trial 1 (ns)**	**Trial 2 (ns)**	**Trial 3 (ns)**	**Trial 4 (ns)**	**Trial 5 (ns)**	**Average (ns)**	**Average (ms)**
1000	31596435	31706205	37151881	34570941	34967631	33998618.6	33.9986186
5000	86635961	89448935	92572926	90122728	90854084	89926926.8	89.9269268

10000	124527676	141920462	140316299	141985164	140754041	137900728.4	137.9007284
50000	166280983	183435042	190365741	183313224	178081287	180295255.4	180.2952554
100000	219296899	193941753	198918450	204625612	188622357	201081014.2	201.0810142
500000	312837232	340571626	330783332	327982404	321051260	326645170.8	326.6451708

Table A.2-4: Test results of Two-Dimensional Array – Run Time on unevenly distributed data

Two-Dimensional Array Implementation (Clustered Distribution) – Run Time							
Test Size	**Trial 1 (ns)**	**Trial 2 (ns)**	**Trial 3 (ns)**	**Trial 4 (ns)**	**Trial 5 (ns)**	**Average (ns)**	**Average (ms)**
1000	321279	224895	239128	278681	333326	279461.8	0.2794618
5000	332434	322618	333327	340466	311908	328150.6	0.3281506
10000	297629	265502	340913	263373	242744	282032.2	0.2820322
50000	256131	213293	328864	208831	207047	242833.2	0.2428332
100000	230696	244529	231589	260146	280249	249441.8	0.2494418
500000	260593	236497	263270	244975	233820	247831	0.247831

A.3: Range Tree Test Results

Table A.3-1: Test results of Range Tree Array – Build Time on uniformly distributed data

Range Tree Implementation (Uniform Distribution) – Build Time							
Test Size	**Trial 1 (ns)**	**Trial 2 (ns)**	**Trial 3 (ns)**	**Trial 4 (ns)**	**Trial 5 (ns)**	**Average (ns)**	**Average (ms)**
1000	28511711	28986490	28609880	30937366	29846804	29378450.2	29.3784502
5000	100533948	119870917	97650470	101022113	115173106	106850110.8	106.8501108
10000	172597164	166874903	158378862	164098964	161395760	164669130.6	164.6691306
50000	227717974	210022650	210301984	220030040	217460702	217106670	217.10667
100000	256180870	270594419	273688513	261230926	256117238	263562393.2	263.5623932
500000	755876725	750381519	745435609	728653699	709234625	737916435.4	737.9164354

Table A.3-2: Test results of Range Tree Array – Run Time on uniformly distributed data

Range Tree Implementation (Uniform Distribution) – Run Time							
Test Size	**Trial 1 (ns)**	**Trial 2 (ns)**	**Trial 3 (ns)**	**Trial 4 (ns)**	**Trial 5 (ns)**	**Average (ns)**	**Average (ms)**
1000	333327	408738	388212	372148	383304	377145.8	0.3771458
5000	382857	393566	354299	370363	346906	369598.2	0.3695982
10000	244975	279917	253900	255239	253454	257497	0.257497
50000	250776	248991	249884	248545	262378	252114.8	0.2521148
100000	290043	252115	244975	253454	259700	260057.4	0.2600574
500000	251669	250776	270856	266840	249437	257915.6	0.2579156

Table A.3-3: Test results of Range Tree Array – Build Time on unevenly distributed data

Range Tree Implementation (Clustered Distribution) – Build Time							
Test Size	**Trial 1 (ns)**	**Trial 2 (ns)**	**Trial 3 (ns)**	**Trial 4 (ns)**	**Trial 5 (ns)**	**Average (ns)**	**Average (ms)**
1000	35766366	29700889	28390786	30106058	30550047	30902829.2	30.9028292
5000	93987445	101051117	105064872	103813223	99447847	100672900.8	100.6729008
10000	159835327	159835327	156620754	175461526	155435591	161437705	161.437705
50000	190824009	205085219	216157737	219728394	235417511	213442574	213.442574
100000	262444200	267275431	270435565	319121356	258770019	275609314.2	275.6093142
500000	839637241	799672385	721707382	716477677	716890878	758877112.6	758.8771126

