Hierarchical Impostors for the Flocking Algorithm in Three Dimensional Space.

Ph.D. in Computer Applications

Noel O' Hara B.Sc., M.Sc.

Computer Applications
Dublin City University
Dublin
Supervisor: Dr. Mike Scott

February 2002

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Doctor of Philosophy in Computer Applications is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work

Signed Noel O' Hara ID 96971223

Date 29 Jan 2002

Acknowledgements

I would like to thank my supervisor Mike Scott for his help and guidance throughout my post graduate time in DCU I would like to thank all the other post grads and staff in the Computer Application Department Special thanks to John Kelleher, Darragh O' Brien, Alan Egan, Jerh O' Connor, Ger Quinn, Tom Sodring and the DCU basketball club for making it a more fun place to work I would like to thank my family for their support all through my education

Table of Contents

Abstract

The availability of powerful and affordable 3D PC graphics boards has made rendering of rich immersive environments possible at interactive speeds The scene update rate and the appropriate behaviour of objects within the world are central to this immersive feeling This thesis is concerned with the behaviour computations involved in the flocking algorithm, which has been used extensively to emulate the flocking behaviour of creatures found in nature The main contribution of this thesis is a new method for hierarchically combining portions of the flocks into groups to reduce the cost of the behavioural computation, allowing far larger flocks to be updated in real-time in the world

Chapter 1

Introduction

1 1 Rendering

Real-time visual simulation of complex three dimensional computer graphic environments has become increasingly important as the personal computer becomes more capable of handling complex graphical imagery As the viewer moves around the world, the view of the scene must be updated by the graphics system Computer graphics environments are typically constructed from polygons The time taken for this update is related to the number of polygons that must be drawn for a given viewpoint Navigating these environments requires a trade-off between realism and speed Either slow drawing of a scene containing many polygons or faster rendering of a scene containing fewer polygons Complex environments could contain many millions of polygons, from any given viewpoint and the number of polygons to be drawn may overload even the most powerful graphics workstations Many algorithms combat this by imposing a hierarchical structure on the scene The algorithms associated with these structures are used to cull large portions of the scene from the rendering process [Air91, Cha95, Tel92, Lue95] These approaches work best for static environments where large portions of the scene are either outside the viewer's field of view or are obscured from view by nearer objects

The visible portions of the scene may still contain many thousand polygons and must be drawn on the screen Several approaches are based on impostors where objects or groups of objects are replaced by a faster and simpler version to be drawn to screen [Che96, Lin94, Mac95, McM95, Sch96, Sh96] The idea behind these algorithms is to trade image quality for interactivity in situations where the environment is too complex to be rendered in full detail

The above approaches work well for mostly static scenes, but for moving objects expensive update operations are performed on the hierarchical structure to reflect the new positions of the objects Sudarsky's [Sud96] approach replaces hidden moving objects with a static volume that contains the object for a time period, so the structure need not updated until the object becomes visible or the time period elapses

The goal of our research is the development of a system that allows interactive rendering of large numbers of objects, which exhibit flock-like motion This is achieved by applying an impostor algorithm to the flocking behaviour

1 2 Flocking

Craig Reynolds [Rey87] introduced a distributed agent based flocking model in which each flock member follows some simple rules This is an extension of particle systems introduced by [Ree83], and is closely related to behavioural animation His paper refers to each simulated bird, fish etc as a **boid** The same terminology is used in this thesis The behaviours involved in flocking, schooling or herding are as follows

Avoidance *Avoid colliding with nearby flock-members*
Match Velocity *Attempt to match the velocity of nearby flock-member*
Flock Centring *Attempt to stay close to the nearby flock-members*
Avoid Obstacles *Avoid obstacles in the environment*

Only obstacles and other boids within a certain visibility range ***R*** are accounted for in the behavioural computations If an obstacle or boid is within a certain distance of a particular boid, then it is visible to that boid and is included in its behavioural computations Each of the behaviours has a weighting associated with it, closer flock members having more an effect than a more distant one The weighting is proportional to the square of the distance between the two flock members The flock members adhere to a basic flight model whereby each has a maximum and minimum speed and maximum acceleration After considering the above behaviours, the resulting acceleration is trimmed to stay in compliance with the flight model Some research has been performed to optimise the behavioural computations of herding creatures [Car97] Their approach focuses on replacing computationally expensive rigid-body dynamics with a simpler particle simulation of herding for one-legged creatures Another approach [Che97} focuses on a virtual fun park with bumper cars and tilt-a-whirl rides Their approach uses buffering of states and computing the probability of a particular state after a certain time interval to increase the efficiency of the state computation Our approach is more specific to the flocking algorithm

1 3 Hierarchical Neighbouring Finding.

In the flocking algorithm, each boid is tested against every other in the world to either include it as a nearby neighbour (within a certain range of the boid) or disregard it Therefore the complexity of the algorithm is $\mathbf{O(n^2)}$ where **n** is the number of boids in the world Reynolds presents an approach using a spatial grid to accelerate the neighbour finding algorithm [Rey2000] Our approach to neighbour finding acceleration is more memory efficient for very larger environments This thesis introduces an algorithm, which computes the behaviour of a group of flock members as a whole rather than computing flocking behaviour for each boid individually

To accelerate the nearby neighbour finding algorithm the scene is first pre-processed by inserting all the objects in a hierarchical structure During runtime, as the boids move around the world, the structure is updated to reflect the new positions of the boids The neighbours of each boid are found by traversing the hierarchical structure, culling large numbers of boids from the computation To accelerate the update phase, we use a technique presented by Sudarsky [Sud96]

1 4 Hierarchical Impostors

During the initial period when the boids are forming into a flock, each of the boids' nearby neighbours may change frequently Once the flock begins to fly in a stable pattern it will remain in a stable state until (a) it meets another flock, (b) or the goal is changed for the boids, (c) or an obstacle is in its path As a flock of boids becomes stable, the boids' velocity varies little from frame to frame There is very little acceleration, only small adjustments in velocity enable the boids to stay close to their neighbours while travelling in the same direction These attributes of the flock are used in devising a more efficient flocking algorithm

The main contribution of the thesis is our approach to group together stable groups within the flock and to compute their behaviour as a whole rather than computing each boid's behaviour in the group individually Computing the group's behaviour is an order of magnitude faster than computing the individual boids behaviour

1 5 Thesis Outline

Chapter 2 outlines some of the considerations needed to render 3D worlds on a computer screen We give a brief history of the field, and outline current research, especially rendering of scenes that contain large numbers of dynamic objects In chapter 2 we also outline a brief history and overview of behavioural animation and flocking behaviour of creatures on computer We outline current research that attempts to increase the efficiency of the behaviour computation of objects in the scene Chapter 3 outlines our approaches to increasing the efficiency of the flocking algorithm in a 3D world using a hierarchical structure to accelerate locality queries Chapter 4 describes our novel approach to further accelerating the flocking behaviour computations by using hierarchical impostors Chapter 5 contains our test and results, obtained from running the implementation of our algorithm The results are obtained directly form the code Chapter 6 contains our conclusions and discussion on future avenues of research in this area

Chapter 2

State Of The Art Review

2 1 Introduction

In this chapter we give an outline of current research for rendering computer graphics worlds outlining some of the fundamental algorithms for correct rendering of a model These will form a foundation for the reader for the following sections Section 2 2 outlines the current research in rendering Sections 2 3 outlines the particle systems approach to animation Section outlines come of the areas where behaviour animation is used in computer graphics Section 2 4 outlines the algorithm used to emulate flocking Section 2 4 12 describes the efficiency considerations in implementing the flocking algorithm Sections 2 5 and 2 6 outline current techniques for acceleration behaviour computations used in animation The final section presents a short review of the chapter

2.2 Rendering

2 2 1 Visible Surface Determination

Computer graphics is roughly divided into two areas One considers the hardware aspects and the other studies algorithmic aspects It is the latter area where the focus of this section lies

The system must compute the objects that are visible for the observer The problem of determining which parts of the objects that are visible and which parts are hidden is called the ***hidden surface removal*** problem The system decides which object is visible for each pixel on the screen so that it can be given the colour of the object

Another approach is to compare each object with every other object and determine which part of the object is visible from the viewpoint and then draw that part of the object This is referred as an ***object precision*** algorithm, object precision algorithms are performed at the precision at which the object is defined

Both of the above approaches get very time consuming as the number of objects grows in the scene This can be especially said for the second approach Typically in an interactive walk/fly-through of an environment the visible portion of the model and the projected image changes very little from frame to frame Sutherland, Sproull, and Schumaker [Sut74] shows how visible-surface algorithms can take advantage of *coherence* -the degree to which parts of an environment or its projection exhibit local similarities

2.2 2 Binary Space Partition (BSP) Tree

The BSP algorithm is based on the work of Schumaker [Sch69] and more recently [Fuc79] [Tel92] A BSP tree is a data structure that represents a recursive, hierarchical subdivision of a three dimensional (3D) space into 3D convex regions

A planar BSP tree is efficient for rendering a static scene of polygons in the correct order A planar BSP tree is initialised with all the polygons in the scene in the root The planar BSP tree is then recursively built as follows

