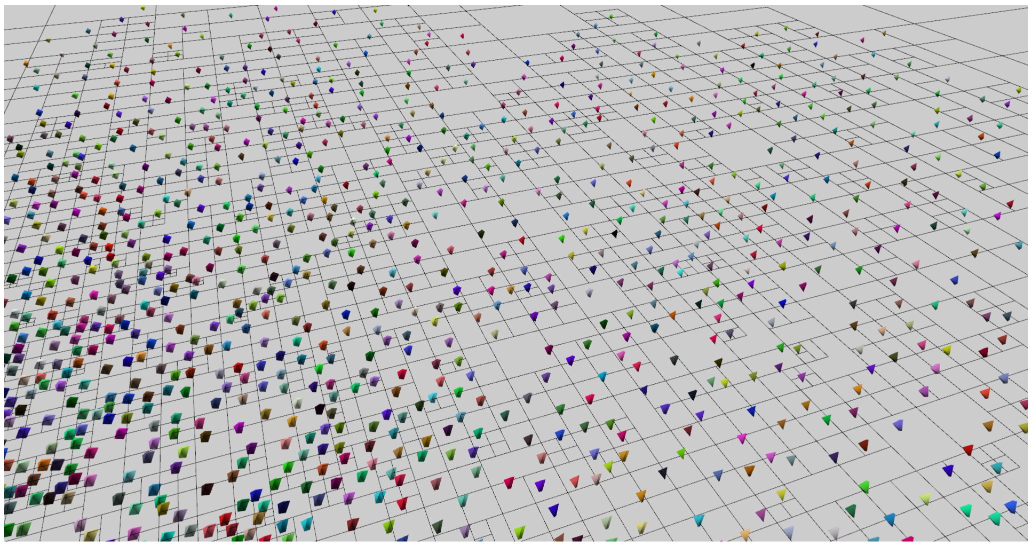
An Investigation into an Assortment of Flocking Algorithms

Calum Devlin

22nd August 2016



1 Introduction

Boids and flocking systems are a relatively recent addition to the programmers arsenal, attributed to Craig Reynolds in 1986.[1] A large group of creatures - such as a school of fish, a herd of cattle, or a flock of boids - are abstracted into points which follow a mutual set of rules, much like a particle system. However, boids are more advanced than a particle system, in that they consider the behaviour of each other, and require velocities to more accurately approach the behaviour of real birds. Each boid can use such data about their neighbours as an influence to how their own behaviour should manifest. If handled correctly, this can quickly give rise to large, multi-agent systems which demonstrate individually complex behaviour, yet simultaneously create a unified effect for the flock as a whole.

2 Boids[3, 5]

2.1 Implemented Behaviour Rules

2.1.1 Avoidance

The most crucial behaviour for a simulation is to prevent objects intersecting. This is achieved with a repulsive force between any pair of boids which are too close to each other. For best results, the force should be considered inversly proportional to their mutual proximity.

My implementation includes a variation on the standard behaviour: instead of averaging the position of all boids which are too close, my boids are designed to prioritize the distance and only fly away from the closest boid which is too close. This avoids the unrealistic behaviour where a lot of boids far away can repel a boid into a single boid which is too close.

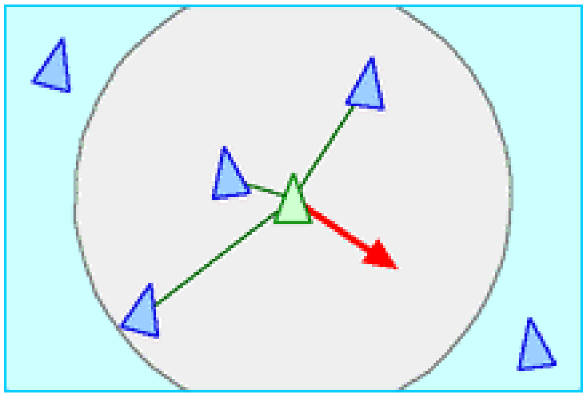


Figure 1: The Avoidance Rule

2.1.2 Cohesion

Opposed to the avoidance rule is cohesion. Again, each boid considers its nearby neighbours, but instead of trying to move away from any boid which is too close, this rule calculates the average position as an influence for boids to move towards. It is good practice to define the area which contains valid neighbouring boids for cohesion as a larger area than neighbouring boids which must be avoided.