Table A.3-4: Test results of Range Tree Array – Run Time on unevenly distributed data

Range Tree Implementation (Clustered Distribution) – Run Time							
Test Size	**Trial 1 (ns)**	**Trial 2 (ns)**	**Trial 3 (ns)**	**Trial 4 (ns)**	**Trial 5 (ns)**	**Average (ns)**	**Average (ms)**
1000	368578	374825	362777	365008	354745	365186.6	0.3651866
5000	343144	357869	355638	357869	352068	353317.6	0.3533176
10000	254792	254792	362777	344482	362777	315924	0.315924
50000	248545	259254	283796	278441	249437	263894.6	0.2638946
100000	258362	247206	276657	290043	243637	263181	0.263181
500000	249883	251222	243636	247206	257469	249883.2	0.2498832

A.4: Quad Tree Test Results

Table A.4-1: Test results of Quad Tree Array – Build Time on uniformly distributed data

Quad Tree Implementation (Uniform Distribution) – Build Time							
Test Size	**Trial 1 (ns)**	**Trial 2 (ns)**	**Trial 3 (ns)**	**Trial 4 (ns)**	**Trial 5 (ns)**	**Average (ns)**	**Average (ms)**
1000	62482932	55648617	81056419	54251947	59109504	62509883.8	62.5098838
5000	153174592	147294297	156853235	150172420	152297768	151958462.4	151.9584624
10000	240926550	241774815	233345709	245451228	250113786	242322417.6	242.3224176
50000	309896637	323982037	320858047	364418548	318691199	327569293.6	327.5692936
100000	356347309	382954105	385706840	375957813	384332035	377059620.4	377.0596204
500000	718447296	694002438	673885475	600791863	745996955	686624805.4	686.6248054

Table A.4-2: Test results of Quad Tree Array – Build Time on uniformly distributed data

Quad Tree Implementation (Uniform Distribution) – Run Time							
Test Size	**Trial 1 (ns)**	**Trial 2 (ns)**	**Trial 3 (ns)**	**Trial 4 (ns)**	**Trial 5 (ns)**	**Average (ns)**	**Average (ms)**
1000	655729	695657	670223	728678	668885	683834.4	0.6838344
5000	923677	925908	918768	949111	902258	923944.4	0.9239444
10000	1172222	1077176	1119121	1075837	1107519	1110375	1.110375
50000	1878589	1974973	1884836	1891529	1864756	1898936.6	1.8989366
100000	4519323	3054826	4675053	3346208	3414034	3801888.8	3.8018888

500000	13343334	15652973	16607885	12920764	14828357	14670662.6	14.6706626

Table A.4-3: Test results of Quad Tree Array – Build Time on unevenly distributed data

Quad Tree Implementation (Clustered Distribution) – Build Time							
Test Size	**Trial 1 (ns)**	**Trial 2 (ns)**	**Trial 3 (ns)**	**Trial 4 (ns)**	**Trial 5 (ns)**	**Average (ns)**	**Average (ms)**
1000	55370622	50496108	54543329	52004780	50136008	52510169.4	52.5101694
5000	119800414	138591656	137778643	154512361	137875026	137711620	137.71162
10000	180402526	190031967	187905726	187118593	206049502	190301662.8	190.3016628
50000	227092820	225519891	217536113	229904901	227123162	225435377.4	225.4353774
100000	252948626	250389997	240099703	261064486	259424625	252785487.4	252.7854874
500000	490972511	519021938	498328012	543636805	517152274	513822308	513.822308

Table A.4-4: Test results of Quad Tree Array – Run Time on unevenly distributed data

Quad Tree Implementation (Clustered Distribution) – Run Time							
Test Size	**Trial 1 (ns)**	**Trial 2 (ns)**	**Trial 3 (ns)**	**Trial 4 (ns)**	**Trial 5 (ns)**	**Average (ns)**	**Average (ms)**
1000	1234246	970084	913859	938848	1140094	1039426.2	1.0394262
5000	1335538	959821	1069145	1410057	1207026	1196317.4	1.1963174
10000	1487700	1775512	1472082	2009777	3711217	2091257.6	2.0912576
50000	15032279	5549646	6310898	5796406	5528673	7643580.4	7.6435804
100000	11226465	11875715	9808375	10586583	11927477	11084923	11.084923
500000	44995100	45829980	44614474	43430651	42966582	44367357.4	44.3673574