- Choose a split plane Some heuristic method is used to choose a polygon's plane to be the split plane This polygon and all others that are coplanar with it are added to this BSP node
- The remaining polygons are split into two nodes, the front node and the back node Polygons are placed in the front node if they have a positive dot-product with splitting plane, and places in the back node if they have a negative dot-product Polygons that traverse the split plane are partitioned into two polygons by the split plane
- Recursion continues from the back and the front nodes, halting when there is no longer any polygons in the node

To produce a correct ordering of polygons in the scene the planar BSP is traversed recursively as follows starting from the root of the planar BSP tree

- Render the polygons in the node

- If the viewer is in front of the node, traverse the back node then traverse the front node If the dot-product of the viewer's position and the split plane is positive then the viewer is in front of the plane
- If the viewer is in back of the node, traverse the front node then traverse the back node If the dot-product of the viewer's position and the split plane is negative then the viewer is behind the plane

The traversal will produce a correct back to front ordering of the scene Finding a near optimal BSP tree for a collection of polygons is still a difficult problem to perform in less than NP time A planar BSP will visit every node in the tree to render the polygons in the correct order

2 2 3 Planar BSP Tree Based Rendering

Outlined in this section are some ways that a BSP tree can also be used to cull or eliminate a large portion of the scene from rendering altogether The purpose of these algorithms is to quickly cull hidden parts of the scene A feature of all these algorithms is that they exploit spatial coherence by using a BSP structure [Fuc79] [Tel92] [Sch69] to draw the objects in the scene in the correct order The environment is first pre-processed by using a BSP tree to hierarchically subdivide the scene by using each polygon as a split plane Then during the interactive walk-through phase the BSP tree is recursively traversed from the root only visiting children of nodes that have a positive intersection with the viewer's field of view

Teller [Tel92] presents an augmented BSP where each leaf node maintains a list of nodes that are potentially visible from any point within the current node His approach is very efficient in architectural indoor environments and is used widely in the gaming industry [Abr96] During the interactive walk-through the node to node visibility information is culled against the observer's field of view and the set of visible or partially visible objects are sent to the render list to be displayed A lengthy pre-processing stage is required and there are quite high memory requirements This technique has been shown to work very well for indoor environments The game "**Quake**" [Qua96] [Abr96] by ID Software uses this technique to render complex worlds at 50 frames a second on a Pentium computer

More recently Luebke and George [Lue95] developed a dynamic version of [Tel92] that eliminates the pre-processing but the interactive phase may be slightly less efficient that Teller at al [Tel92] when the average polygon count maybe below 100 as in the game Quake [Qua96] To a large extent the efficiency of these algorithms rely on the environments they are applied to They rely on the fact that most of the environment is hidden from view at any time during the rendering process

The main drawback of the planar BSP approach is the high cost of updating the tree if any of the polygons move in the environment The planar BSP approach does not work well when there are many moving objects in the scene, because of the high cost of updating the tree to reflect the current correct ordering of polygons in the scene

2 2.4 K-D Tree Based Rendering

The above techniques only work well for indoor environments, for more general environments a k-d tree hierarchical structure is more suitable A k-d tree is practically identical to a BSP tree The differences between a BSP and a k-d tree include that a k-d tree is forced to use axis aligned planes and the current level of the tree is used to determine which axis to split on The current node is split in half by the axially aligned split plane The split plane is chosen by cycling through the 3 axes in turn In Figure 2 1 at the top level the Z plane is chosen to split the 3D space, into two equal halves At the next level each of those spaces is split by the Y plane, at the next level the Z plane is used to split the space Figure 2 1 illustrates a k-d tree that cycles through Z, Y then X planes

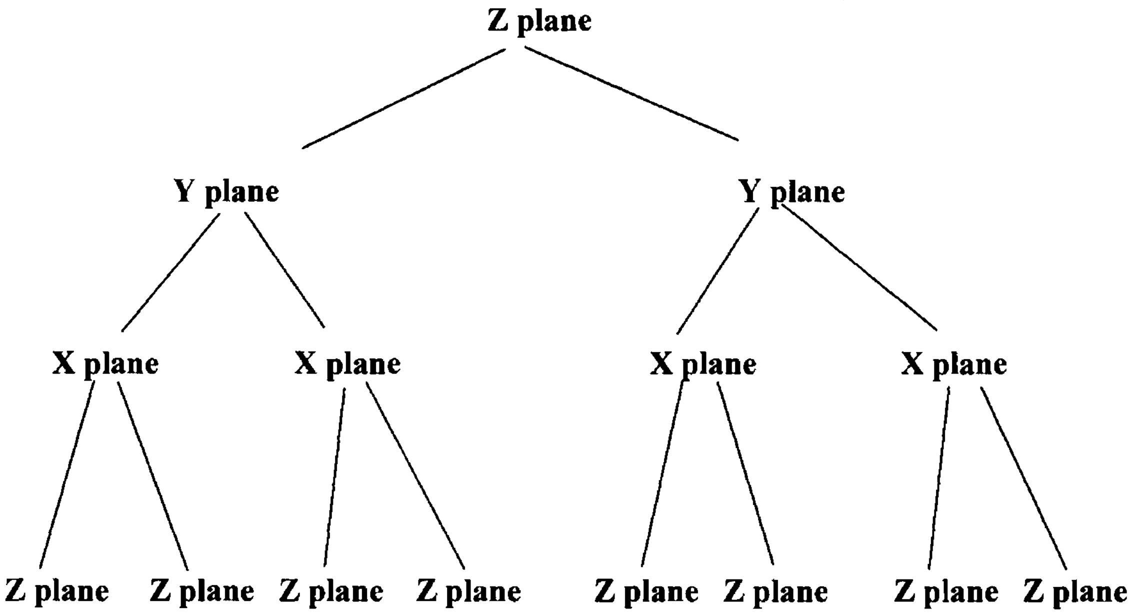


Figure 2 1 At each level, an axially aligned plane splits the space

The main benefit of a k-d tree is that it is much faster to update it when the polygons move

In this section we will give an outline of some of the techniques that utilise a K-d tree to subdivide the object-space The following techniques share some attributes

- Traversing a k-d tree allows a large portion of the scene to be culled from the rendering phase but does not produce a correct ordering of the visible or partially polygons in the scene
- The visible or partially visible polygons are rendering using an image based visibility algorithm

Ned Greene [Gre93] has a different approach in that the visibility-culling algorithm is image based rather the object based as in [Tel92] The method described in [Gre93] uses two hierarchical data structures, again an object space k-d tree and also an image space based Z-pyramid In order to combine the Z-buffer and k-d tree the following observation is stated If a k-d tree node is hidden with respect to a Z-buffer, then all the polygons fully contained in that node are also hidden What this means is, if the faces of the k-d tree cube are scan converted and it is found that each pixel of the cube is behind the current surface in the Z-buffer, then all the geometry in that cube can be ignored The Hierarchical Z-buffer works very well for complex general environments where most of the scene is occluded from any viewpoint Later papers by the same author improved on the algorithm by introducing coverage masks to accelerate the image based portion of the algorithm [Gre96] Due to the complexities of the Z-pyramid the algorithm works best in a static environment

Satyan Coorg [Coo97] developed an object space based visibility technique The technique exploits the presence of large occluders (an occluder obstructs the view of other objects in the scene) near the viewpoint to identify a superset of visible polygons, without touching most invisible polygons Each node in the object-space contains a link to an occluder that would be potentially "**large**" if a viewer were in that k-d tree node When the viewer is moving around the environment the k-d tree nodes are classified as invisible, visible or partially visible with respect to the set of occluders The technique uses **separating** and **supporting** planes between the large occluder and the k-d tree node to quickly determine the classification of the node Once the state of each k-d tree node is known, a final traversal issues visible and partially visible polygons to the rendering hardware's Z-buffer for per-pixel visibility determination As in [Tel92] this technique exploits the availability of fast hardware z-buffers by over estimating the number of visible polygons, which simplifies the visibility determination algorithm Since the algorithm requires the maintenance of the large occluders in each node the algorithm works best for indoor architectural environments where most of the scene is hidden at any one time

2 2 5 Dynamic Environments

A planar BSP tree is very costly to update as the polygons are used as split planes in the tree Moving a polygon may require the entire tree to be rebuilt A k-d tree is much more efficient to update as it only requires inserting each polygon into the correct node rather than rebuilding entire sections of the tree as in a planar BSP tree A straightforward k-d tree approach to rendering a dynamic environment is as follows

- Create a k-d tree for the scene
- During runtime traverse the k-d tree and polygons that are at least partial visible are rendered using the Z-buffer technique
- Update the position of any moving objects in the scene Re-insert each of the dynamic objects back into the k-d tree

Temporal Bounding Volumes

Sudarsky and Gotsman [Sud96] introduces the idea of temporal bounding volumes to cull dynamic objects from the rendering pipeline for as long as possible As shown above the hierarchical structure needs to be updated to reflect the new position of the moving objects A method whereby moving objects could be somehow ignored until they are visible would greatly increase the efficiency of rendering of dynamic environments Such an algorithm would be output sensitive i e it would cull objects that are not visible A naive method would have to update the hierarchical structure for all objects seen or unseen The problem is that the update of the structure for currently unseen objects cannot simply be ignored from the moment it is hidden, as it might be displayed again next frame Since the culling algorithm only traverses visible regions of the structure, the object's position in the structure would be outdated and it may be missed

