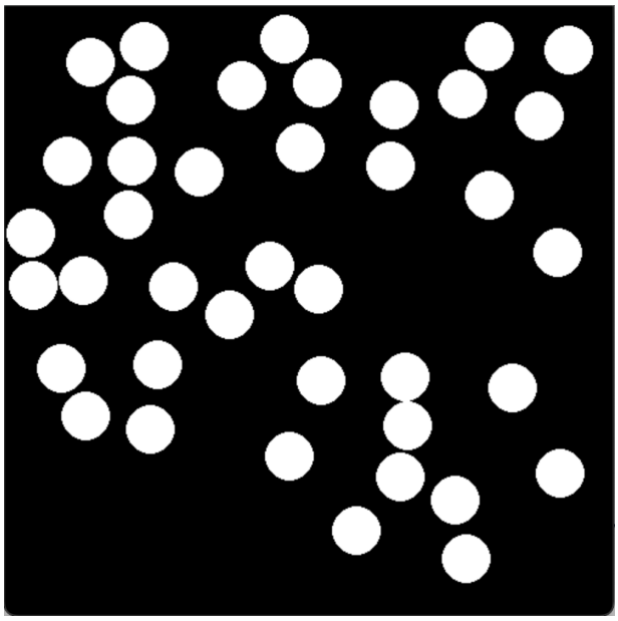
Performance of broad-phase collision detection algorithms

 [1]

How does the performance of spatial partitioning and sweep-and-prune approaches to broad-phase collision detection compare across varying object numbers, sizes, and distributions?

Computer Science Extended Essay

Personal Code: jtk047

Word Count: 3914

[1] Screenshot from self-made simulation

Table of Contents

1. Introduction 3

2. Background Information 5

2.1 Brute force approach 5
2.2 Sweep and Prune 6
2.3 Spatial Partitioning 8

3. Methodology 10

3.1 Experimental Procedure 10
3.2 Independent Variables 12
3.3 Dependent Variable 13
3.4 Software and Hardware used 14
3.5 Experiments (code in Appendix A.5) 14

4. Experimental Results and Analysis (raw data in Appendix B) 16

4.1 Experiment 1: Impact of grid size on performance 16
4.2 Experiment 2: Impact of quadtree capacity on performance 17
4.3 Experiment 3: Impact of number of particles on the optimum value of p 18
4.4 Experiment 4: Performance with a uniform distribution 19
4.5 Experiments 5 and 6: Performance in a non-uniform distribution 20

5. Limitations and further research opportunities 21

5.1 Sizes of particles 21
5.2 Using Sweep and Prune in multiple dimensions 22
5.3 Using temporal coherence 22
5.4 Using a better implementation for the simulation 22
5.5 Analyzing each stage of the algorithm 23

6. Conclusion 23

7. Bibliography 25

8. Appendix 27

Appendix A – Simulation code 27
Appendix B – Raw data 38

1. Introduction

Collision detection is the computational problem of finding all objects in a scene that overlap with each other. This is crucial in eventually handling the collision, whether it be ending the game when the ghost eats Pacman or warning the user about overlapping components in Computer-Aided design (CAD) software. Collision detection is often performed by physics engines, alongside other features such as simulating gravity and movement, performing soft body and fluid dynamics computations, etc.

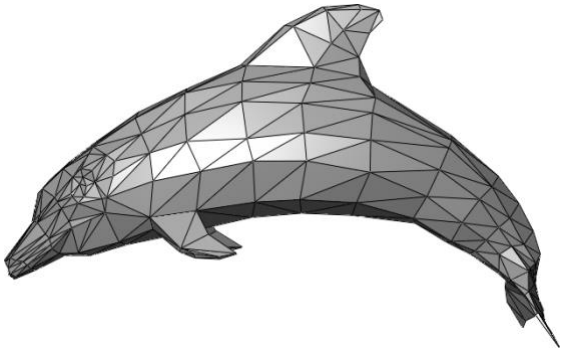


Figure 1: Dolphin triangle mesh[2]

Checking for intersection between two objects requires complicated algorithms, especially if the objects are not simple shapes such as spheres or cubes but are complex objects such as dolphins or laptops. These objects are often represented as polyhedral meshes as shown in Figure 1. Collision checks performed against two objects require complex algorithms such as the GJK (Gilbert–Johnson–Keerthi) algorithm which runs for multiple iterations.[3] In a scene with thousands of objects, performing a lot of expensive pairwise checks can cause a massive performance hit, leading to a fall in frame rate for video games or slowing down running time

[2] Chrschn. File:Dolphin Triangle mesh.png - *Wikimedia Commons*. 27 Mar. 2007, commons.wikimedia.org/wiki/File:Dolphin_triangle_mesh.png. Accessed 7 Jan. 2024.
[3] Ericson, Christer. "Real-Time Collision Detection." *CRC Press eBooks*, 2004, p. 401. https://doi.org/10.1201/b14581.

for simulations. Hence, the need for splitting collision detection into *broad* and *narrow* phases arises.

The broad phase shortlists pairs of objects that could *potentially* intersect each other in some scene. Then, each pair is individually tested for collision in the *narrow* phase, in which expensive polygon intersection algorithms are performed to check whether two objects overlap exactly.

Improving the efficiency of collision detection is an ongoing area of research and constant optimization in the development of physics engines and game engines. For example, NVIDIA's PhysX, used in Unity, Unreal Engine and Autodesk, continues to get updated with new broadphase collision detection algorithms (eMBP in PhysX 3.3, eABP in PhysX 4 and ePAPB in PhysX 5), giving developers more freedom to choose between Sweep and Prune and spatial partitioning depending on the scenario.[4] Similarly, Chipmunk2D (used in 2D mobile games built using Cocos such as FarmVille, Geometry Dash and Shadow Fight 2) supports two different algorithms for broadphase collision detection, as well as an experimental sweep and prune approach, allowing developers to pick the appropriate algorithm and tune its parameters to improve performance.[5]

This investigation will compare the performance of two different approaches to the *broad phase* of collision detection. These include spatial partitioning approaches (which work by dividing a scene into individual cells where objects only in the same cell are checked against

[4] "Rigid Body Collision." Rigid Body Collision - Physx 5.3.1 Documentation, *NVIDIA*, 11 Dec. 2023, nvidia-omniverse.github.io/PhysX/physx/5.3.1/docs/RigidBodyCollision.html#broad-phase-algorithms. Accessed 15 Dec. 2023

[5] Lembcke, Scott. "Chipmunk2D 7.0.3." *Chipmunk Game Dynamics Manual*, chipmunk-physics.net/release/ChipmunkLatest-Docs/#CollisionDetection. Accessed 15 Dec. 2023.

each other) or the sweep and prune approach (which groups objects together depending on intersecting intervals). The performance of these algorithms will be tested under different scenarios of varying number, size, and distribution of objects.

2. Background Information

2.1 Brute force approach

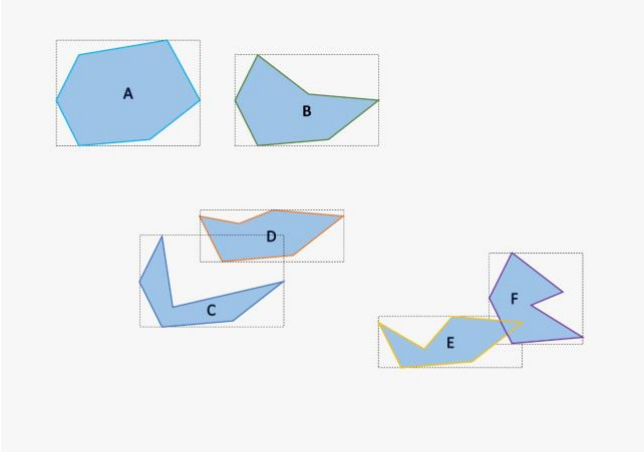


Figure 2: Collision detection in broad and narrow phase[6]

The brute force approach to broad phase collision detection involves passing all potential pairs to the narrow phase and not filtering out any pairs of objects. With this approach, the number of collisions and hence the time complexity of the brute force method was derived mathematically by me as $\binom{n}{2} = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n = O(n^2)$. In a scene with thousands or millions of objects, this brute force approach would get exceedingly slow due to its quadratic time complexity the slow involved in checking for the intersection of complex objects.

To avoid this, a broad phase collision detection algorithm is used to shortlist pairs of potentially intersecting objects. In Figure 2, a broad phase collision detection algorithm might identify pairs DC and EF as potentially colliding, as their AABBs (Axis Aligned Bounding Boxes) intersect. Then, these pairs would be passed to a narrow phase collision detection

[6] Self-made using macOS Preview

algorithm which would perform more complex checks to detect whether the objects intersect. This narrow stage algorithm would then identify E and F as objects that intersect, for which the collision would ultimately get handled.