Appendix B: Code Fragments

B.1: Test Data Generation

```
public class TestDataGen {
	public static void main(String[] args) throws IOException {
		final int DATA_SET_SIZE = 1000000;
		final int GAME_SCREEN_WIDTH = 2000;
		final int GAME_SCREEN_HEIGHT = 2000;

		BufferedWriter bufWriter = new BufferedWriter(new FileWriter(
				"testdata1000000.txt"));

		ArrayList<Point> pointArrayList = new ArrayList<Point>();
		Random random = new Random();
		random.setSeed(3141592L);

		// Generate points and write them to a file
		while (pointArrayList.size() < DATA_SET_SIZE) {
			Point point = new Point(random.nextInt(GAME_SCREEN_WIDTH),
					random.nextInt(GAME_SCREEN_HEIGHT));
			if (!pointArrayList.contains(point))
				pointArrayList.add(point);
		}

		for (Point point : pointArrayList) {
			bufWriter.write("(" + point.x + "," + point.y + ")");
			bufWriter.newLine();
		}

		bufWriter.close();
	}
}
```

B.2: List Structure Implementation (Naïve Method)

```
public class NaiveMethod {
	public static void main(String[] args) throws IOException {

		long buildStart = 0;
		long buildEnd = 0;
		long lookupStart = 0;
		long lookupEnd = 0;

		buildStart = System.nanoTime();

		Point player = new Point(1550, 939);
		ArrayList<Point> obstacleArray = new ArrayList<Point>();
		String temp;

		String inputLocation = "./testdata.txt";
		BufferedReader bufReader = new BufferedReader(new FileReader(
				inputLocation));
		while ((temp = bufReader.readLine()) != null) {
			temp = temp.substring(1, temp.length() - 1);
			String[] temp2 = temp.split(",");
			Point newObstaclePoint = new Point(Integer.parseInt(temp2[0]),
					Integer.parseInt(temp2[1]));
			obstacleArray.add(newObstaclePoint);
		}
		bufReader.close();
		buildEnd = System.nanoTime();

		lookupStart = System.nanoTime();
		for (Point point : obstacleArray) {
			if (point.equals(player)) {
				System.out.println("Collision detected.");
				break;
			}
		}
		lookupEnd = System.nanoTime();

		System.out.println("Build Time: " + (buildEnd - buildStart));
		System.out.println("Lookup Time: " + (lookupEnd - lookupStart));
	}
}
```

B.3: Two Dimensional Array Implementation

```
public class TwoDArrayTest {
	public static void main(String[] args) throws IOException {
		final int GAME_SCREEN_WIDTH = 2000;
		final int GAME_SCREEN_HEIGHT = 2000;

		long buildStart = 0;
		long buildEnd = 0;
		long lookupStart = 0;
		long lookupEnd = 0;

		buildStart = System.nanoTime();
		Point player = new Point(382, 628);

		boolean[][] obstacleArray = new boolean[2000][2000];
		for (int i = 0; i < GAME_SCREEN_WIDTH; i++)
			for (int j = 0; j < GAME_SCREEN_HEIGHT; j++)
				obstacleArray[i][j] = false;

		String temp;
		String inputLocation = "./testdata.txt";
		BufferedReader bufReader = new BufferedReader(new FileReader(
				inputLocation));
		while ((temp = bufReader.readLine()) != null) {
			temp = temp.substring(1, temp.length() - 1);
			String[] temp2 = temp.split(",");
			Point newObstaclePoint = new Point(Integer.parseInt(temp2[0]),
					Integer.parseInt(temp2[1]));
			obstacleArray[newObstaclePoint.x][newObstaclePoint.y] = true;
		}
		bufReader.close();
		buildEnd = System.nanoTime();

		lookupStart = System.nanoTime();
		if (obstacleArray[player.x][player.y] == true)
			System.out.println("Collision detected.");
		else
			System.out.println("No collision detected.");
		lookupEnd = System.nanoTime();

		System.out.println("Build Time: " + (buildEnd - buildStart));
		System.out.println("Lookup Time: " + (lookupEnd - lookupStart));
	}
}
```