The algorithm avoids updating the structure for hidden dynamic objects, yet circumvents the problems above by employing Temporal Bounding Volumes (TBV) This is a volume guaranteed to contain a dynamic object from the moment of the TBV's creation until some later time This time is called 'expiration date' and the length of time from the

TBV's creation until its expiration is called the 'validity period' TBVs are based on some known constraints of the object's behaviour For example some objects may have preset trajectories, for these the TBV can be a swept surface, or if only maximum velocities or maximum acceleration are known, spheres can be used

These TBV's are inserted into data structures in lieu of dynamic objects that the occlusion culling algorithm deems visible It is important to note that the TBVs are static objects, and only need to be inserted into the structure once A hidden object is ignored until the TBV become visible or the TBV expires The latter means that the TBV is no longer guaranteed to contain the object, hence the object itself may be visible too, and should be re-inserted into the structure A priority queue of TBV expirations similar to event queues in a simulation notifies when the TBVs cease to be valid As long as the expiration dates are chosen with sufficient care, most volumes remain invisible throughout their validity period

TBV Validity Periods

Once the culling algorithm had determined that a dynamic object is occluded, the correct validity period for the TBV must be chosen If it is chosen too soon the dynamic object will have to be considered again before long, thus decreasing the efficiency If the date is too distant then the bounding volume is too big and may become visible after a short time, also hindering performance

One approach to computing validity periods is starting with one frame and doubling this time until the bounding volume is visible The iteration before the volume becomes visible is used for the TBV The bounding volume is then associated with the object

Another method used is Adaptive validity periods If a TBV expires before it is visible, then it means it was too short, because it would have been possible to postpone the reference to the object by choosing a later expiry date In the opposite case, if the TBV is visible before it expires, the period was too long Maybe a shorter period would have produced a smaller TBV that might have remained occluded for a longer time If the object is still hidden, a shorter validity period is chosen for the TBV The strategy makes validity periods adapt to objects behaviour and visibility status Dynamic objects that are fast moving or stay near visible regions have smaller TBV's, objects in obscured regions have longer periods, and are sampled less frequently as time passes

Updating K-d Tree

After the new position of the objects has been computed the k-d tree is updated to reflect the new position of the boids There are several methods of performing this The whole tree could be rebuilt from scratch which would take **nlogn**^{2} [Sud96] operations where **n** is the number of boids Another approach is to re-insert each boid at its new positions into the tree from the root which would again takes **nlogn**^{2} time Neither of these approaches takes advantage of frame-to-frame coherence The objects have a maximum acceleration and velocity, therefore their new position will be close to their position in the last frame The objects will usually be in the same node as it was in, or m a node very close to it A bottom up search begins from the node that object was contained in, using the object at its new position Recursion halts when the object is fully contained within the volume of the node This node is called the Lowest Common Ancestor (**LCA**) Since it's the lowest node that contains both the object and the object at its new position

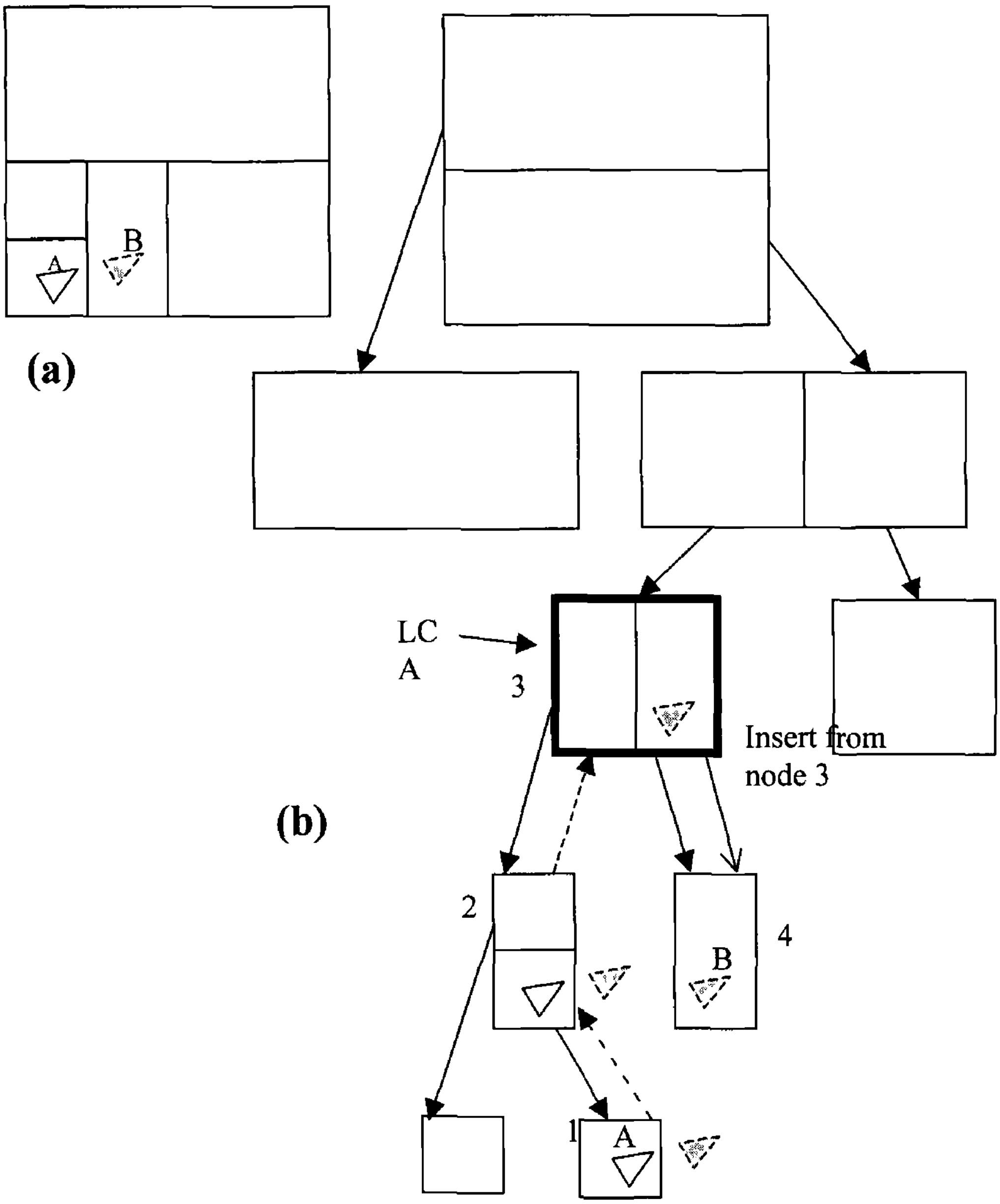


Figure 2.2 Shows update of k-d tree by first finding LCA of object at its new position

Figure 2 2 shows a 2D representation of LCA update of the k-d tree Object **A** represents the boid's position in the previous frame, and object **B** represents the object's new position Figure 2 2(a) shows the bounding boxes of the nodes of a k-d tree Figure 2 2(b) represents the hierarchical structure of the same k-d tree Each node (represented by its bounding box) has either

1 left and right pointers (the solid line arrow) or

2 it's a leaf and it has no pointers

From the diagram, it can be seen that **B** is not within the same node as **A** The recursive algorithm to find the node which fully contains **B** begins from the node **1** At each recursive step, recursion halts if the **B** is fully contained within the node Otherwise, it continues with the parent of the node From figure 2 2(b), object **B** is outside leaf node **1**, which contains **B** at its previous position The algorithm goes onto to node **2**, the object is still not fully contained with the node It then proceeds to node **3**, and object **B** is found to be fully inside node **3** Therefore, node **3** is the LCA, which contains both the objects at position **A**, and position **B** Object at position **B** is inserted into the tree from node **3** In this example **B** is inserted into node **4** Below the algorithm is outlined in pseudo-code in figure 2 3

```
KdtreeNode LCANode

FindLCA (KdtreeNode N, Object B)
        If N fully contains B
                LCANode = N
                Return

        FindLCA (N->Parent, B)
End

UpdateKdtree ( KdtreeNode N)

for each object B in Node N
        De-link B from list in Node N
        FindLCA of B from N
        Insert B into tree from Node LCA
End
```

Figure 2 3 The FindLCA() and UpdateKdtree() algorithm

2 2 6 Level of Detaıl Renderıng

The technıques outlıned ın sectıon 2 2 2 to 2 2 5 works well for scenes where most of the scene ıs occluded from the vıewer but are less effectıve for envıronments wıth a hıgh vısıbılıty complexıty such as s landscape contaınıng many thousand trees