2.2 Sweep and Prune

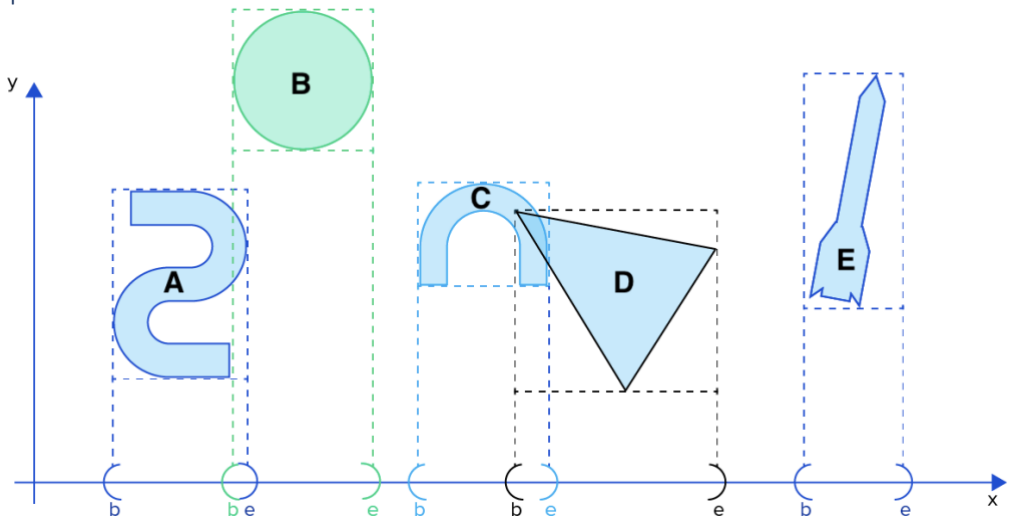


Figure 3: Sweep and Prune along the x axis[7]

Sweep and Prune is a broad phase algorithm that checks for overlapping intervals over an axis to shortlist pairs of potentially intersecting objects.

The first step is to construct a sorted array of the boundaries of the intervals for each object. Let Z_b denote the x-coordinate of the beginning of the interval of object Z and Z_e denote the x-coordinate of the end of the interval of object Z. In Figure 3, this sorted array becomes $\{A_b, B_b, A_e, B_e, C_b, D_b, C_e, D_e, E_b, E_e\}$.

We then loop over the array and maintain a list of "active" objects to check for intersection and perform pairwise checks over this "active" array. We add objects when encountering their

[7] Image edited from Souto, Nilson. "Video Game Physics Tutorial - Part II: Collision Detection for Solid Objects." *Toptal Engineering Blog,* 2 Mar. 2015, www.toptal.com/game/video-game-physics-part-ii-collision-detection-for-solid-objects. Accessed 12 Jun 2023.

begin boundary and remove them when encountering their end boundary. The idea is that it is necessary for the projections of objects along some axis to overlap if the objects themselves intersect.

Iteration	Boundary	Active Objects	Pairwise tests
0	A_b	$\{A\}$	$\emptyset$
1	B_b	$\{A, B\}$	$\{\{A, B\}\}$
2	A_e	$\{B\}$	$\emptyset$
3	B_e	$\emptyset$	$\emptyset$
4	C_b	$\{C\}$	$\emptyset$
5	D_b	$\{C, D\}$	$\{\{C, D\}\}$
6	C_e	$\{D\}$	$\emptyset$
7	D_e	$\emptyset$	$\emptyset$
8	E_b	$\{E\}$	$\emptyset$
9	E_e	$\emptyset$	$\emptyset$

Table 1: Trace Table for Sweep and Prune performed on Figure 2

From Table 1, it is apparent that sweep and prune reduced the number of pairwise checks in this example from $\binom{5}{2} = 10$ in the brute force approach to 2. However, the improvement might be negligible when objects clump up along the axis the algorithm is performed over[8]. If we had performed Sweep and Prune along the y-axis of Figure 2 instead, we would end up performing $\binom{4}{2} = 6$ pair-wise checks between A, C, D and E and then another pairwise check

[8] Reducible. "Building Collision Simulations: An Introduction to Computer Graphics." *YouTube*, 19 Jan. 2021, www.youtube.com/watch?v=eED4bSkYCB8. Accessed 2 March 2023.

between E and B, resulting in only a marginal improvement from 10 to $6 + 1 = 7$ checks. A similar situation would occour with very large objects.[9]

2.3 Spatial Partitioning

Spatial partitioning techniques partition the screen into different regions to reduce pairwise checks to objects only within those regions.

2.3.1 Uniform Grid

A uniform grid divides the screen into regions of rectangles with the same width and height. For objects that are a part of multiple regions, they are counted in all the regions that they intersect. Then, pairwise checks are performed among the objects that are part of each region. in each region the screen into different regions to reduce pairwise checks to objects only within those regions.

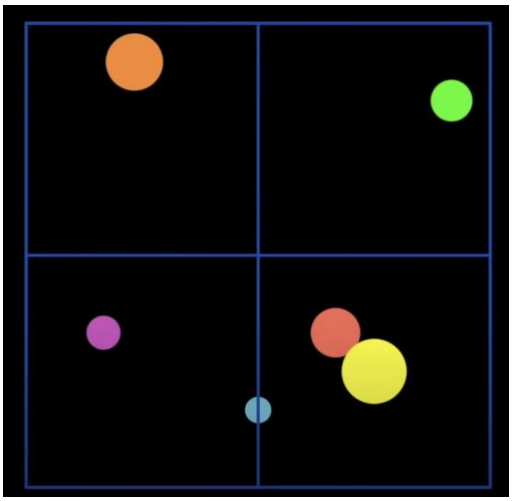


Figure 4: Uniform Partitioning[10]

The uniform grid in Figure 4 would result in pairs from the bottom-left ($\{Purple, Blue\}$) and pairs from the bottom-right $\{Red, Yellow, Blue\}$ being checked against each other. This

[9] Reducible. "Building Collision Simulations: An Introduction to Computer Graphics." *YouTube*, 19 Jan. 2021, www.youtube.com/watch?v=eED4bSkYCB8. Accessed 2 March 2023.
[10] Screenshot IBID.

reduced the collision checks from $\binom{6}{2} = 15$ in the brute-force approach to $0 + 0 + \binom{2}{2} + \binom{3}{2} = 4$.

However, if the number of partitions of the grid is too large or too small, this is likely to make the algorithm inefficient[11]. An extremely fine grid will have the unnecessary overhead of associating each grid with objects and looping over each region. An extremely coarse grid will require too many pairwise tests for all the objects falling inside the grid, rendering the spatial partitioning approach not much better than brute force. Moreover, this algorithm is best suited for situations in which objects are evenly distributed.[12]

2.3.2 Quadtrees

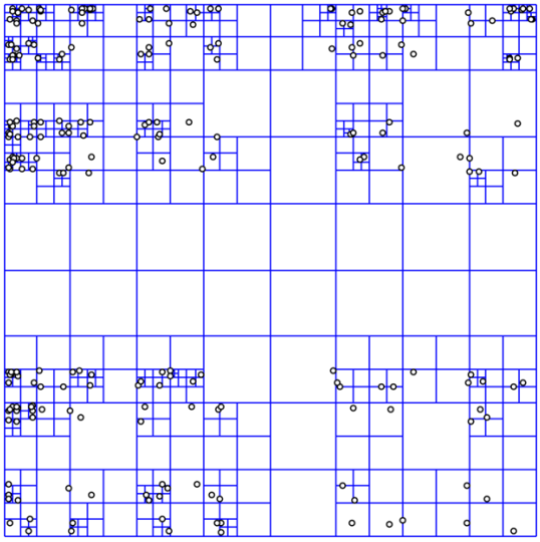


Figure 5: Quadtree based partitioning[13]

[11] Ericson, Christer. "Real-Time Collision Detection." *CRC Press eBooks*, 2004, p. 286. https://doi.org/10.1201/b14581.
[12] Rayner, Darcy. "How can I implement fast, accurate 2D collision detection?" *Game Development Stack Exchange,* 8 Oct. 2011, gamedev.stackexchange.com/a/18271. Accessed 29 May 2023.
[13] Eppstein, David. File:Point quadtree.svg - *Wikimedia Commons*. 31 May 2005, commons.wikimedia.org/w/index.php?curid=2489019. Accessed 12 Jun 2023.