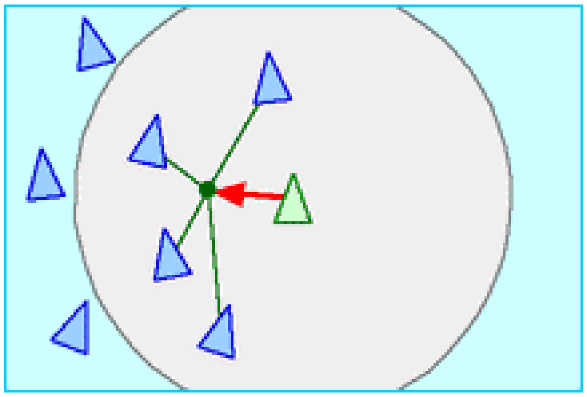


Figure 2: The Cohesion Rule

2.1.3 Alignment

Alignment is the third crucial component to calculating a boids behaviour - similar to the implementation of the cohesion rule, boids inspect their radius of neighbours, and try to align themselves with the average of their neighbours velocity. Without this rule, the result can allow for groups of boids to fly at cross-purposes through each others paths, and create unrealistic simulations.

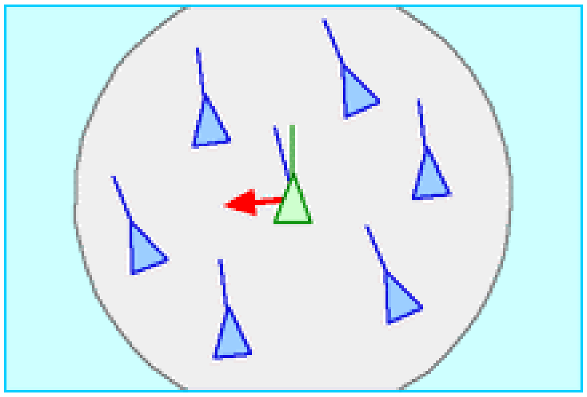


Figure 3: The Alignment Rule

2.1.4 Field of View

One common augmentation to implement is to reduce the number of boids that contribute to other rules, based on their relative position to the boid under consideration. Boids which are in front retain their influence, while boids which are behind are labelled out of sight - if one boid cannot see another boid, then it is considered unable to retrieve any information about that boid to use in its decision making.

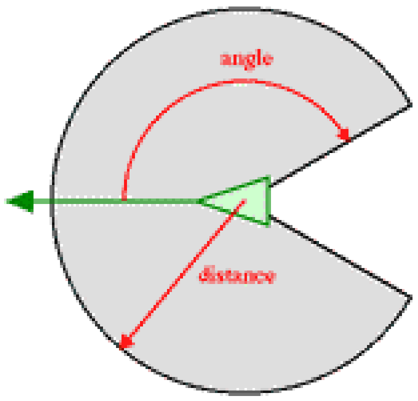


Figure 4: Reducing the Field of View of a Boid

2.1.5 Flock Target

Instead of flying around aimlessly, a flock of boids can be given a distant destination to aim for, or a target to follow. A static target will terminate with circling boids focussing on avoiding collision; mobile targets can be used to provide an influence on how to navigate a landscape. Following targets is a responsibility best assigned to boids leading the rest of the flock.

2.1.6 Behaviour: Order of Preference

When given a set of behavioural rules to follow, the simplest response for a boid to take is to attempt to obey all rules equally and simultaneously. However, this can result in conflicting directions the boid attempts to follow, which can cancel each other out and leave boids with no direction to take.