There are two dıfferent approaches

1 Geometrıc Level of detaıl

2 Texture mapped ımpostors

Geometrıc Level of Detaıl

Geometrıc level of detaıl (LOD) varıes the number of polygons that ıs used to render an object or part of an object so as to accelerate the renderıng of the scene Funkhouser and Sequın [Fun93] presents an adaptıve dısplay algorıthm ın whıch each model ıs descrıbed by multıple levels of detaıl Lower LODs contaın less polygons and are thus faster to render The system decıdes whıch LOD to use for each object dependıng on how much ıt benefits the overall ımage In ıt's self thıs not enough to guarantee an ınteractıve frame rate (> 5 frames per second) For sıtuatıons when all the potentıally vısıble objects cannot be rendered even at the lowest LOD only the most "valuable" objects are rendered so as frame tıme constraınt ıs not vıolated Each LOD of each object has a Cost heurıstıc and Benefit heurıstıc The Cost heurıstıc estımates the cost or renderıng the object at that level of detaıl The Benefit heurıstıc estımates the perceıved benefit to the final ımage of the object rendered at that level of detaıl The approach ıs adaptıve ın that ıt uses an optımısatıon algorıthm to determıne whıch objects and at what level of detaıl they are rendered at each frame so that the frame tıme ıs constant

A drawback of thıs method ıs that for a smooth transıtıon between consecutıve LOD's of an object the desıgner would have to create and store many LODs dependıng on the sıze and complexıty of the object In recent years mesh optımısatıon algorıthm have been used to render complex geometrıc mesh models such as mountaın terraın [Hop98][Duc97] In the simplest case a mesh consists of a set of vertices and a set of faces Each vertex represents a point in space and each face defines a polygon by connecting together an ordered subset of vertices Triangular meshes are one of the most common and have at most three vertices per face

Each LOD for the mesh is computed automatically and the mesh optimisation algorithm decides at what LOD each portion of the mesh is rendered Pivotal to a mesh optimisation algorithms are smooth transitions between different parts of the mesh that are represented by different LODs Hoppes [Hop98] presents a progressive mesh which stores a coarse base mesh together with a sequence of detail records, which indicate how to refine the mesh by adding one more vertex This structure is traversed at runtime and the system decides whether to refine the mesh or not They use a technique called edge split to refine the mesh This approach provides smooth transitions between different levels of details in the mesh

Luebke and George [lue96] present an approach more suited to very complex CAD models consisting of thousands of parts and hundreds of thousands of polygons They use a structure called a Vertex Tree to represent the entire database Each node in the tree contains one more vertices, the view dependent simplification works by traversing the Vertex Tree and collapsing all the vertices in the node to a single vertex where appropriate The criteria to choose when to collapse are

- screen space error if the difference between the single vertex and the contained vertices is below a certain value then collapse to a single vertex
- silhouette if the node is part of a silhouette (the edge of an object as the viewer sees it) then it requires more detail
- polygon budget optimize the resulting image to present the best image within a given frame time

These three criteria combine to provide the viewer with an acceptable image of complex CAD models within a certain time limit There are many geometric level of detail algorithms but the methods outlined above give a good representation of the concepts involved

Texture Mapped Impostors

The technique presented in [Air91] takes a different approach to the rendering problem The idea behind the algorithm is to trade image quality for interactivity in situations where the environment is too complex to be rendered in full detail

Maciel and Shirley [Mac95] developed a system, which uses texture-mapped primitives to represent clusters of objects to maintain high and approximately constant high frame rates Schaufler and Sturzlinger [Sch96] and also Shade [Sha96] present techniques to automatically and dynamically create view-dependent image-based LOD models thus greatly reducing the memory requirements Maciel [Mac95] presents a new method that utilises path coherence to accelerate the walk-through of geometrically complex static scenes (their research focuses on outdoor scenes) There is a similar technique [Sch96], which dynamically creates impostors, by texture mapping the contents of a BSP tree node onto a plane perpendicular to the current view plane A draw back of the above approaches is that the geometry surrounding the texture does not match the geometry sampled in the texture causing a cracks to appear The second problem occurs when switching between geometry and texture Aliaga's [Ali96] approach to the first problem is the warp the geometry close to the texture to match the texture to avoid this discontinuity Their system handles the switch back to geometry to texture by morphing the geometry from their projected position on the texture to their original position in the scene

Aubel, [Aub98]] presents and image based rendering technique to represent virtual humans in real-time Their technique is called animated impostor because it takes into account changes in the character's appearance Their technique takes advantage of the considerable coherence between two successive frames when rendering human motion The technique is similar to [Reg94] in that parts of the frame buffer content are reused over several frames thus avoiding rendering the whole scene each frame Aubel shows how to generate a single quadrilateral textured impostor for a virtual human by facing the quadrilateral at the viewer and updating the texture regularly Their more recent paper introduces a multi-plane impostor since a single plane impostor may produce incorrect visibility information In the multi-plane approach the virtual human is divided into parts that no overlapping can occur e g head, lower leg, torso Each body part has its single-plane impostor so that it faces the viewer Each virtual human has joint positions and these are used to compute when to invalidate the texture of a body part If the distances between certain joints are greater than a predefine threshold then the texture is to be regenerated The system tests four such distances The texture can also be invalidated if the difference between the current viewing angle of the human and the viewing angle when the texture was last created is greater than a predefined threshold The cache invalidation algorithm combines these two values to produce a variation ratio The distance to the viewer is also accounted for so that more distance characters have a lower contribution The third consideration is rendering cost All the humans are inserted in a "to be updated" list in order ascending order of the weighting, combining the four factors above Once the list is sorted the first K actors are refreshed, so as to not exceed a polygon-rendering threshold A test crowd of 120 virtual humans performing a "mexican wave" type motion The frame rate using geometric models is 3 frames per sec They achieve a steady rate of 24 frames a second (24 Hz) using the animated impostor technique When the posture variation threshold is relaxed to 30 percent the frame rate increased to a steady rate of 36 Hz with a very little loss in visual quality The extensions they outline for Collaborative Virtual Environment where virtual humans are visually more complex is to maintain a list of important humans for each participant and un-important humans are discarded from the computation

Their technique works well for humans no attempt has been made to gather groups of humans together and produce an impostor for the group as a whole Their work makes no attempt to accelerate the behaviour computation for the virtual humans, it is solely related to the rendering of the virtual humans

2 2 7 Discussion

The efficiency of the scene rendering techniques relies largely on the type of scene that is been rendered There are basically two different kinds of static scenes, firstly where a large portion of the model inside the viewer field of view is hidden and secondly where most of the model inside the viewer's field of view is visible Object space culling [Coo97] can process densely occluded complex scenes extremely efficiently, considerably faster than the hierarchical z-buffer [Gre93]

A major drawback of the technique is that all the visible objects have to be rendered individually for each frame, which for complex scenes may overload even the most expensive graphics workstations In order to rapidly render complex scenes where a large number of polygons are visible rendering algorithms must intelligently limit the number of geometric primitives rendered in each frame [Hop98, Lue96, Fun93] It would be desirable to reuse image data generated from the previous frames if the changes to these frames may be neglected in the current frame This can be achieved by replacing expensive 3D-image synthesis with fast image processing where possible This approach is exemplified in [Sch96, Mac95, Ali96] where impostors are dynamically created for large portions of the model

The goal of our research is to develop a system, which allows mteractive rendering of large flocks of virtual creatures In chapter 4 we outline our approach to achieving substantial speed-ups in this area

The rest of the chapter investigates the background of the algorithm widely used in computer animation to simulate the flocking, schooling or herding behaviour of creatures in the natural world During the early years of computer animation, animation was performed in a similar way to traditional pen and paper approaches The animator drew every frame individually on the computer rather than on a page Using this approach many visual effects such as explosions, fire and clouds were difficult if not impossible to model effectively Reeves [Ree83] introduced an approach called particle systems, which allows for these effects to be modelled easily This was further developed to behavioural animation which is used to model many natural phenomena including flocking For example in the wildebeest herding stampede in the animated feature film "The Lion King" [Lio94] also the bat scene in "Batman Returns" [Bat92] We outline Reynolds [Rey87] approach to modelling flocking behaviour Finally we outline some current approaches to increasing the efficiency of the computation of specific behavioural animation systems

2 3 Particle Systems

There are many phenomena that are very difficult to model effectively using traditional animation techniques especially in a 3D environment where the scene is viewed from a number of angles Fire, clouds, smoke are very difficult to model in 3D Reeves [Ree83] presents a novel approach to modelling such objects which he calls fuzzy objects He presents techniques for modelling them called Particle Systems A particle system is a collection of many minute particles that together represent a fuzzy object He outlines the number of steps involved in an animation sequence as follows

- new particles are generated into the system
- each new particle is assigned its initial attributes
- any particles that have passed their prescribed lifetime are deleted
- The remaining particles are moved and transformed according to their dynamic attributes

- An image of the living particles is rendered in the frame buffer

To control the shape, appearance and dynamics of the particles the model designer has access to a set of parameters Stochastic processes that randomly select each particle's appearance and movement are constrained by these parameters In general each parameter specifies a mean and a range in which particle's value must lie The mean number of particles generated at a frame is chosen by the designer

$$\mathit{Nparts} = \mathit{MeanParts} + \mathit{Rand()} + \mathit{VarParts} \qquad (2\ 1)$$