A Quadtree is a tree in which each node has either four children or no children. If the number of objects inside a quadtree exceeds its *capacity,* the region is split into four smaller regions by adding four children to the node corresponding to the region. Figure 5 shows a Quadtree with a capacity of 1 since each cell has one or fewer objects.

Collision detection works slightly differently from the uniform grid. We insert each object into the Quadtree, then query all the nodes of the Quadtree in a region around the object to find potential candidates. The query then recursively keeps searching only the children for which the region of interest intersects with the node's boundary[14].

Because of the ability for this data structure to change depending on the distribution of objects, this makes Quadtrees an appropriate spatial partitioning scheme for scenes with clusters of objects in places and empty space everywhere else[15].

3. Methodology

3.1 Experimental Procedure

[14] The Coding Train. "Coding Challenge #98.1: Quadtree - Part 1." *YouTube,* 26 Mar. 2018, www.youtube.com/watch?v=OJxEcs0w_kE. Accessed 15 Aug. 2023.
[15] Figueiredo, Mauro, et al. 'A Survey on Collision Detection Techniques for Virtual Environments'. *Proc. of V Symposium in Virtual Reality, Brasil*, vol. 307, 01 2002. Accessed 25 Jun. 2023.

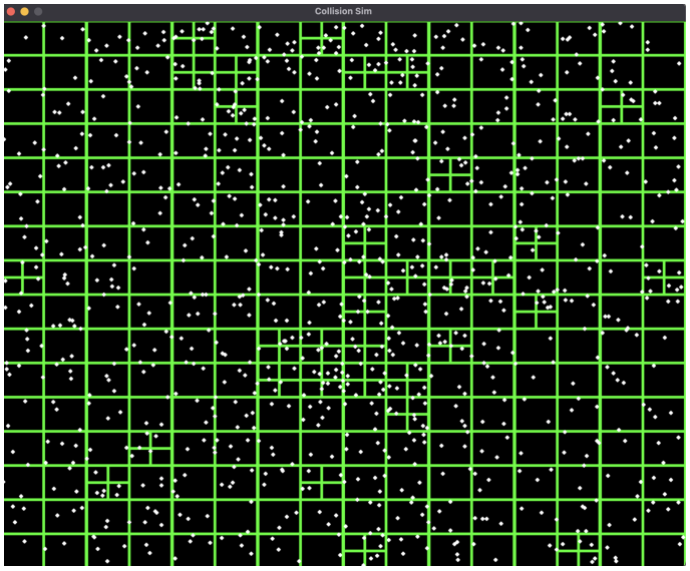


Figure 6: (1000x800) Simulation screenshot with n=1024, r=3 and κ= 4

To collect empirical data for the performance of these algorithms, I wrote a program in Python. The code includes the data structures to support the spatial partitioning algorithms (Appendix A.2), the implementation of the broad-phase algorithms (Appendix A.3), the code for the simulation (Appendix A.4), and the code for the experiments which vary the parameters and graph the results (Appendix A.5).

The 2000x2000 px simulation is initialized with n circular objects of radius r distributed on the screen according to the specified initial distribution. The particles initially move at a velocity of 10 $\frac{pixels}{sec}$ at a random initial angle.

Each frame, the following process occurs. One of the broad phase collision detection algorithms is run to identify pairs of particles that could intersect. These pairs are passed to a simple narrow phase algorithm (since the objects are circular particles, the narrow phase has a very simple implementation). The time taken during this process is measured using the function time.time() to evaluate the performance of the algorithm. Next, if these particles do

indeed collide, conservation of momentum is used to make the particles rebound by changing their velocities. Finally, the position of the particles is updated, and they are re-drawn to the screen. While the live simulation with collision response isn't strictly necessary to evaluate the performance of the broad phase algorithms, it allows me to ensure that the algorithms are working correctly by spotting collisions and data structures visually (as shown in Figure 6).

3.2 Independent Variables

Section 2 provided background information about the algorithms and discussed potential factors that could affect their performance by analyzing their workings or referring to existing literature. Eventually, the following parameters were decided to be varied to evaluate the performance of the algorithms under a variety of circumstances (Table 2).

Letter	**Parameter**	**Values**
n	Number of particles	4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048
r	Size of particles (radius in pixels)	3, 10, 25
-	Distribution of particles **Uniform:** Particles are uniformly distributed **Gaussian X:** Particles are clumped more at the center on the X axis ($X \sim N(1000, 200^2)$), but distributed uniformly along the Y axis **Gaussian Y:** Particles are clumped more at the center on the Y axis ($Y \sim N(1000, 200^2)$), but distributed uniformly along the X axis	-

p	Number of partitions of the uniform grid in each axis (number of grids is p^2)	6-29 (optimum value lied in this range according to initial experimentation)
κ	Capacity of the Quadtree	1-19 (optimum value lied in this range according to initial experimentation)

Table 2: Independent variables of experiment

The experiment was run both with clumping along the X axis and with clumping along the Y axis. This is because the performance of the sweep and prune algorithm depends on the axis where clumping is occurring. The sweep and prune algorithm was performed along the X axis in the experiment.

3.3 Dependent Variable

The time taken to detect collisions in each frame was measured. The time taken by the broad-phase algorithm to shortlist pairs of colliding particles **and** the time taken for the narrow phase algorithm to check each pair for collision was included. While the aim of this essay is to evaluate the performance of the broad-phase algorithms, it is important to consider the performance of the algorithm both in terms of **how many pairs the algorithm shortlisted out of all the possible pairs** and **how much time it took the algorithm to shortlist these pairs.** Measuring the time taken in both broad-phase and narrow-phase considers performance in terms of both aspects, which makes it a holistic measurement of performance. The simulation was run for 3 seconds, and the average of the time taken per frame was calculated to eliminate outliers and random errors. This average value will be referred to as the “frame period”.

3.4 Software and Hardware used

Library	**Purpose**
Pygame	Drawing the scene of the simulation on the screen
Numpy	General data processing utilities (random number generation, transposing arrays, etc.)
Matplotlib	Generating graphs for analysis

Table 3: Libraries used in experiment

Python 3.9 was used as a programming language to develop the simulation. This is because of its ease of use and the availability of PIP libraries for data collection and manipulation allowing data to be processed directly in the simulation. The limitations for this choice are discussed in section 5.4.

The program was run on an Arch Linux machine with minimum background applications or daemons running to reduce any external impact on performance. The machine has an i5-9400F CPU and 16 gigabytes of RAM.

3.5 Experiments (code in Appendix A.5)

3.5.1 Experiment 1: Impact of grid size on performance

The simulation was run with the uniform grid spatial partitioning algorithm. The parameter p was varied, and the frame period was measured. This experiment was conducted to ensure that the best performing uniform grid (where the value of p minimized the frame period) would be used when comparing the uniform grid to the other algorithms.

3.5.2 Experiment 2: Impact of quadtree capacity on performance

The simulation was run with the Quadtree spatial partitioning algorithm, varying κ and measuring the frame period. This experiment was conducted to ensure that the optimal capacity would be used later on.

3.5.3 Experiment 3: Impact of number of particles on optimum value of p

When running **Experiment 1** and varying the number of particles, it was determined that the optimum value of p which minimized the frame period depended on the number of particles.

```
prev_prev = run_sim(n, size,
Particles.UNIFORM_PARTITIONING, Particles.UNIFORM, 1, 0)[1]
prev = run_sim(n, size, Particles.UNIFORM_PARTITIONING,
Particles.UNIFORM, 2, 0)[1]
i = 3
while True:
    cur = run_sim(n, size, Particles.UNIFORM_PARTITIONING,
Particles.UNIFORM, i, 0)[1]
    if cur > prev and prev_prev > prev:
        # local minima
        print(n, size, prev, i-1)
        maxs.append(i-1)
        break
    prev_prev = prev
    prev = cur
    i += 1
```

Figure 7: Code to detect minima

The code in Figure 7 was written to automatically detect the optimum value of p by finding local minima. The value of n was varied and the optimum value of p was recorded.

3.5.4 Experiment 4: Performance in a uniform distribution

The simulation was run with a uniform distribution of particles varying the number and size of particles. The simulation was run thrice for each configuration, each time with a different broad phase algorithm or spatial partitioning scheme. Optimal values for the parameters of

the Quadtree and Uniform Grid determined in **Experiments 1, 2 and 3** were used. The average frame period of the Quadtree, Uniform Grid and Sweep and Prune algorithm were compared against each other.

3.5.5 Experiment 5: Performance with clumping along the X axis

The experiment from Experiment 4 was repeated but with a Gaussian distribution of particles along the X axis.