B.4: Range Tree Implementation

B.4.1: Range Tree Testing Methods

```
public class RangeTreeTest {
	public static void main(String[] args) throws IOException {
		long buildStart = 0;
		long buildEnd = 0;
		long lookupStart = 0;
		long lookupEnd = 0;
		long rangelookupStart = 0;
		long rangelookupEnd = 0;

		buildStart = System.nanoTime();
		RangeTree2D rangeTree2D = new RangeTree2D(0);
		Point player = new Point(1550, 939);
		String temp;

		String inputLocation = "./testdata.txt";
		BufferedReader bufReader = new BufferedReader(new FileReader(
				inputLocation));
		while ((temp = bufReader.readLine()) != null) {
			temp = temp.substring(1, temp.length() - 1);
			String[] temp2 = temp.split(",");
			Point newObstaclePoint = new Point(Integer.parseInt(temp2[0]),
					Integer.parseInt(temp2[1]));
			rangeTree2D.add(newObstaclePoint);
		}
		bufReader.close();
		buildEnd = System.nanoTime();

		lookupStart = System.nanoTime();
		System.out.println(rangeTree2D.search(player));
		lookupEnd = System.nanoTime();

		rangelookupStart = System.nanoTime();
		rangeTree2D.rangeSearch(player.x - 500, player.x + 500, player.y - 500,
				player.y + 500);
		rangelookupEnd = System.nanoTime();

		System.out.println("Build Time: " + (buildEnd - buildStart));
		System.out.println("Lookup Time: " + (lookupEnd - lookupStart));
		System.out
				.println("Range Time: " + (rangelookupEnd -
				rangelookupStart));
	}
}
```

B.4.2: One Dimensional Range Tree Node Implementation

```java
public class RangeTree1DNode {
	Point point;
	RangeTree1DNode left;
	RangeTree1DNode right;

	public RangeTree1DNode(Point point, RangeTree1DNode left,
			RangeTree1DNode right) {
		this.point = new Point(point.x, point.y);
		this.left = left;
		this.right = right;
	}

	public boolean add(Point point) {
		if (point.x == this.point.x)
			return false;
		else if (point.x < this.point.x) {
			if (this.left == null) {
				this.left = new RangeTree1DNode(point, null, null);
				return true;
			} else
				return left.add(point);
		} else if (point.x > this.point.x) {
			if (this.right == null) {
				this.right = new RangeTree1DNode(point, null, null);
				return true;
			} else
				return right.add(point);
		}
		return false;
	}

	public boolean search(Point point) {
		if (point.x == this.point.x)
			return true;
		else if (point.x < this.point.x) {
			if (this.left == null)
				return false;
			else
				return this.left.search(point);
		} else if (point.x > this.point.x) {
			if (this.right == null)
				return false;
			else
				return this.right.search(point);
		}
		return false;
	}

	public void rangeSearch(int xLo, int xHi) {
		if (this.point.x > xHi) {
			if (this.left != null)
				this.left.rangeSearch(xLo, xHi);
		} else if (this.point.x < xLo) {
				if (this.right != null)
					this.right.rangeSearch(xLo, xHi);
			} else {
				if (this.left != null)
					this.left.rangeSearch(xLo, xHi);

				System.out.println(this.point.x + " " + this.point.y);

				if (this.right != null)
					this.right.rangeSearch(xLo, xHi);
			}

		}
	}
```

B.4.3: One Dimensional Range Tree Implementation

```
public class RangeTree1D {
	RangeTree1DNode root;
	int yVal;

	public RangeTree1D(int yVal) {
		this.root = null;
		this.yVal = yVal;
	}

	public boolean add(Point point) {
		if (this.root == null) {
			root = new RangeTree1DNode(point, null, null);
			return true;
		} else {
			return this.root.add(point);
		}
	}

	public boolean search(Point point) {
		if (this.root == null)
			return false;
		else
			return this.root.search(point);
	}

	public void rangeSearch(int xLo, int xHi) {
		if (this.root != null)
			this.root.rangeSearch(xLo, xHi);
	}

	public int getyVal() {
		return this.yVal;
	}

	public void setyVal(int yVal) {
		this.yVal = yVal;
	}
}
```