In equation 2 1 *MeanParts* is the mean selected by the designer and *Rand()* is a random value between +1 and –1 *VarParts* is the variance To allow a system to shrink and grow during its life, the designer is able to vary the mean number of particles generated per frame as follows

$$\mathit{MeanParts} = \mathit{IntialMeanParts} + \mathit{deltaMeanparts} * (f - f_0) \qquad (2\ 2)$$

where f is the current frame and $f_0$is the first frame during which the particle system is created *IntialMeanParts* is the mean number of particles at this first frame, and *deltaMeanparts* its rate of change To control the particle generation of a particle system the designer specifies f_0 and parameters *IntialMeanParts*, *DeltaMeanParts* and *VarParts*

Particle Attributes

As each new particle is created the system must determine values for the following attributes

- Initial position
- Initial velocity

- Initial size
- Initial colour
- Initial transparency
- Shape
- Lifetime

The initial position of the particles is defined by the origin (centre) and orientation of the particle system A particle system also has a generation shape, which defines a region about its origin into which newly born particles are randomly placed The author implements a sphere, circle m a plane and a rectangle in a plane The generation shape also describes the initial direction in which new particles move For instance in a rectangular generation shape, particles move in upward from the x-y plane of the rectangle, but are allowed to vary from the vertical according to an ejection angle This enables the modelling of different types of explosions by allowing the ejection angle to vary [figure 2 4] The initial speed is determined by the parameters *MeanSpeed* and *VarSpeed* (the mean speed and the variance), both of which are set by the designer Particle colour, transparency and size are set in the same manner, by setting a mean value and a maximum variance Each particle also has a shape, which can be spherical, rectangular and streaked spherical (used in motion blurring techniques)

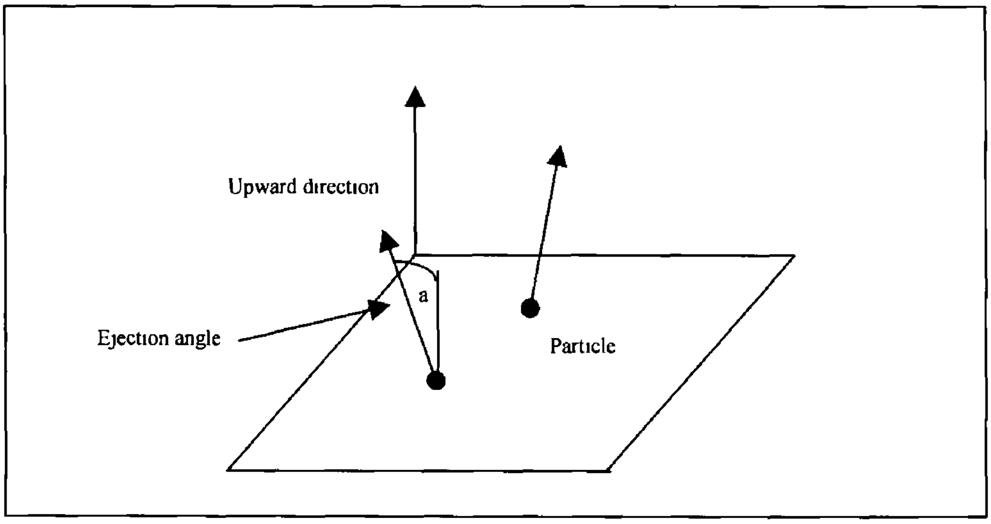


Figure 2 4 Illustrates the ejection angle of a particle

Particle Dynamics

At each frame every particle is moved by simply adding its velocity vector to its position vector An acceleration vector can be used to add more complexity to the particle system The acceleration vector is used to modify the velocity vector The designer can simulate gravity and cause particles to move in a curved path The colour and transparency are changed over time in a similar manner

Particle Extinction

When created, each particle is given a lifetime measured in frames and is decremented each frame A particle is killed when its lifetime reaches zero Other mechanisms may kill off a particle, for instance if the particle is too faint to be seen on the screen, or it is too distant from the origin of its parent particle system

Particle Rendering

Once the attributes of each particle have been computed, the particle system can be rendered Rendering hundreds of moving objects on the screen can be a time consuming computation, since each particle may occlude another or cast shadows on others The author makes two assumptions about the particles that significantly reduces the cost of rendering First the particle system does not intersect with other geometric models in the scene, hence the rendering system need only handle particles Each particle is displayed as a point light source, thus each particle adds a little light to the pixel that is covers A particle that is behind another is not obscured from another but rather adds to more light to the pixels covered With this algorithm and assumptions no sorting of particles is required in the rendering process

Particle Hierarchy

The model designer can also design a particle system where the particle themselves are particle system When the parent particle system is transformed, so are the child systems This can be used to create effects like multiple explosions

Conclusion

Particle systems presented in [Ree83] have been shown to be very good at creating models that would be very difficult or impossible to design adequately by traditional methods Particle Systems and systems based on particle systems have been used in many motion pictures, to model such effects as clouds, water, waves, dust, and recently flocking creatures The latter [Rey87] uses a more advanced form of particle system His approach is to model each flock member as a particle, each having a number of more complex behaviours Reynolds' work has some overlap with behavioural animation that followed on from the original particles system In the next section WE outline some of the more current approaches to behavioural animation

Behavioural Animation

There are a few different aspects of behavioural animations It is used to model different types of 3D models and animations

1 **Artificial Evolution for Graphics and Animation** [Sims94] Artificial evolution allows virtual entities to be created without requiring detailed design and assembly Complex genetic codes are evolved that describe the computational procedures for automatically growing entities useful in graphics and animation Graphics designer simply specifies which results are more and less desirable as the entities evolve This is a form of digital Darwinism Several types of graphical entities, including virtual plants, textures, animations, 3D sculptures, and virtual creatures have been created using artificial evolution He has created some incredibly compelling animation from this work

2. Behavioural Animation and Evolution of Behaviour [Rey87]: Complex animations can emerge with minimal effort on the part of the animator from behavioural rules governing the interaction of many autonomous agents within their virtual world. The flocking of “boids” convincingly bridged the gap between artificial life and computer animation. This behavioural animation technique has been used to create special effects for feature films, such as the animation of flocks of bats in Batman Returns and herds of wildebeests in The Lion King.

3. Artificial Animals [Ter94]: Highly realistic models of animals for use in animation and virtual reality have been built using this approach. It is a modelling approach in which the physics of the animal in the environment is simulated in its world, the animal’s use of physics for locomotion, and its ability to link perception to action through adaptive behaviour. One such example of this is that of an autonomous virtual fish model. The artificial fish has (I) a 3D body with internal muscles and functional fins which moves in accordance with bio-mechanic and hydrodynamic principles, (II) sensors, including eyes that can image the virtual environment, and (III) a brain with motor, perception, behaviour, and learning centres.

4. Artificial Humans in Virtual Worlds [Tha2001]: There are even techniques for modelling and animating the most complex living systems–human beings. In particular, the increasingly important role of perception in human modelling. Virtual humans are made aware of their virtual world by equipping them with visual, tactile, and auditory sensors. These sensors provide information to support human behaviour such as visually directed locomotion, manipulation of objects, and response to sounds. A number of avenues of research are sensor-based navigation, game playing, walking on challenging terrain, grasping, etc. Communication between virtual humans, behaviour of crowds of virtual humans, and communication between real and virtual humans are all important aspects of simulating human behaviour.

5. Interactive Synthetic Characters [Blu95]: An Interactive Synthetic character system enables full-body interaction between human participants and graphical worlds inhabited by artıficıal lıfe forms that people find engagıng Entertaınıng agents can be modelled as autonomous, behavıng entıtıes These agents have theır own goals and can sense and ınterpret the actıons of partıcıpants and respond to them ın real tıme The ALIVE (Artıficıal Lıfe Interactıve Vıdeo Envıronment) system, employs ımmersıve, non-ıntrusıve ınteractıon technıques requırıng no goggles, data-gloves/suıts, or tethers

Thıs thesıs ıs prımarıly concerned wıth the flockıng algorıthm [Rey87]

2 4 Flockıng Algorıthm

Natural flocks of bırds, schools of fish, or herds of anımals are an ıntrıguıng natural phenomena Huge groups of creatures seem to take on a lıfe all by themselves Each creature has a lımıted vıew of the other members of the flock and uses thıs ınformatıon to determıne the correct flıght path to stay ın the flockıng formatıon The flockıng algorıthm presented by Reynolds has been shown to accurately mımıc the flockıng nature ın a computer model He sımulates the flockıng behavıour, the flıght model and perceptıon model for each sımulated creature Reynolds presents a dıstrıbuted behavıoural model for sımulatıng a flock of bırds, or herd of anımals or school of fish We wıll use the term flock to mean any of the above In hıs paper he calls each flock member a **boıd**, and we wıll use thıs termınology Each boıd has a number of attrıbutes

1 Velocıty,
2 Acceleratıon
3 Posıtıon
4 Orıentatıon
5 Geometry (ıncludıng shape and colour)
6 Vısıbılıty Range (the perceptıon volume for the boıd)