3.5.6 Experiment 6: Performance with clumping along the Y axis

The experiment from Experiment 4 was repeated but with a Gaussian distribution of particles along the Y axis.

The size of the particles was kept constant at 10px for Experiments 5 and 6 to reduce the volume of data.

4. Experimental Results and Analysis (raw data in Appendix B)

4.1 Experiment 1: Impact of grid size on performance

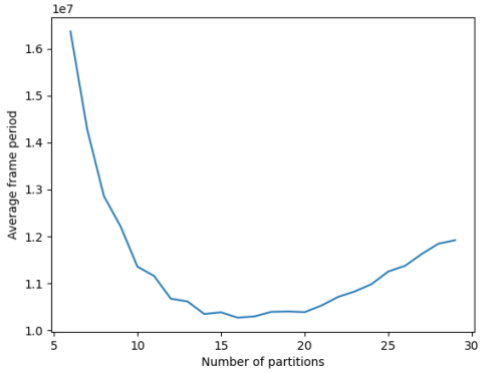


Figure 8: Average frame period against number of partitions for n=1024, r=10

Figure 8 shows the data collected from **Experiment 1** graphically. Initially, as the number of partitions is increased, the performance of the uniform grid drastically improves. This is because the number of pairwise collision checks are reduced since there are fewer particles in each cell. However, eventually, the grid becomes too large, after (~15-20), the performance starts to degrade. This is likely because of the additional overhead in creating the large data structure, assigning particles to each cell, and particles overlapping in multiple cells. The experiment was repeated, alternating the number and size of the particles, and it was determined that the number of particles had a significant impact on the optimum value of p (see Section 4.3), but the size of the particles did not. The optimum value remained at (~15-20) despite changing the size. Existing literature predicts that the size of the particles should have a significant impact on the optimum value[16]. My results likely contradicts this because only a small range of sizes were used in the experiment, while existing literature suggests that the grid size should only be reconsidered if there are very large discrepancies between the grid size and the object size.

4.2 Experiment 2: Impact of quadtree capacity on performance

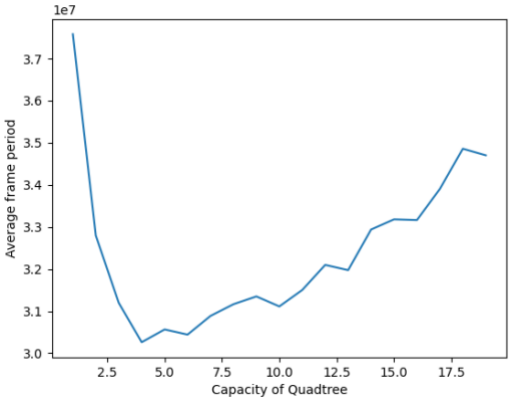


Figure 9: Average frame period against capacity of Quadtree for n=1024, r=10

[16] Kroiss, Ryan Robert. "Collision Detection Using Hierarchical Grid Spatial Partitioning on the GPU." *Faculty of the Graduate School of the University of Colorado*, Jan. 2013, p. 7, scholar.colorado.edu/csci_gradetds/67. Accessed 28 Sep. 2023.

A similar pattern to **Experiment 1** can be observed in the data for **Experiment 2.** Initially, as the capacity is increased from 1 to 3, the performance of the Quadtree improves (Figure 9). This is likely because each parent node can now have more child nodes before the child nodes themselves start having children, which reduces depth of the tree and hence the time taken to insert particles into the Quadtree. However, as the capacity is increased beyond 6, the time saved in **building the data structure** due to the shorter tree is tiny compared to the extra time needed in **more pairwise checks** due to the greater number of particles in each cell. Therefore, the performance continues to degrade as the capacity of the Quadtree is increased. The size and number of particles had no significant impact on the optimum value of capacity, which remained at around 3-5.

4.3 Experiment 3: Impact of number of particles on the optimum value of p

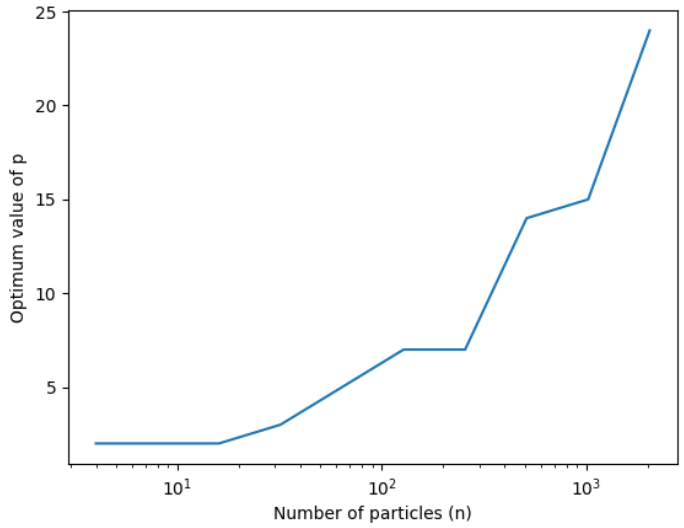


Figure 10: Optimum value of p against number of particles (see Appendix B.3 for raw data)

From Figure 10, we can see that the optimum value of p increases as the number of particles increases. For a larger number of particles, a **larger grid size** is required to maintain the **same number of particles per grid**. This is likely the reason why a larger number of partitions are favored for a larger number of particles, so that the number of particles and hence the number of pairwise checks within each grid does not grow much.

4.4 Experiment 4: Performance with a uniform distribution

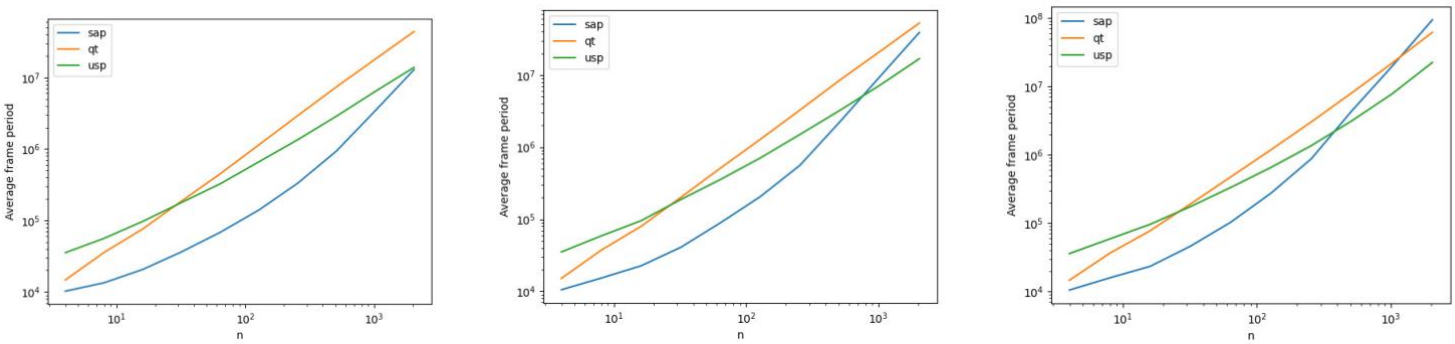


Figure 11.a, 11.b, 11.c: Average frame period against number of particles (3px (left), 10px (middle) and 25px (right))[17]

Both axes in Figures 11 were displayed using logarithmic scales due to the large variation in the independent and dependent variables.

All the algorithms were adversely impacted by an increase in the number of particles, shown by the increasing time as n increases in Figures 11. This is obvious since a greater number of particles increases both the number of pairwise checks and the time spent in the broad phase for both sweep and prune and the spatial partitioning algorithms. Initially, sweep and prune performed better than both spatial partitioning algorithms, indicated by the lower y-value of the blue graph in Figures 11. However, the performance of sweep and prune quickly deteriorated, and it performed worse than the spatial partitioning algorithms for a greater number of particles.