This can be improved by defining some rules as absolute (must always be obeyed) and defining optional rules as taking precedence over others. In my implementation, boids always attempt to follow the target and avoid boids which are too close; alignment only applies if there exists a boid within range to align to, and boids will only attempt to average their position when there are no boids close enough to trigger the conditions for avoidance.

One further customization my implementation uses is to set a limit on the number of neighbouring boids that can influence the alignment rule. If left unrestricted, the mutual alignment and cohesion causes feedback in small flocks which takes a long time to respond to a global change in alignment, such as following a target. My implementation includes a condition that small sub-flocks of boids are to ignore the cohesion rule and focus on following the target, which steers them back towards the main body of the flock quicker, at which point the cohesion rule can be applied again.

2.2 Additional Behaviour (not implemented)

2.2.1 Obstacle Avoidance

In addition to avoiding each other, boids can be provided with static elements in the scenery to avoid collision with as well, such as buildings, trees or a ground surface. For such objects, the rules of alignment and cohesion usually do not apply to such objects. However, such objects are better represented as a volume, rather than a point, which has been noted to introduce unrealistic behaviour if not handled carefully.[5]

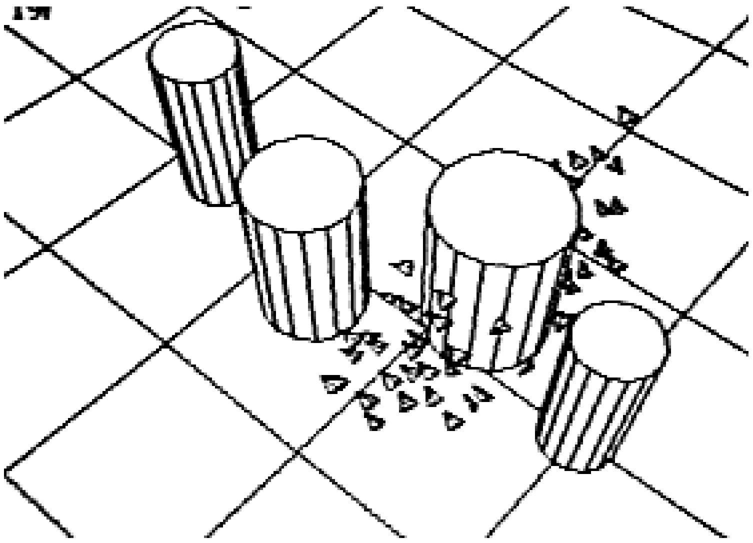


Figure 5: Building

2.2.2 Predator

There can be more than one type of agent inhabiting the simulation space. Predatory boids can be introduced to affect the priority of rules: prey boids should attempt to maintain a safe distance from any dangerous elements that are attacking before rejoining the flock. Predators do not need to implement the rule of avoidance if they are hunting down another boid. A good predator should have a trade-off in abilities to interact with the flock, E.G. an increased velocity parameter in exchange for reduced turning capability. This could be extended even further by defining an arbitrary quantity of boid types, and their interactions with each other.

2.2.3 Behaviour: Weighting

As another alternative to obeying every order at every step, boids are given a finite amount of "action" to take on each turn. All rules are given a precedence ordering again and a cost of "action". Once a boid runs out of "action", no more rules can influence a boids behaviour.

Another style of weighting is based on boid proximity: closer boids produce greater influence than ones further away, as is intuitive given no other distinction between boids.[1]

3 Planar Structures

3.1 Naïve Algorithm

The most basic, intuitive and easiest to remember algorithm. Every boid is compared against every other comparable item. Unfortunately, simple algorithms are rarely efficient in terms of time, and it remains the case here: $\mathcal{O}\left(n^2\right)$time.

3.2 Quadtree Algorithm

The main fault with the naïve approach is that the boids have no spatial memory of each other. For large flocks, many boids will exist far away from the space of another boids radius of awareness. Much time is wasted on these comparisons, so a structure is required that can quickly eliminate large subsections of the flock from consideration.