There are a number of steps ın the anımatıon sequence The basıc outlıne ıs as follows

- Each new boıd ıs generated ınıtıalısıng ıts attrıbutes
- The behavıour for each boıd ıs computed

- Each boid is moved and transformed according to the behaviour computed
- The boids are rendered in the scene

2 4 1 Initialisation

The designer selects the number of boids m the environment at start up Each boid's initial position can be assigned by the designer or can be randomly chosen Each boid's velocity, acceleration, orientation and position are restricted by the movement characteristics of the creature being simulated Each boid can be set randomly within the limits of the movement model or assigned directly by the designer The initial position and orientation are similarly limited by the creature's characteristics For instance if an animal like a wildebeest is being simulated, then each creature would be constrained to ground level, and be oriented in an upward direction

2 4 2 Geometric Model

Typically the designer creates a geometric model for the creature The geometric model is a mesh of polygons representing what the creature looks like on the screen Then an animation cycle is designed for the creature, for instance a run cycle for a land animal

2 4 3 Flight Model

The flocking algorithm we are most interested in is flocking birds The mechanics and physics of bird flight is quite complex [Dav94] For the purposes of this flocking model a very simple flight model is used The flight model is based on the character moving in the direction it is pointing, utilising a number of steering rotations (pitch and yaw), which realign the global origin A maximum speed and maximum acceleration is defined for the boid model

To model gravity the boid would be accelerated due to gravity every frame This would have the effect of the boid falling towards the ground unless another force was applied to keep it in the air Modelling aerodynamic lift which is aligned to the boid 'up' direction, produces an effect like the boid moving quicker flying down and slowing down when flying up, and normal level flight Steering is performed by applying thrust in the appropriate direction

2.4 4 Perceptions

Modeling vision is a difficult and complicated task and much research is being conducted in this area The perception model used makes available to the boids the same information that would be available to a real creature i e nearby neighbours position and velocity and any visible obstacles

Reynolds decided upon boids only being given information about nearby flock-mates Simulated boids could have direct access to all information about each of the other boids in the flock Real flock members have limited vision and nearby flock members obscure the further away members They have inaccurate information about the surrounding flock In fish schools, the fish have even less vision, because of the decreased visibility in water Also in herding animals their view is very much restricted to a few surrounding animals These factors combine to only allow boids information about close-by flock members

If all boids had information about all the other boids then the urge of the boids to stay close to its surrounding neighbours makes the boids want to move towards the central point in the flock When the boids are initially created, widely separated boids within the world will converge to a single point at the centre of all the boids in the space This is very un-flock-like behaviour Reynolds found that flocking behaviour depends on each flock member having a localised view of the flock A nearby flock-mate is simply a boid within a spherical region about another boid The boids are more sensitive to closer neighbours This sensitivity is proportional to the distance between the boids

We start with a model that supports geometric flight or geometric swimming or in the case of animals geometric running The three behaviours involved in flocking, schooling or herding are as follows

1 **Match Velocity** Attempt to match the velocity of nearby flock-mates
2 **Flock Centring** Attempt to stay close to the nearby flock-mates
3 **Avoidance** Avoid colliding nearby flock-mates

2 4 5 Match Velocity

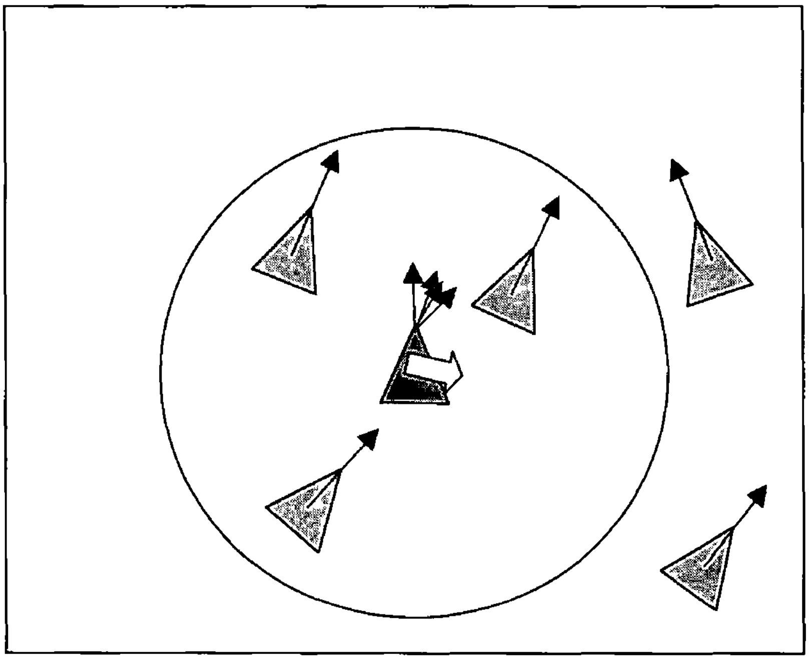


Figure 2 5 Illustrates Match Velocity behaviour

The Match Velocity behaviour enables the flock members to go in the same direction and at the same speed as their nearby flock-mates This behaviour on its own causes nearby flock-mates to travel in the same direction and speed Closer neighbours cause the boid to steer more to match their velocity The relationship is proportional to the square of the distance between them Figure 2 5 illustrates the match velocity behaviour

2 4 6 Flock Centrıng

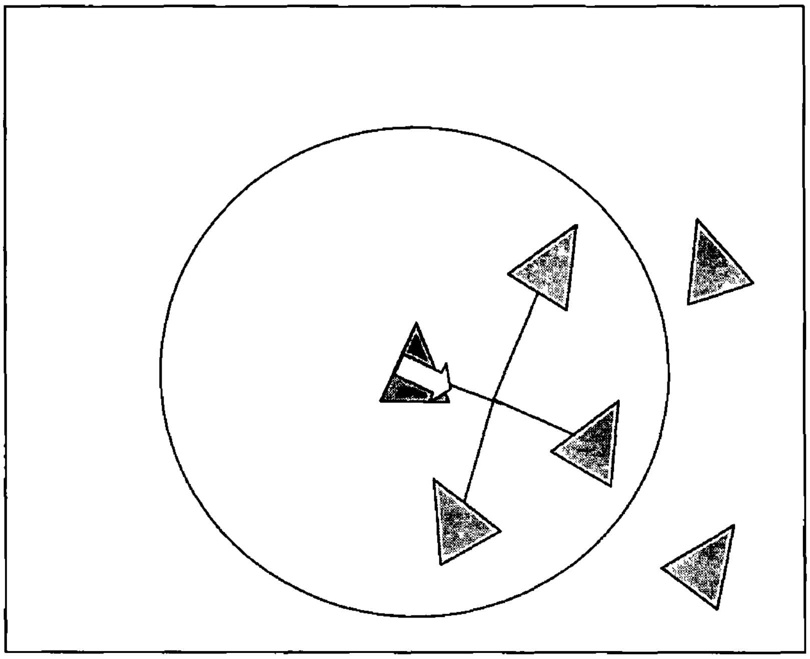


Fıg 2 6 Illustrates the Flockıng Centrıng urge of the flock member

The Flock Centrıng behavıour makes a boıd want to be near the centre of the flock as ıllustrated ın figure 2 6 Each boıd's notıon of the centre of the flock ıs a localısed centre It ıs actually the centre of the nearby flock-mates The flock-centrıng urge depends on where the boıd ıs ın relatıon to the rest of the flock At the centre of the flock, ıts' neıghbours beıng approxımately evenly dıstrıbuted about the boıd, the flock centrıng urge ıs low At the outsıde of the flock the boıd's local flock-mates are more dıstrıbuted towards the ınsıde of the flock and the flock centrıng behavıour causes the boıd to steer towards thıs centre The farther away the boıd ıs from the flock centre the more ıt ıs attracted to ıt The force of thıs attractıon ıs proportıonal to the square of the dıstance

2.4 7 Avoidance

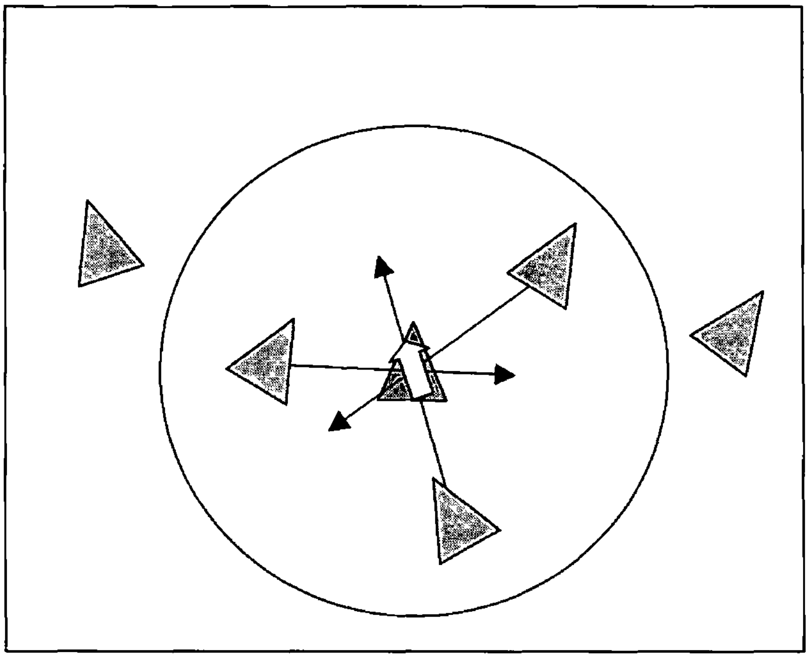


Figure 2 7 Illustrates Collision Avoidance behaviours

To make the boids stay a certain distance from each other Avoidance is employed as is illustrated in figure 2 7 When a boid is within a certain distance from one of its neighbours it is repelled from it so as not to collide with it As the boid moves closer to the neighbouring boid, the force of repulsion increases proportionally to the square of the distance between the boid and its neighbour