This observation can be explained logically by considering the working of the sweep and prune algorithm. For a small number of particles, there are very few interval overlaps, so most of the time is spent in sorting the intervals in $O(n \log(n))$ time[18]. As the number of

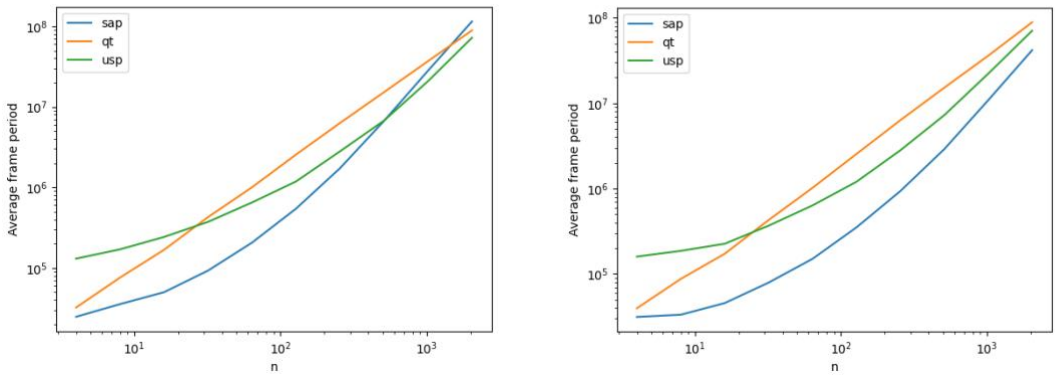
[17] Figure Legend abbreviations: SAP: Sweep and Prune, QT: Quadtree, USP: Uniform grid

[18] Auger, Nicolas, et al. "On the Worst-Case Complexity of TimSort." *HAL (Le Centre Pour La Communication Scientifique Directe)*, Aug. 2018, https://doi.org/10.4230/lipics.esa.2018.4. Accessed 14 Nov 2023.

particles is increased to the point where there are many axial overlaps, the performance approaches $O(n^2)$ since there are more pairwise checks. The spatial partitioning algorithms do not approach this worst-case scenario of pairwise checks as quickly because they work on both axes.

Increasing the size of the particles also worsened the performance of all algorithms, with frame period increasing from 13.9ms to 22.4ms for the uniform grid and 13ms to 94ms for the sweep and prune algorithm when the size was increased from 3px to 25px[19]. This is because axial overlaps and overlaps of particles between grids increase when size is increased. Sweep and prune was clearly hit harder by the increase in size, likely due to the same reasoning as the previous paragraph.

4.5 Experiments 5 and 6: Performance in a non-uniform distribution



Figures 12.a, 12.b: Gaussian distribution across x-axis (left) and y-axis (right)[20]

The performance of all algorithms was worse in the non-uniform distributions compared to the uniform distribution. This is because of clumping of particles in a location increasing the

[19] Refer to appendix B.4 (n=2048)
[20] Log by Log plot, Figure Legend Abbreviations: SAP: Sweep and Prune, QT: Quadtree, USP: Uniform grid.

number of pairwise checks. Both sweep and prune and the uniform grid fared significantly worse, with frame period increasing from 39ms to 115ms and 17ms to 72ms respectively[21]. This is because these algorithms can't adjust themselves to a change in the distribution of particles and are considerably affected by clumping. The Quadtree only fared slightly worse, with frame period increasing from 53ms to 89ms[22]. This is likely because the Quadtree can adjust itself to the scene and only subdivide a cell where its required instead of wasting resources maintaining empty grids where there are no objects.

Comparing Figure 12.a and 12.b, we can see that the green and orange curves have barely changed. This is expected since the spatial partitioning algorithms subdivide into both axes, so the axis that is clumped should not affect their performance. Sweep and Prune, however, performs much worse when particles are clumped along the X-axis, with its frame period exceeding the spatial partitioning algorithms in Figure 12.a but not in Figure 12.b. This is expected by the explanation in Section 2.2 and the fact that Sweep and Prune was performed along the X axis.

5. Limitations and further research opportunities

5.1 Sizes of particles

A scenario in which the grids were too small for the particle size was not encountered, due to the small range of sizes used. Moreover, uniform grids tend to face problems when not all objects are of the same size (which is often the case in most real-world scenarios), wherein

[21] Refer to appendices B.4 and B.5 (n=2048, r=10)
[22] Refer to appendices B.4 and B.5 (n=2048, r=10)

sweep and prune would tend to perform better[23]. This scenario could be explored in a future investigation, and a wider range of sizes could be used.

5.2 Using Sweep and Prune in multiple dimensions

The version of the algorithm of Sweep and Prune used in this essay works only using a single axis. However, the versions prominently used in industry are ones that automatically select the axis with the least clumping by computing the variance of the center of the objects along each axis and sorting along the axis with the most variance[24]. Using this version of sweep and prune would likely make the algorithm more efficient and be a better indicator of its performance in the real world.

5.3 Using temporal coherence

Temporal coherence is a major optimization technique in collision detection algorithms that was not discussed or utilized in this paper. This technique updates the data structures instead of creating them again each frame, with the idea that objects do not move much between frames. A future study could look at how the performance of the algorithms changes when temporal coherence optimizations are introduced, potentially exploring how the sorting algorithm used[25] affects these optimizations as well.

5.4 Using a better implementation for the simulation

While this paper gives insight into the strengths and weaknesses of each algorithm, any direct comparisons made on the time to run each algorithm are probably not that fruitful since the

[23] Ericson, Christer. "Real-Time Collision Detection." *CRC Press eBooks*, 2004, pp. 286-287. https://doi.org/10.1201/b14581

[24] Souto, Nilson. "Video Game Physics Tutorial - Part II: Collision Detection for Solid Objects." *Toptal Engineering Blog,* 2 Mar. 2015, www.toptal.com/game/video-game-physics-part-ii-collision-detection-for-solid-objects. Accessed 12 Jun. 2023.

[25] Cohen, Jonathan D.; Lin, Ming C.; Manocha; Ponamgi, Madhav K. (April 9–12, 1995), *I–COLLIDE: an Interactive and Exact Collision Detection System for Large Scale Environments) (PDF),* Proceedings of the 1995 Symposium on Interactive 3D Graphics (Monterey, CA), p. 192. Accessed 2 Jan. 2024.

simulation was written in Python, which is slow and inefficient as an interpreted language. Hence, the implementation probably caused unnecessary overhead in certain places that would not be present in a video game or simulation. Further research could conduct the experiment using a language like C and utilizing parallelism via the GPU, giving a more real-world result for the performance of these algorithms.

5.5 Analyzing each stage of the algorithm

This investigation only looked at one dependent variable, the frame period. The time spent in each stage (making the data structure, shortlisting pairs and narrow phase) could be recorded in a future investigation. This would give insight into how changing certain parameters affect the performance in each stage (which this investigation had to make educated guesses about), giving deeper insights into the advantages and drawbacks of the algorithms.

6. Conclusion

This experiment shed light on the strengths and weaknesses of both the sweep and prune and the spatial partitioning approaches to broad phase collision detection in terms of their performance when varying object **number**, **size,** and **distribution**.

The simplicity of sweep and prune meant it performed extremely well for a small **number** of objects but did not scale as well as the spatial partitioning approaches did, due to axial overlaps. Sweep and Prune also did not perform well with **large objects** or when objects were **clumped** along the axis it was performed over but performed better with **small objects** and **less clumping**. Additionally, out of the two spatial partitioning approaches investigated, the Quadtree worked better when objects were **not evenly distributed** but the Uniform Grid's simple data structure meant it worked better for **uniform distributions**. Furthermore,

the impact of the **number of particles** on the optimal value for grid size means that the algorithm is not suitable for scenarios in which the **number of objects changes or is unknown**.

These results could be applied when deciding the collision detection system to be used. Uniform grids are a better idea for something like an ideal gas simulation, where particles are **uniformly distributed**, have **similar sizes** and the **number of particles is known**. Quadtrees (or the 3D equivalent – octrees) might be more suited for complex environments in video games. Sweep and prune may be a good idea for a scenario **without too many particles coinciding** along an axis, which may be true for a very **large scene with spread out objects**. Furthermore, sweep and prune may perform well in scenarios with **varying object sizes**, which could be explored as an extension to this investigation.

7. Bibliography

Auger, Nicolas, et al. “On the Worst-Case Complexity of TimSort.” *DROPS-IDN/v2/Document/10.4230/LIPIcs.ESA.2018.4*, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. *drops.dagstuhl.de*, https://doi.org/10.4230/LIPIcs.ESA.2018.4. Accessed 14 Nov. 2023.

Building Collision Simulations: An Introduction to Computer Graphics. Directed by Reducible, 2021. *YouTube*, https://www.youtube.com/watch?v=eED4bSkYCB8. Accessed 2 Mar. 2023.

Chrschn. *An Example of a Polygon Mesh. Illustration of a Dolphin, Represented with Triangles.* 27 Mar. 2007. http://en.wikipedia.org/wiki/File:Dolphin_triangle_mesh.png, *Wikimedia Commons*, https://commons.wikimedia.org/wiki/File:Dolphin_triangle_mesh.png. Accessed 7 Jan. 2024.

Coding Challenge #98.1: Quadtree - Part 1. Directed by The Coding Train, 2018. *YouTube*, https://www.youtube.com/watch?v=OJxEcs0w_kE. Accessed 15 Aug 2023.