A quadtree is a good structure to employ for this purpose. Each node of a quadtree points to four children, which are either leaves ,or recursively, are nodes themselves. Leaves are defined by a capacity of elements they can hold, and when that threshold is exceeded, splits into four children, and the elements are distributed between them.

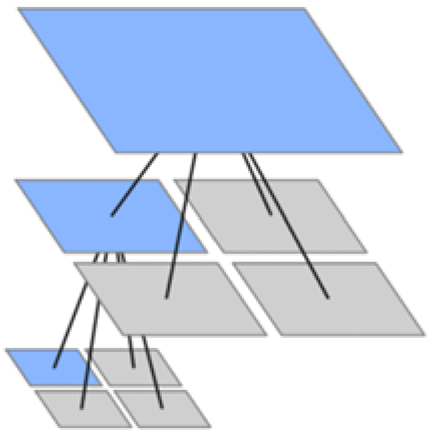


Figure 6: Quadtree Structure

For flocking boids, the quadtree nodes are defined over space, so that boids which are spatially close are likely to share a parent only a few levels away, although it is not impossible for two boids to be on either side of a boundary such that their common parent is the root quadtree.

3.2.1 Efficiency

Each boid has to at least traverse up the tree until it reaches a node which completely contains it's radius of awareness, and then recursively iterate through the nodes and leaves looking for nodes which it overlaps and intersects.

My implementation is inefficient, in that it does not maintain the tree between steps, but the reconstruction quickly provides two elements of efficiency.

Firstly, every boid is provided with a pointer directly to the smallest quadtree that it completely resides inside.

Secondly, each quadtree maintains a list of all boids which occupy children leaves, in addition to its four quadtree children nodes. If, at any stage, the square of a quadtree is completely inside the radius of a boid, then every leaf is a candidate to be a neighbour of the boid, and it becomes possible to trade quadtree recursion for boid neighbour iteration.

3.3 Delaunay Graph[4]

My final, incomplete, algorithm incorporates a Delaunay Triangulation to maintain a notion of spatial proximity between the boids.

A Delaunay graph is the dual of the Voronoi diagram: where the Voronoi diagram displays the irregular polyhedra and polygons of space that are closest to a point, with edges and facets being equidistant from two points, a Delaunay graph connects points that share a Voronoi boundary. This results in triangles whose circumcircles do not contain any other point, which is equivalent to three boids who are "closest" to each other. Triangles tend towards the equilateral and away from long, thin triangles, although such triangles can still be found with the correct placement of points, such as on the convex hull.

One algorithm is required to create any triangulation of the boids locations, but should only require a single execution in setup. Thereafter, as the boids move in space, only a maintenance algorithm is required, such as an Edge-Flip algorithm. For random triangulations, this algorithm runs in $\mathcal{O}\left(n^2\right)$time at worst, but as each frame only disturbs the boids locations a minor amount, many Delaunay edges persist and fewer edges require flipping and invalidating other edges.

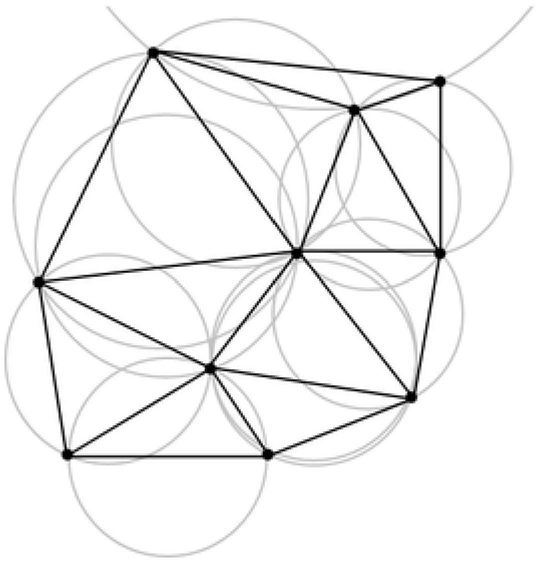


Figure 7: Delaunay Graph

One advantage of this approach to the other two, is that it maintains a relationship between a divided flock, E.G. if the boids split in half to fly around a building. Boids on the edges of the convex hulls of each sub-flock will remain connected to other sub-flocks, and appropriate action can be taken to join sub-flocks back together. Other algorithms would return false positives from boids on the opposite side of the same sub-flock, or require additional complexity to detect the other sub-flock.