B.4.4: Two Dimensional Range Tree Node Implementation

```
public class RangeTree2DNode {
	RangeTree1D subRange1DTree;
	RangeTree2DNode left;
	RangeTree2DNode right;

	public RangeTree2DNode(Point point, RangeTree2DNode left,
			RangeTree2DNode right) {
		this.subRange1DTree = new RangeTree1D(point.y);
		this.subRange1DTree.add(point);
		this.subRange1DTree.setyVal(point.y);

		this.left = left;
		this.right = right;
	}

	public boolean add(Point point) {
		if (point.y == this.subRange1DTree.yVal)
			return this.subRange1DTree.add(point);
		else if (point.y < this.subRange1DTree.yVal) {
			if (this.left == null) {
				this.left = new RangeTree2DNode(point, null, null);
				return true;
			} else
				return left.add(point);
		} else if (point.y > this.subRange1DTree.yVal) {
			if (this.right == null) {
				this.right = new RangeTree2DNode(point, null, null);
				return true;
			} else
				return this.right.add(point);
		}
		return false;
	}

	public boolean search(Point point) {
		if (point.y == this.subRange1DTree.yVal)
			return this.subRange1DTree.search(point);
		else if (point.y < this.subRange1DTree.yVal) {
			if (this.left == null)
				return false;
			else
				return this.left.search(point);
		} else if (point.y > this.subRange1DTree.yVal) {
			if (this.right == null)
				return false;
			else
				return this.right.search(point);
		}
		return false;
	}

	public void rangeSearch(int xLo, int xHi, int yLo, int yHi) {
		if (this.subRange1DTree.getyVal() > yHi) {
```

```
            if (this.left != null)
                this.left.rangeSearch(xLo, xHi, yLo, yHi);
        } else if (this.subRange1DTree.getyVal() < yLo) {
            if (this.right != null)
                this.right.rangeSearch(xLo, xHi, yLo, yHi);
        } else {
            if (this.left != null)
                this.left.rangeSearch(xLo, xHi, yLo, yHi);
            this.subRange1DTree.rangeSearch(xLo, xHi);
            if (this.right != null)
                this.right.rangeSearch(xLo, xHi, yLo, yHi);
        }
    }
}
```

B.4.5: Two Dimensional Range Tree Implementation

```java
public class RangeTree2D {
	RangeTree2DNode root;
	int zVal;

	public RangeTree2D(int zVal) {
		this.root = null;
		this.zVal = 0;
	}

	public boolean add(Point point) {
		if (this.root == null) {
			root = new RangeTree2DNode(point, null, null);
			return true;
		} else
			return this.root.add(point);
	}

	public boolean search(Point point) {
		if (this.root == null)
			return false;
		else
			return root.search(point);
	}

	public void rangeSearch(int xLo, int xHi, int yLo, int yHi) {
		if (this.root != null)
			this.root.rangeSearch(xLo, xHi, yLo, yHi);
	}

	public void setzVal(int zVal) {
		this.zVal = zVal;
	}

	public int getzVal() {
		return this.zVal;
	}
}
```

B.5: Quad Tree Implementation

B.5.1: Quad Tree Testing Implementation

```java
public class QuadTreeTest {
	public static void main(String[] args) throws IOException {
		long buildStart = 0;
		long buildEnd = 0;
		long lookupStart = 0;
		long lookupEnd = 0;

		buildStart = System.nanoTime();
		Point player = new Point(549, 350);
		String temp;
		QuadTree gameGrid = new QuadTree(0, new Rectangle(0, 0, 2000, 2000));

		String inputLocation = "./testdata.txt";
		BufferedReader bufReader = new BufferedReader(new FileReader(
				inputLocation));
		while ((temp = bufReader.readLine()) != null) {
			temp = temp.substring(1, temp.length() - 1);
			String[] temp2 = temp.split(",");
			Point newObstaclePoint = new Point(Integer.parseInt(temp2[0]),
					Integer.parseInt(temp2[1]));
			gameGrid.insert(newObstaclePoint);
		}
		bufReader.close();
		buildEnd = System.nanoTime();

		lookupStart = System.nanoTime();
		ArrayList<Point> returnObjects = new ArrayList<Point>();
		gameGrid.retrieve(returnObjects, player);
		System.out.println(returnObjects);
		lookupEnd = System.nanoTime();

		System.out.println("Build Time: " + (buildEnd - buildStart));
		System.out.println("Lookup Time: " + (lookupEnd - lookupStart));
	}
}
```