Cohen, Jonathan D., et al. “I-COLLIDE: An Interactive and Exact Collision Detection System for Large-Scale Environments.” *Proceedings of the 1995 Symposium on Interactive 3D Graphics - SI3D ’95*, ACM Press, 1995, p. 189-196. *DOI.org (Crossref)*, https://doi.org/10.1145/199404.199437. Accessed 2 Jan. 2024.

Collingridge, Peter. *Pygame Physics Simulation*. 1 Feb. 2010, https://www.petercollingridge.co.uk/tutorials/pygame-physics-simulation/. Accessed 14 Aug 2023.

Eppstein, David. *A Point Quadtree*. 31 May 2005. self-made; originally for a talk at the 21st ACM Symp. on Computational Geometry, Pisa, June 2005, *Wikimedia Commons*, https://commons.wikimedia.org/w/index.php?curid=2489019. Accessed 12 Jun. 2023.

Ericson, Christer. *Real-Time Collision Detection*. CRC Press, 2013, https://doi.org/10.1201/b14581.

Figueiredo, Mauro, et al. “A Survey on Collision Detection Techniques for Virtual Environments.” *Proc. of V Symposium in Virtual Reality, Brasil*, vol. 307, Jan. 2002. Accessed 25 Jun 2023.

Harada, Takahiro. “Chapter 29. Real-Time Rigid Body Simulation on GPUs.” *NVIDIA Developer*, https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulation/chapter-29-real-time-rigid-body-simulation-gpus. Accessed 14 Sep. 2023.

Kroiss, Ryan Robert. *Collision Detection Using Hierarchical Grid Spatial Partitioning on the GPU*. 18 Nov. 2019, https://scholar.colorado.edu/concern/graduate_thesis_or_dissertations/vd66w0289. Accessed 28 Sep. 2023.

Le Grand, Scott. “Chapter 32. Broad-Phase Collision Detection with CUDA.” *NVIDIA Developer*, https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulation/chapter-32-broad-phase-collision-detection-cuda. Accessed 14 Sep. 2023.

Lembcke, Scott. *Chipmunk Game Dynamics Manual*. http://chipmunk-physics.net/release/ChipmunkLatest-Docs/#CollisionDetection. Accessed 15 Dec. 2023.

Petersen, Andrew. *Broad Phase Collision Detection Using Spatial Partitioning - Build New Games*. 8 Oct. 2012, http://buildnewgames.com/broad-phase-collision-detection/. Accessed 24 Aug. 2023.

Rayner, Darcy. “Answer to ‘How Can I Implement Fast, Accurate 2D Collision Detection?’” *Game Development Stack Exchange*, 8 Oct. 2011, https://gamedev.stackexchange.com/a/18271. Accessed 29 May 2023.

Rigid Body Collision — Physx 5.3.1 Documentation. 11 Dec. 2023, https://nvidia-omniverse.github.io/PhysX/physx/5.3.1/docs/RigidBodyCollision.html#broad-phase-algorithms. Accessed 15 Dec. 2023.

Souto, Nilson. "Video Game Physics Tutorial - Part II: Collision Detection for Solid Objects | Toptal®." *Toptal Engineering Blog*, https://www.toptal.com/game/video-game-physics-part-ii-collision-detection-for-solid-objects. Accessed 12 Jun. 2023.

Technologies, Unity. *Unity - Manual: Physics*. https://docs.unity3d.com/Manual/PhysicsSection.html. Accessed 29 Dec. 2023.

Tracy, Daniel J., et al. "Efficient Large-Scale Sweep and Prune Methods with AABB Insertion and Removal." *2009 IEEE Virtual Reality Conference*, 2009, pp. 191–98. *IEEE Xplore*, https://doi.org/10.1109/VR.2009.4811022. Accessed 7 Jun. 2023.

8. Appendix

Appendix A – Simulation code

A.1: Libraries imported (sim.py)

```
import pygame
import random
import math
import time
from numpy import random as nrandom
import itertools
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import numpy as np
```

A.2: Implementation of grid and quadtree data structures (sim.py)[26]

```
# --- ALGORITHM DATA STRUCTURES (grid, interval, quadtree, rectangle) ---
class Grid:
    def __init__(self, boundary):
        self.boundary = boundary
        self.particles = []
    def insert(self, particle):
        if self.boundary.intersects_particle(particle):
            self.particles.append(particle)

class Interval:
    def __init__(self, value, particle, is_min):
        self.value = value
        self.particle = particle
        self.is_min = is_min
    def __repr__(self):
        if self.is_min: return "[" + str(self.value)
        return str(self.value) + "]"

class QuadTree:
    def __init__(self, boundary, capacity, show):
```

[26] Parts inspired by The Coding Train. "Coding Challenge #98.1: Quadtree - Part 1." YouTube, 26 Mar. 2018, www.youtube.com/watch?v=OJxEcs0w_kE. Accessed 15 Aug 2023.

```
        self.boundary = boundary
        self.capacity = capacity
        self.show = show
        self.subdivided = False
        self.particles = []
        if show:
            pygame.draw.rect(screen, (0, 255, 0), (self.boundary.x - self.boundary.w, self.boundary.y -
self.boundary.h, self.boundary.w*2, self.boundary.h*2), 2)
    def insert(self, particle):
        if len(self.particles) < self.capacity:
            self.particles.append(particle)
            return True
        if not self.subdivided: self.subdivide()
        if self.ne.boundary.contains(particle):
            return self.ne.insert(particle)
        if self.nw.boundary.contains(particle):
            return self.nw.insert(particle)
        if self.se.boundary.contains(particle):
            return self.se.insert(particle)
        if self.sw.boundary.contains(particle):
            return self.sw.insert(particle)

    def query(self, region):
        if not self.boundary.intersects(region):
            return []
        found = []
        for particle in self.particles:
            if region.contains(particle):
                found.append(particle)
        if not self.subdivided: return found
        return found + self.ne.query(region) + self.nw.query(region) + self.se.query(region) + self.sw.query(region)

    def subdivide(self):
        x = self.boundary.x
        y = self.boundary.y
        w = self.boundary.w
        h = self.boundary.h

        self.ne = QuadTree(Rectangle(x+w/2, y-h/2, w/2, h/2), self.capacity, self.show)
        self.nw = QuadTree(Rectangle(x-w/2, y-h/2, w/2, h/2), self.capacity, self.show)
```

```
        self.se = QuadTree(Rectangle(x+w/2, y+h/2, w/2, h/2), self.capacity, self.show)
        self.sw = QuadTree(Rectangle(x-w/2, y+h/2, w/2, h/2), self.capacity, self.show)


        self.subdivided = True


class Rectangle:
    def __init__(self, x, y, w, h):
        self.x = x
        self.y = y
        self.w = w
        self.h = h
    def contains(self, particle):
        return particle.x > self.x - self.w and particle.x < self.x + self.w and particle.y > self.y - self.h and particle.y <
self.y + self.h
    def intersects(self, rectangle):
        return not (rectangle.x - rectangle.w > self.x + self.w or rectangle.x + rectangle.w < self.x - self.w or
rectangle.y - rectangle.h > self.y + self.h or rectangle.y + rectangle.h < self.y - self.h)
    def intersects_particle(self, particle):
        particle_distance_x = abs(particle.x - self.x)
        particle_distance_y = abs(particle.y - self.y)
        if particle_distance_x > (self.w + particle.size): return False
        if particle_distance_y > (self.h + particle.size): return False
        if particle_distance_x <= self.w: return True
        if particle_distance_y <= self.h: return True
        corner_distance_squared = (particle_distance_x - self.w)**2 + (particle_distance_y - self.h)**2
        return (corner_distance_squared <= particle.size**2)
```