4 Volumetric Structures

The original intent was to uplift the structures and algorithms into the third dimension, in order to closer emulate flocking behaviour. Due to time constraints, such behaviour has not been universally implemented and tested.

4.1 Naïve Algorithm

The Naïve structure is the easiest to uplift, as there is no structure, and thus it requires no extra code at all. The boids would require some vertical jitter to stop being restricted to the plane, but this is true of all algorithms. With no notion of spatial proximity when comparing all boids against each other, there is no technical hurdle to increasing the number of dimensions a naïve flocking algorithm can operate in.

4.2 Octree Algorithm

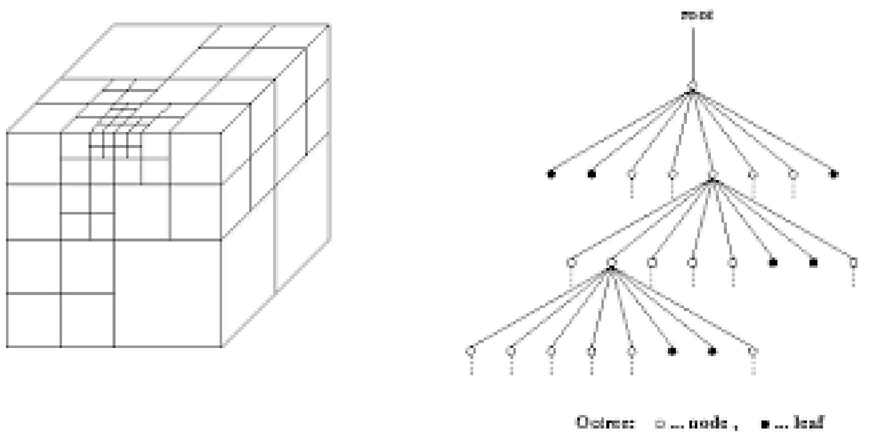


Figure 8: Octree cubes

Uplifting from a Quadtree structure to an Octree structure should be mostly as easy as transitioning from squares to cubes that form the basis of the respective structures. The QuadTree class contains non-functional code in several places where two-dimensional comparisons can intuitively be expanded to include additional dimensions.[6]

However, there is currently one method for testing overlap which cannot be quickly extended.

In the planar quadtree, it looks to detect if there is any overlap at all between a square quadtree and a circular boid radius. There are three conditions which can achieve this:

1. Complete inclusion - either the circle is inside the square, or vice versa.

2. Any corner of the square is inside the radius.

3. The radius only overlaps one edge of the square. Taking advantage of the alignment of the squares, it becomes possible to test the cardinal extremes of the radius against the size and location of the square.

For a cubic octree, the conditions extend to:

1. Complete inclusion.
2. Any corner of the cube is inside the radius.
3. The radius only overlaps one face of the cube.
4. The radius only overlaps one edge. No corners or cardinal extremes overlap, although two of each will be close.

The difficulty lies in the fact none of the extreme points of each object overlap the other, but that does not imply that the objects do not overlap. The three orthographic cardinal planes that bisect the sphere must be individually tested for overlap with the quadtree.

4.3 Delaunay Algorithm

Delaunay graphs in the planar case are well documented. The mathematics behind Delaunay graph triangulations in higher dimensions increases in complexity. In only two dimensions, the Flip Graph of a planar set of points is completely connected, and it is possible to traverse from one trinagulation to any other, including Delaunay. For five dimensions and higher, it is theorised that the Flip Graph can become disconnected, which would mean there would be no guarantee that the algorithm would terminate in a valid Delaunay triangulation. Though in practice, a five-dimensional flock is not a commonly required simulation.[7]

5 Further Work

The three structures presented above are by no means the limit of what is possible with flocking systems.