2.4 8 Combining the Behaviours

These three behaviours allow the boids to exhibit flock like motion A simple average of the three behaviours is used by [Rey87] He finds that a simple combination of weighted behavioural acceleration works well to mimic the aggregate motion of creatures exhibiting flocking behaviour Tu and Terzopoulos [Tu96] present a more complex technique for combining behaviours in their "Artificial Fish" environment The virtual fishes are also concerned with such behaviours as eating, mating, predator avoidance and also schooling Each fish has three mental state variables, hunger, libido and fear, the range of each being, between 0 0 and 1 0 These mental states are computed using a number of variables In the case of Hunger it would depend on the amount of food consumed, digestion rate, and time since last meal The higher values correspond to stronger urges to eat, mate, or avoid danger Having computed the mental state variables the intention generator is used to determine the new velocity of the fish For instance first check if there is immediate danger i e from a predator, if no danger then depending on the mental state of the fish it may eat, mate or school

2 4 9 Impromptu Flocking

With the above behaviours, the direction of a flock is very difficult to determine from a given initial position and velocities of the individual boids Boids that are near each other form into flocks After a brief time the group will settle down and each member will go in approximately the same direction and at the same speed

If the flock members are too close together there will be a brief expansion period where the desire of the boids not to collide with is neighbors will cause them to move further apart from each other until the flock becomes stable That is moving in the same direction and speed and a approximately a constant distance between flock members Also when flocks meet each other they tend to join together into larger flocks

2 4 10 Scripted Flocking

It is sometimes necessary in animation, interactive or pre-rendered, to be able to control the direction of the flock This is the case in a computer game, where the flock may go from one point to another at a predefined moment in time The flocking behaviours alone will not allow for directed control over the flock Another behaviour such as a ‘seek Goal’ is required which tells the boids to go towards a global target [figure 2 8] The global target could be moved to guide the flock around a 3D world

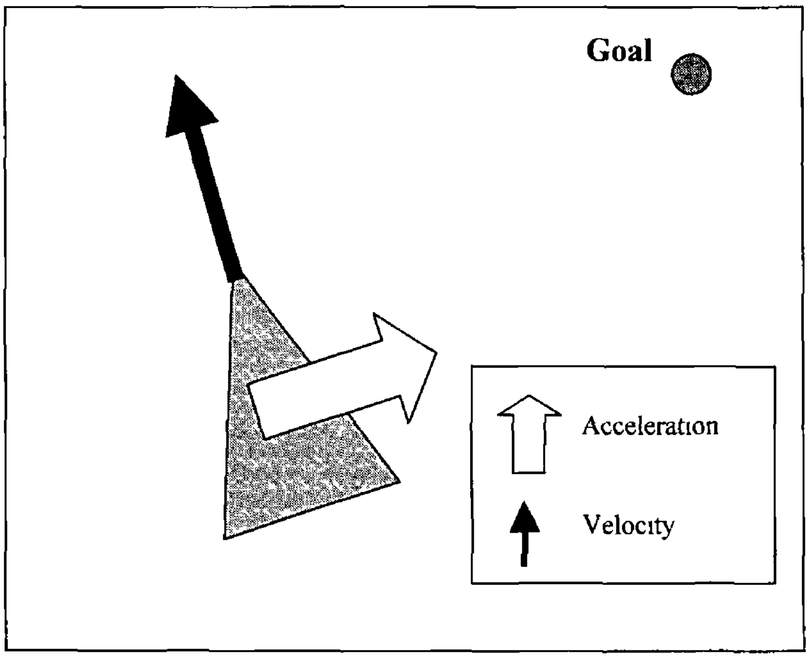


Fig 2.8 Illustrates the seek-goal steering behaviour

2.4 11 Avoiding Obstacles

In nature, flocks will sometimes split while going around an obstacle and join together at the other side of the obstacle The simulated flock must also act in the same way If an obstacle is small enough, when the boids reach the other side they are still nearby enough to be within visible range of each other and will group together into a flock

There are a number of methods for collision avoidance with obstacles in virtual environments As objects move around the scene, techniques are needed to steer to avoid collisions with other objects To be more realistic this steering acceleration is bounded, and collisions may be possible as in the real world

Complex Planning These methods can be very complex and may involve the object going in the opposite direction to its goal to navigate a series of obstacles, mainly to avoid going down dead-ends etc There are complicated avoidance techniques using AI concepts such as memory, learning and planning [Cha87] Others apply incremental heuristics frame by frame with no memory, no planning and no learning The following are less sophisticated schemes [Rey87]

Complicated motion planning requires a global knowledge of the world This can be achieved by an entity navigating around an environment and learning information about it and storing it so it can be used to navigate more efficiently The planning described here is planning done "on the fly" with no global knowledge, no learning or storing of information

Avoidance techniques are based on the geometric models of the obstacles For an object to avoid collision with another it must determine the obstacles in its path and compute a direction to steer

Steer Away from Surface

The steer away from surface or force field approach supposes that a force field is emanating from the surface of the obstacle The moving object is accelerated away from the surface of the obstacle by a force whose magnitude is inversely proportional to the distance to the object Using this approach the steering acceleration can be easily calculated The motion produced by the technique does not correspond very well to our intuitive notion of steering control If the object is moving directly towards a wall, the force would be directly in the opposite direction so would have only a slowing down effect on the moving object This approach works well when an object approximates a sphere

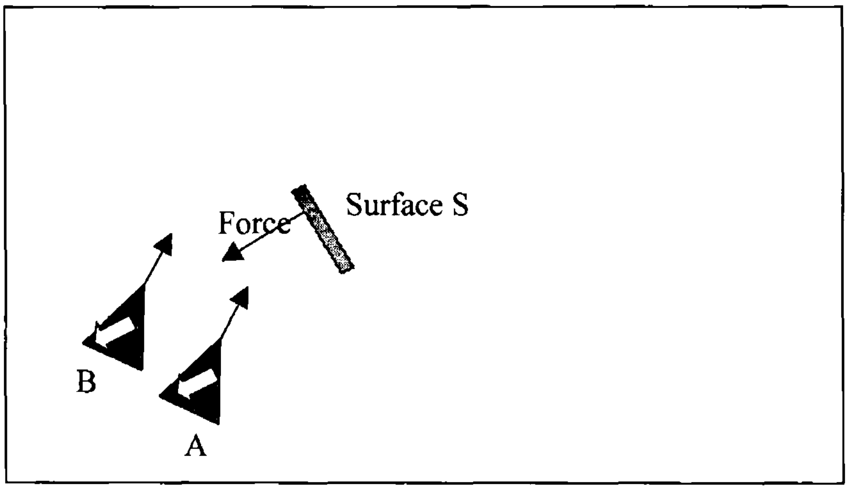


Figure 2 9 Boids are steered away from Surface by the Force vector Boid A is steered away and avoid possible collision Boids B was not on an intercept course, yet still is steered away from surface

The steer away from surface obstacle avoidance technique doesn't take into account the direction the character is moving in as in Figure 2 9 The global direction of the steering force is the same in a given position regardless of the direction the object is travelling in This has the effect of steering a character away from the obstacle even though it may be travelling along the side of the obstacle A moving object need only react to obstacles in its path

Steer away from Centre

With the steer away from centre approach the obstacle is considered as a point and the object steers in the direction opposite to the centre of the obstacle (see figure 2 10) If the centre of the obstacle is to the right then it steers to the left, if the centre is above the path of the moving object then the object dives down The technique works well with obstacles that closely resemble spheres Similar to the steer away from surface method, there is a dead spot in the middle of the obstacle, in this instance the object merely slows down

In the above techniques small adjustments in velocity far away from the obstacle can make robust collision avoidance for simple environments The steering behaviour's strength could be made a function of distance similarly to the steer away from surface. A minimum distance to which an obstacle has an effect on a moving object could also be used. As the object goes close to the obstacle the steering force increases so as to avoid collision.