A.3: Implementation of broad-phase algorithms (sim.py)

```
# --- BROADPHASE ALGORITHMS ---


def brute_force_collision_detection(particles):
    n = 0
    for i, particle in enumerate(particles):
        for particle2 in particles[i+1:]:
            n += 1
            collide(particle, particle2)
    return n
```

```
def quadtree(particles, capacity):
    qt = QuadTree(Rectangle(width/2, height/2, width/2, height/2), capacity, False)
    n = 0
    for particle in particles:
        qt.insert(particle)
        boundary = Rectangle(particle.x, particle.y, particle.size*2, particle.size*2)
        candidates = qt.query(boundary)
        n += (len(candidates) - 1) # candidates includes particle itself
        for candidate in candidates:
            collide(particle, candidate)
    return n


def uniform_grid(particles, p):
    n = 0
    plane = [[Grid(Rectangle(width*(0.5+j)/p, height*(0.5+i)/p, width/(2*p), height/(2*p))) for j in range(0, p)] for i in
range(0, p)]
    for particle in particles:
        i = int(particle.y//(height/p))
        j = int(particle.x//(width/p))
        neighbours = [(i-1, j-1), (i-1, j), (i-1, j+1), (i, j-1), (i, j), (i, j+1), (i+1, j-1), (i+1, j), (i+1, j+1)]
        for neighbour in neighbours:
            try:
                grid = plane[neighbour[0]][neighbour[1]]
                grid.insert(particle)
            except IndexError:
                pass
    for i in range(0, p):
        for j in range(0, p):
            for comb in itertools.combinations(plane[i][j].particles, 2):
                n += 1
                collide(comb[0], comb[1])
    return n

def sweep_and_prune(particles):
    intervals = []
    for particle in particles:
        xmin = particle.x - particle.size
        xmax = particle.x + particle.size
        intervals.append(Interval(xmin, particle, True))
```

```
            intervals.append(Interval(xmax, particle, False))
        intervals.sort(key = lambda interval: interval.value)
        active_particles = []
        pairsx = set()
        for interval in intervals:
            if interval.is_min:
                active_particles.append(interval.particle)
                continue
            # interval is max
            active_particles.remove(interval.particle)
            for active_particle in active_particles:
                pairsx.add((interval.particle, active_particle))
        for pair in pairsx:
            collide(pair[0], pair[1])
        return len(pairsx)
```

A.4: Implementation of live simulation (sim.py)[27]

```
# --- SIMULATION CODE ---

def collide(p1, p2):
    if p1 == p2:
        return False # particle cannot collide with itself
    dx = p1.x - p2.x
    dy = p1.y - p2.y
    distance = math.hypot(dx, dy)
    if distance > p1.size + p2.size:
        return False # did not collide
    tangent = math.atan2(dy, dx)
    p1.angle = 2*tangent - p1.angle
    p2.angle = 2*tangent - p2.angle
    (p1.speed, p2.speed) = (p2.speed, p1.speed)
    angle = 0.5*math.pi + tangent
    p1.x += math.sin(angle)
    p1.y -= math.cos(angle)
    p2.x -= math.sin(angle)
    p2.y += math.cos(angle)
```

[27] Parts adapted from Collingridge, Peter. "Pygame Physics Simulation." petercollingridge.co.uk, www.petercollingridge.co.uk/tutorials/pygame-physics-simulation. Accessed 14 Aug 2023.

```
class Particle:
    def __init__(self, center, angle, size, speed):
        self.x = center[0]
        self.y = center[1]
        self.size = size
        self.speed = speed
        self.angle = angle
        self.colour = (255, 255, 255)
        self.thickness = 0 # fill
        self.time = time.time()
    def tick(self):
        dt = time.time() - self.time
        self.x += math.sin(self.angle) * self.speed * dt
        self.y -= math.cos(self.angle) * self.speed * dt
        self.wall_bounce()
        pygame.draw.circle(screen, self.colour, (self.x, self.y), self.size, self.thickness)
        self.time = time.time()
    def wall_bounce(self):
        if self.x > width - self.size:
            self.x = 2 * (width - self.size) - self.x
            self.angle = - self.angle
        elif self.x < self.size:
            self.x = 2 * self.size - self.x
            self.angle = - self.angle
        if self.y > height - self.size:
            self.y = 2 * (height - self.size) - self.y
            self.angle = math.pi - self.angle
        elif self.y < self.size:
            self.y = 2 * self.size - self.y
            self.angle = math.pi - self.angle
    def __repr__(self):
        return f"({self.x}, {self.y}, {self.size} px)"


def generator(number, size, x_gaussian_spread, y_gaussian_spread):
    particles = []

    if x_gaussian_spread != 0:
```

```
            meanx = width/2
            xs = list(nrandom.normal(loc=meanx, scale=width/x_gaussian_spread, size=number))
        else:
            xs = list(nrandom.uniform(low=size, high=width-size, size=number))

        if y_gaussian_spread != 0:
            meany = height/2
            ys = list(nrandom.normal(loc=meany, scale=height/y_gaussian_spread, size=number))
        else:
            ys = list(nrandom.uniform(low=size, high=height-size, size=number))

        speed = 10
        for i in range(number):
            theta = random.uniform(0, 2*math.pi)
            particles.append(Particle((xs[i], ys[i]), theta, size, speed))
        return particles

class Particles:
    BRUTE_FORCE = "bf"
    SWEEP_AND_PRUNE = "sap"
    QUADTREE = "qt"
    UNIFORM_PARTITIONING = "usp"
    GAUSS_X = "gaussx"
    GAUSS_Y = "gaussy"
    GAUSS = "gauss"
    UNIFORM = "uniform"
    def __init__(self, number, size, tmax, algo, distribution, p, capacity):
        self.number = number
        self.particles = []
        self.tinit = time.time_ns()
        self.tmax = tmax
        self.algo = algo
        self.size = size
        self.distribution = distribution
        self.number_frames = 0
        self.total_time = 0
        self.total_checks = 0
        self.p = p
        self.capacity = capacity
    def generate(self):
```

```
        if self.distribution == Particles.GAUSS_X:
            self.particles = generator(self.number, self.size, 10, 0)
        if self.distribution == Particles.GAUSS:
            self.particles = generator(self.number, self.size, 10, 10)
        if self.distribution == Particles.UNIFORM:
            self.particles = generator(self.number, self.size, 0, 0)
        if self.distribution == Particles.GAUSS_Y:
            self.particles = generator(self.number, self.size, 0, 10)
    def tick(self):
        tframeinit = time.time_ns()
        if tframeinit - self.tinit > self.tmax * 10**9:
            return (self.total_checks/self.number_frames, self.total_time/self.number_frames)
        if self.algo == Particles.BRUTE_FORCE: self.total_checks += brute_force_collision_detection(self.particles)
        if self.algo == Particles.SWEEP_AND_PRUNE: self.total_checks += sweep_and_prune(self.particles)
        if self.algo == Particles.QUADTREE : self.total_checks += quadtree(self.particles, self.capacity)
        if self.algo == Particles.UNIFORM_PARTITIONING: self.total_checks += uniform_grid(self.particles, self.p)
        dt = time.time_ns() - tframeinit
        self.total_time += dt
        for particle in self.particles:
            particle.tick()
        self.number_frames += 1

(width, height) = (2000, 2000)
screen = pygame.display.set_mode((width, height))
pygame.display.flip()
pygame.display.set_caption('Collision Sim')
background_colour = (0, 0, 0)



def run_sim(number, size, algorithm, distribution, p, capacity):
    particles = Particles(number, size, 3, algorithm, distribution, p, capacity)
    particles.generate()
    running = True
    while running:
        screen.fill(background_colour)
        x = particles.tick()
        if x:
            return x
        pygame.display.update()
```

```
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False
    pygame.quit()
```