5.1 Improvements to the Existing Algorithms & Structures

5.1.1 Naïve Algorithm

For all its simplicity, the naïve algorithm can be sped up to be twice as fast via reflexiveness: If and only if boid 1 is close to boid 2, then boid 2 is close to boid 1, and they can be evaluated at the same time. However, this does not improve the bounds; it continues to execute in $\mathcal{O}\left(n^2\right)$time.

5.1.2 Quadtree Algorithm

My current quadtree approach is inefficient in that it gets reconstructed from scratch at every frame. An alternative - and more complicated - approach would be to save the tree between frames, and only reassign boids that cross the boundaries of the quadtree structure.

My current approach assumes all boids are limited to remaining in a defined area/volume, and their behaviour outside the boundaries is undefined. If the area is not known beforehand, it would be better practice to allow the root of the quadtree structure to generate a larger parent as the new root. Alternatively, for the current quadtree which gets recalculated at every frame, a more compact and balanced tree could be achieved by recording the position and the size of the flock to use for the parameters of the root node.

The methods for testing overlap are designed around the assumption that a node of the Quadtree can be represented by a square (or a cube in the Octree equivalent) and the area of interest that affects a boid's movement is circular/spherical respectively. As noted in 4.2, the strive for accuracy and speed has led to a complex set of conditions that must be tested. These conditions could be simplified by treating both volumes as either spheres or cubes, but this would reduce accuracy by falsely identifying possible matches that the current heuristic would identify as impossible and thus ignore.

5.1.3 Delaunay Algorithm

The Delaunay approach has a more restricted radius of awareness compared to the other two approaches described. Boids will always be aware of at least their nearest neighbour, so an appropriate response for avoidance will exist. But responses for alignment and cohesion will be limited by only inspecting immediate neighbours in the Delaunay graph. The range could be extended if required by inspecting neighbours of neighbours, or continuing through the graph until the boids exceed the radius of consideration; care must be taken to avoid evaluating boids repeatedly, or by providing a weighting depending on the length of the chain of neighbours.

The initial algorithm to create the Delaunay graph is on obvious candidate for improvement, currently running in cubic $\mathcal{O}\left(n^3\right)$time. It remains a low cost however, as it is only executed once in the constructor. There is an interesting relationship between Delaunay Graphs and Convex Hulls, such that an algorithm that generates a convex hull can quickly be transformed into a Delaunay graph through the use of an extra dimension. Boid positions are uplifted with their Euclidean distance, E.G. in two dimensions$(x, y) \implies \left(x, y, x^2 + y^2\right)$to lie on a parabolic curve. The convex hull of the uplifted points is calculated, and any upward-facing faces are discarded. The remaining structure is projected back into the original space, and becomes a Delaunay graph.[9]

Due to time constraints, no Edge-Flipping algorithm has been implemented for the Delaunay flock. 2-D edge-flipping algorithms are designed to start work on any valid triangulation of points and return a valid Delaunay triangulation in

$n \log n$. However, in a random triangulation, there will be many non-Delaunay edges that require correcting and have a domino effect on other edges. But starting from a valid Delaunay triangulation subject to minor disturbance (I.E., the small motion of movement at every frame), many Delaunay edges will remain, and the domino effect will be reduced, resulting in a faster execution.[8]

5.1.4 Parallelism

My Naïve and Quadtree approaches contain some limited implementation of parallelism, which can increase the size of the flock that can be rendered at maximum framerate. The Delaunay algorithm contains no parallel elements, but it would be interesting to see if the Edge-Flip algorithm can be improved in this manner.

5.2 Alternative Algorithms

5.2.1 Binary Partition

The Binary Partition is a generalisation of the Quadtree and Octree algorithms. At every level, the flock is divided in half, like the quadtree divides into four children, and the octree maintains eight.

An ideal binary partition is balanced, I.E., both children have the same number of leaves.