B.5.2: Quad Tree Implementation [4]

```java
public class QuadTree {
	ArrayList<Point> pointArrayList;
	int depth;
	QuadTree[] subQuadrant;
	Rectangle bounds;

	final int MAX_OBJECTS = 1;
	final int MAX_LEVELS = 5;

	public QuadTree(int depth, Rectangle bounds) {
		this.bounds = bounds;
		this.subQuadrant = new QuadTree[4];
		this.depth = depth;
		this.pointArrayList = new ArrayList<Point>();
	}

	// Determine to which node the object belongs.
	// Index of -1 means point belongs to the parent node.
	public int getIndex(Point point) {
		int index = -1;

		if (point.getX() < (bounds.getX() + (bounds.getWidth() / 2))) {
			if (point.getY() < (bounds.getY() + (bounds.getHeight() / 2))) {
				index = 1;
			} else if (point.getY() > (bounds.getY() + (bounds
					.getHeight() / 2))) {
				index = 2;
			}
		} else if (point.getX() > (bounds.getX() + (bounds.getWidth() / 2))) {
			if (point.getY() < (bounds.getY() + (bounds.getHeight() / 2))) {
				index = 0;
			} else if (point.getY() > (bounds.getY() + (bounds
					.getHeight() / 2))) {
				index = 3;
			}
		}

		return index;
	}

	public void splitQuadrant() {
		subQuadrant[0] = new QuadTree(depth + 1,
				new Rectangle((int) bounds.getX()
						+ ((int) (bounds.getWidth() / 2)),
						((int) bounds.getY()),
						((int) (bounds.getWidth() / 2)),
						((int) (bounds.getHeight() / 2))));
		subQuadrant[1] = new QuadTree(depth + 1, new Rectangle(
				(int) bounds.getX(), ((int) bounds.getY()),
				((int) (bounds.getWidth() / 2)),
				((int) (bounds.getHeight() / 2))));
		subQuadrant[2] = new QuadTree(depth + 1, new Rectangle(
                    (int) bounds.getX(), ((int) bounds.getY())
                                 + ((int) (bounds.getHeight() / 2)),
                    ((int) (bounds.getWidth() / 2)),
                    ((int) (bounds.getHeight() / 2))));
            subQuadrant[3] = new QuadTree(depth + 1,
                    new Rectangle((int) bounds.getX()
                                 + ((int) (bounds.getWidth() / 2)),
                                 ((int) bounds.getY())
                                        + ((int) (bounds.getHeight() / 2)),
                                 ((int) (bounds.getWidth() / 2)),
                                 ((int) (bounds.getHeight() / 2))));
    }

    public void insertPoint(Point point) {
            if (subQuadrant[0] != null) {
                    int index = getIndex(point);

                    if (index != -1) {
                            subQuadrant[index].insertPoint(point);
                            return;
                    }
            }

            pointArrayList.add(point);

            if (pointArrayList.size() > MAX_OBJECTS && depth < MAX_LEVELS) {
                    if (subQuadrant[0] == null) {
                            splitQuadrant();
                    }
                    int i = 0;
                    while (i < pointArrayList.size()) {
                            int index = getIndex(pointArrayList.get(i));
                            if (index != -1) {
                                    subQuadrant[index].insertPoint(pointArrayList
                                            .remove(i));
                            } else {
                                    i++;
                            }
                    }
            }
    }

    public ArrayList<Point> retrieve(
                    ArrayList<Point> retrievedPoints, Point point) {
            int index = getIndex(point);
            if (index != -1 && subQuadrant[0] != null) {
                    subQuadrant[index].retrieve(retrievedPoints, point);
            }

            retrievedPoints.addAll(retrievedPoints);
            return retrievedPoints;
    }
}
```