A.5: Implementation of experiments (sim.py)

```
# --- EXPERIMENTS CODE ---

# Experiment 1
def vary_p():
    print("p      t")
    ps = np.arange(6, 30)
    ts = np.zeros(30-6)
    for i, p in enumerate(ps):
        ts[i] = run_sim(1024, 10, Particles.UNIFORM_PARTITIONING, Particles.UNIFORM, p, 0)[1]
        print(f"{p} {ts[i]}")
    plt.plot(ps, ts)
    plt.xlabel("Number of partitions")
    plt.ylabel(f"Average frame period")
    plt.savefig(f"p.png")

# Experiment 2
def vary_capacity():
    print("κ      t")
    capacities = np.arange(1, 20 , 1)
    ts = np.zeros(capacities.size)
    for i, capacity in enumerate(capacities):
        ts[i] = run_sim(1024, 10, Particles.QUADTREE, Particles.UNIFORM, 0, capacity)[1]
        print(f"{capacity} {ts[i]}")
    plt.plot(capacities, ts)
    plt.xlabel("Capacity of Quadtree")
    plt.ylabel(f"Average frame period")
    plt.savefig(f"capacity.png")

# Experiment 3
def optimize_p_n():
    print("n    r   Tavg    p*")
    ns = [4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048]
    maxs = []
```

```
    for n in ns:
        size = 20
        prev_prev = run_sim(n, size, Particles.UNIFORM_PARTITIONING, Particles.UNIFORM, 1, 0)[1]
        prev = run_sim(n, size, Particles.UNIFORM_PARTITIONING, Particles.UNIFORM, 2, 0)[1]
        i = 3
        while True:
            cur = run_sim(n, size, Particles.UNIFORM_PARTITIONING, Particles.UNIFORM, i, 0)[1]
            if cur > prev and prev_prev > prev:
                # local minima
                print(n, size, prev, i-1)
                maxs.append(i-1)
                break
            prev_prev = prev
            prev = cur
            i += 1
    plt.plot(ns, maxs)
    plt.xscale("log")
    plt.xlabel("Number of particles (n)")
    plt.ylabel("Optimum value of p")
    plt.show()

# Experiment 4
def uniform_dist(size):
    optimum_ps = [2, 2, 3, 3, 7, 8, 11, 15, 17, 21]
    capacity_opt = 5
    numbers = [2**i for i in range(2, 12)]
    alg_results = [["sap"], ["qt"], ["usp"]]
    for i, n in enumerate(numbers):
        for j in range(0, 3):
            alg_results[j].append(run_sim(n, size, alg_results[j][0], Particles.UNIFORM, optimum_ps[i],
capacity_opt)[1])
    print(alg_results)
    plt.xlabel("n")
    plt.ylabel("Average frame period")
    plt.xscale("log")
    plt.yscale("log")
    for j in range(0, 3):
        times = alg_results[j][1:]
        name = alg_results[j][0]
        plt.plot(numbers, times, label=name)
```

```
    plt.legend(loc="upper left")
    plt.savefig(f"uniform_{size}px.png")


# Experiment 5
def gaussx_dist():
    p=8
    capacity_opt = 5
    numbers = [2**i for i in range(2, 12)]
    alg_results = [["sap"], ["qt"], ["usp"]]
    for i, n in enumerate(numbers):
        for j in range(0, 3):
            alg_results[j].append(run_sim(n, 10, alg_results[j][0], Particles.GAUSS_X, p, capacity_opt)[1])
    print(alg_results)
    plt.xlabel("n")
    plt.ylabel("Average frame period")
    plt.xscale("log")
    plt.yscale("log")
    for j in range(0, 3):
        times = alg_results[j][1:]
        name = alg_results[j][0]
        plt.plot(numbers, times, label=name)
    plt.legend(loc="upper left")
    plt.savefig(f"gaussian_x.png")


# Experiment 6
def gaussy_dist():
    p = 8
    capacity_opt = 5
    numbers = [2**i for i in range(2, 12)]
    alg_results = [["sap"], ["qt"], ["usp"]]
    for i, n in enumerate(numbers):
        for j in range(0, 3):
            alg_results[j].append(run_sim(n, 10, alg_results[j][0], Particles.GAUSS_Y, p, capacity_opt)[1])
    print(alg_results)
    plt.xlabel("n")
    plt.ylabel("Average frame period")
    plt.xscale("log")
    plt.yscale("log")
    for j in range(0, 3):
```

```
        times = alg_results[j][1:]
        name = alg_results[j][0]
        plt.plot(numbers, times, label=name)
    plt.legend(loc="upper left")
    plt.savefig(f"gaussian_y.png")



# Experiment 1
# vary_p()

# Experiment 2
# vary_capacity()

# Experiment 3
# optimize_p_n()



# Experiment 4
#uniform_dist(3)
#uniform_dist(10)
#uniform_dist(25)

# Experiment 5
#gaussy_dist()

# Experiment 6
# gaussx_dist()

pygame.quit()
```

Appendix B – Raw data

T and Tavg refers to frame period in nanoseconds

Each array in Experiments 4-7 has data for n in the powers of 2 from 4 to 2048

p refers to the optimum vale of p*

B.1: Raw data for experiment 1

p t
6 16367375.0
7 14280282.75862069
8 12858217.948717948
9 12210319.018404908
10 11357309.941520467
11 11156800.0
12 10673646.067415731
13 10613668.539325843
14 10347060.773480663
15 10383206.703910615
16 10268756.906077348
17 10296044.198895028
18 10392905.02793296
19 10401433.333333334
20 10386600.0
21 10526245.810055867
22 10711327.683615819
23 10828090.909090908
24 10983057.471264368
25 11252263.157894736
26 11373159.76331361
27 11625404.761904761
28 11844793.93939394
29 11919781.818181818

B.2: Raw data for experiment 2

κ t
1 37580731.34328358
2 32796857.14285714
3 31197525.0
4 30262402.43902439
5 30565158.536585364
6 30442283.950617284
7 30886987.65432099
8 31164716.049382716
9 31350212.5
10 31112506.172839507
11 31503075.0
12 32099430.379746836
13 31974341.7721519
14 32939883.116883118
15 33181311.68831169
16 33164805.194805194
17 33905346.666666664

18 34859739.7260274
19 34703162.16216216

B.3: Raw data for experiment 3

n r Tavg p*
4 20 57753.51014040562 2
8 20 91048.78048780488 2
16 20 154155.59157212317 2
32 20 271516.97530864197 3
64 20 581207.5471698113 5
128 20 1081246.2121212122 7
256 20 2351242.718446602 7
512 20 4958501.7921146955 14
1024 20 11238887.5 15
2048 20 26174864.197530866 24

B.4: Raw data for experiment 4

3px
[['sap', 10231.204620462046, 13414.77, 20631.77, 36348.291946308724,
68252.53103448276, 140269.80427046263, 332520.47744360904, 938380.3112033195,
3435692.7204301073, 12897833.770642202], ['qt', 14676.801980198019,
35669.91333333333, 76125.09731543624, 186457.71821305843, 450721.89130434784,
1146515.7250996016, 2934777.924170616, 7413781.805194805, 18101455.114583332,
44005321.807692304], ['usp', 35391.7342192691, 56086.578595317726,
96962.29966329967, 178241.36458333334, 326142.3417266187, 663414.8593155893,
1349150.8215767634, 2879505.970588235, 6393327.615384615, 13859644.504761904]]

10px
[['sap', 10505.56, 15240.433447098976, 22544.387543252597, 40695.82746478873,
89224.67391304347, 205326.05263157896, 560713.4738955824, 2233939.797101449,
9384768.961538462, 38923506.81818182], ['qt', 15097.501683501683, 37163.67010309279,
79380.73776223777, 200400.59782608695, 512181.8244274809, 1278430.9707112971,
3268469.8844221104, 8478778.11971831, 21005099.850574713, 52580253.27272727],
['usp', 35141.13559322034, 58600.05902777778, 95133.4542253521, 187217.36363636365,
356092.8533834586, 706504.2390438247, 1498606.5682819383, 3222643.492146597,
7173093.965277778, 16816716.13043478]]

25px
[['sap', 10471.96357615894, 15788.847176079735, 23256.20930232558,
45883.72818791946, 102960.0813559322, 274544.1236749117, 874283.6147859922,
4367466.098360656, 19608458.066666666, 94101168.44444445], ['qt',
14587.168316831683, 36234.60132890366, 77224.55369127517, 190103.35034013606,
474190.67253521126, 1183650.5984555983, 3026837.6854460095, 8017098.114864865,

21827512.13095238, 61803275.578947365], ['usp', 35792.07284768212,
58623.77408637874, 95813.10402684564, 174199.8, 332313.5804195804,
653211.0073260074, 1360963.5346938774, 3122556.9285714286, 7763763.0,
22355987.07792208]]

B.5: Raw data for experiment 5

gaussian x, size=10px
[['sap', 24709.35960591133, 35519.4401244168, 49907.6682316119, 92583.46333853355,
206064.78405315615, 544197.8798586573, 1729900.6211180124, 6682875.0,
28113329.545454547, 115009480.0], ['qt', 32189.473684210527, 75867.80715396578,
167684.375, 427196.8, 1003109.756097561, 2554247.7477477477, 6303232.974910394,
15206738.255033556, 36938157.14285714, 88989741.93548387], ['usp',
130388.21752265861, 170048.8188976378, 242222.4025974026, 372909.5238095238,
649410.5621805792, 1184548.623853211, 2802394.230769231, 6748102.362204724,
20834270.27027027, 72148342.10526316]]

B.6: Raw data for experiment 6

gaussian y, size=10px
[['sap', 31435.075885328835, 33375.420875420874, 45689.44099378882, 79376.5625,
150698.11320754717, 351699.83686786296, 936670.9090909091, 2901722.222222222,
10851097.701149425, 41851500.0], ['qt', 39700.170357751274, 87680.13468013468,
172783.2512315271, 428677.8846153846, 1022259.5818815331, 2546591.5178571427,
6336687.050359712, 15213554.054054054, 36025295.774647884, 88809290.32258065],
['usp', 160063.97306397307, 186848.22934232716, 226355.24256651016,
368386.32750397455, 636368.9482470785, 1204140.510948905, 2812431.654676259,
7254413.223140496, 22192961.904761903, 70624421.05263157]]