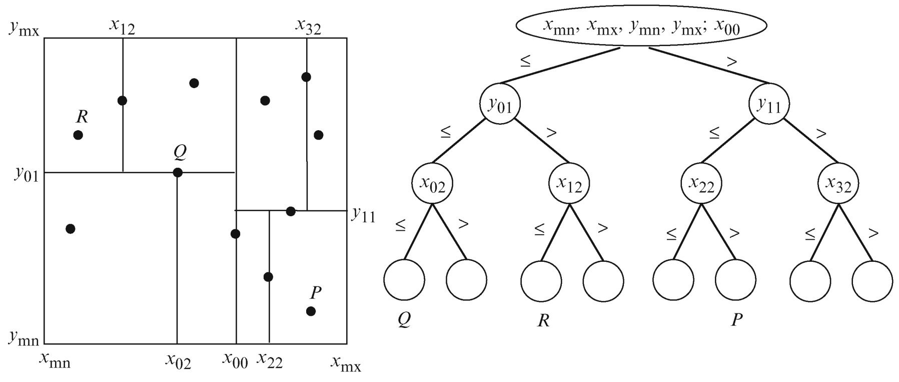


Figure 9: Binary

5.2.2 Bucketing Algorithm[1, 2]

This contains elements that have been incorporated into the Quadtree algorithm already presented. The infinite plane or space is subdivided into an array of buckets or bins, with a size that is defined relative to the radius of a boid's awareness. Such constraints on size means boids are only required to check the current bin they are inside, and a small, finite number of adjacent bins. The size of a bin remains constant, which can result in no features which prevent all boids gathering in one spot, and requiring evaluation of the same order as the Naïve algorithm.

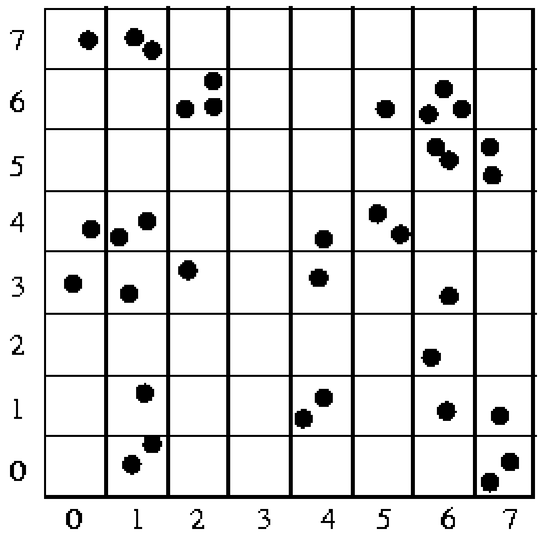


Figure 10: Bucketing

References

[1] Reynolds, C., 2007. *Boids* [online]. Available from: www.red3d.com/cwr/boids.

[2] Ohlsson, J. & Lindgren, M., 2016. *Boids Project* [online]. Available from: iontea.github.io/dgraf-project.

[3] Parker, C., 2007. *Boids Pseudocode* [online]. Available from: www.kfish.org/boids/pseudocode.html.

[4] Kreveld, M., 2016. *Computational Geometry* [online]. Place of Publication: Utrecht University. Available from: www.cs.uu.nl/docs/vakken/ga.

[5] Parent, R., 2008. *Computer Animation Algorithms & Techniques.* 2nd Edition. Amsterdam: Morgan Kaufmann PUblishers.

[6] Portegys, T. E. & Greenan, K. M. Managing Flocking Objects with an Octree Spanning a Parallel Message-Passing Computer Cluster [online].

[7] Maur, P., 2002. Delaunay Triangulation in 3D [online].

[8] Su, P. & Drysdale, R. L. S., 1996. A Comparison of Sequential Delaunay Triangulation Algorithms [online].

[9] Shewchuk, J., 2012. *Two-dimensional Delaunay triangulations* [online]. Available from: people.eecs.berkeley.edu/~jrs/papers/meshbook/chapter2.pdf.