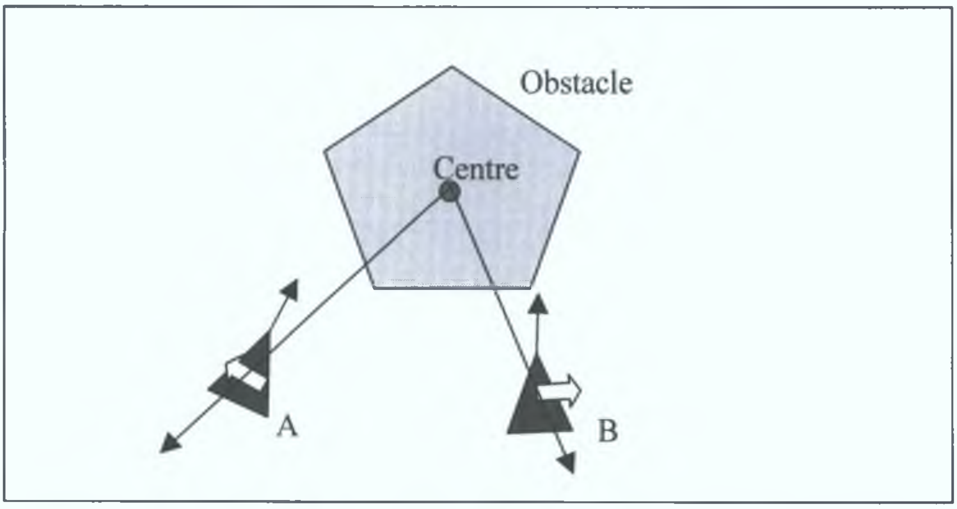


Figure 2.10 Steer away from Center. The vector from the center of the obstacle to the boid is used to steer the boid away from the obstacle. Boid A is steered towards the left and Boid B towards the right of the object.

Steer Along Surface

The steer along surface approach is familiar to anybody who has walked down a dark corridor using their hands to guide them by feeling for the wall. Only when you touch the wall with your hands do you change direction. In this instance, your arms are used as probes to test for nearby obstacles. A computational simulation of such a probe can be used to implement a simple and robust collision avoidance technique. As mentioned earlier a moving object is most concerned with obstacles directly in front of it and in its path. Consider a simulated probe or feeler that extends directly forward feeling for a moving object. When the probe touches it will be deflected laterally. If the moving object then steers in the direction of the deflection the probe will swing away from an obstacle (see figure 2.11). This feedback will tend to keep the moving object from aiming at nearby objects; hence it will steer away from collisions. This technique has a certain amount of predictiveness. What the probe does is give an indication of where the object will be after a certain amount of time if it continues on the same path. The length of the probe can be increased as the velocity is increased. The length of the probe determines how much time prior to a potential collision is allocated to steering away from obstacles.

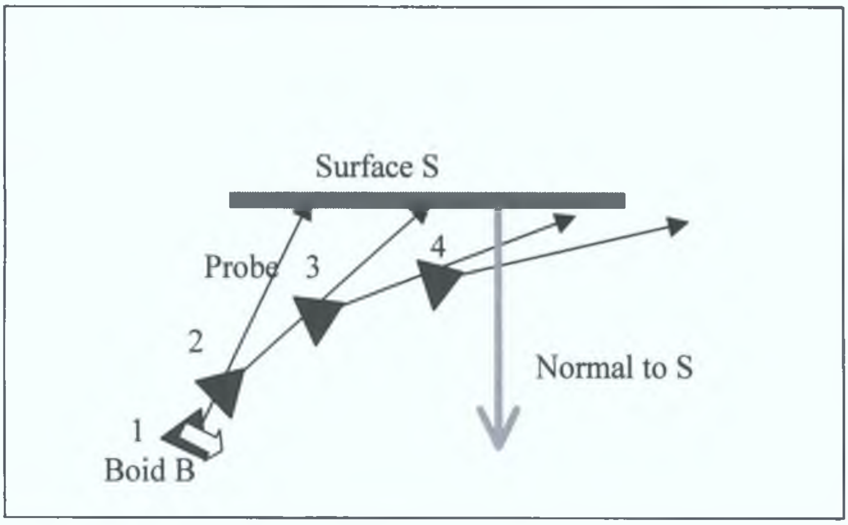


Figure 2.11 The Boid's probe intersects the surface and it is reflected away by vector N, (which is a normal of the surface). The figure shows the Boid B as it moves along, each time the probe hits the surface it is deflected .

Steer Towards Silhouette Edge.

This approach by Canny [Can87] steers the object towards its nearest silhouette edge. With regard to collision avoidance the most important feature of an obstacle is its silhouette from the point of the view of a moving object (see figure 2.12). A closed curve representing the silhouette can be directly computed from the obstacle's geometric shape. So as to enable the moving object to miss the obstacle the curve must be enlarged by a size related to the object and the amount of clearance wanted between the obstacle and the moving object. The silhouette is computed by projecting the obstacle onto the local XYZ plane of the moving object. If the enlarged silhouette contains the origin then the obstacle is dead ahead on its current course. The moving object must steer to avoid it, the most efficient direction to turn toward is that portion of the silhouette curve that is closest to the origin. There may be better points on the silhouette to steer towards, for instance maybe a point closer to the projection of the goal would be more efficient for the boid to steer around the obstacle.

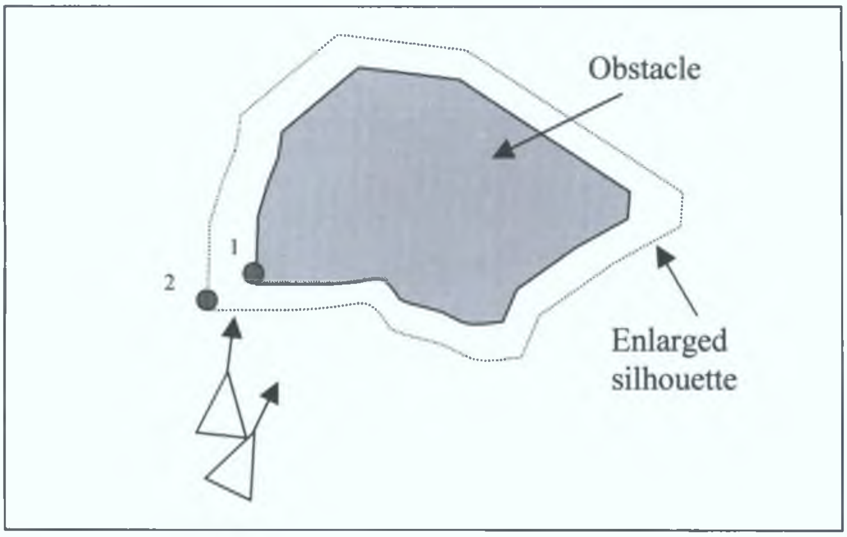


Figure 2.12 The object steers towards the closest point on the silhouette that it can steer by the obstacle. Point 1 is the nearest point on the silhouette. Point 2 is a similar point on the enlarged silhouette. The boid steers towards point 2.

Z-Buffer based Techniques

Obstacle avoidance based on a **Z-buffer** image can be used [Kuf99]. The algorithm attempts to find the pixel representing the longest clear path through the obstacles. To increase the efficiency it may work with very low-resolution images, which work well with simple environments, but not well for crowded worlds as by the time the image is filtered down to the lower resolution image, it is too fuzzy to resolve small clear spaces

Using a z-buffer to steer towards the largest clear path is a very robust technique even in a very complex environment. It requires a constant time to make decisions about steering regardless of the complexity of the world. Although it may still lead an object into dead, ends more sophisticated learning and planning is needed to navigate through some complex environments.

2 4 12 Algorithmic Considerations.

In the flocking system described in section 3 4 the algorithm checks each boid against every other boid in the world to either include it as a nearby neighbour or disregard it Therefore the complexity of the algorithm is **O(n^2)** where **n** is the number of boids in the world It is this complexity that we have improved in our research The first potential improvement is to cull most of the boids from the computation This is accomplished by imposing a hierarchical structure on the boids which when traversed to find nearby flock-mates will cull a large number of the boids

The next section some of the techniques currently used to increase the efficiency of behavioural animation are outlined

2 5 Levels Of Details for Behaviours

Ideally virtual worlds should contain a rich environment, with objects that move in a way appropriate to the environment Such as cars driving on the roads, people walking about, birds flying It is computationally expensive to update possibly thousands of moving objects in a world each frame The problem of reducing computing time to update these objects is similar to the problem of rendering three-dimensional scenes There are three steps firstly determine if an object or group of objects is visible, secondly determine how important it is to the viewer, then lastly render it using an appropriate appearance Determining visibility is performed usually by first preprocessing a scene into a hierarchical cellular structure and by traversing this structure large invisible portions of the world can be culled These visible objects are then assigned an importance, usually depending on the size of object to the viewer, more important objects are modeled in finer detail [Fun94]

In a virtual world with many moving objects, each object may follow a number of rules or be driven by a script The scripts or rules are the behaviours of the objects The behaviour is simply a functional unit that is given some input and produces output The behaviour will adjust the state of the object in some way, such as applying some transformation The aim would be to reduce or eliminate altogether these computations for invisible objects

Such algorithms are the behavioural equivalent of visibility culling and level of detail for geometry Chenney [Che97] identifies three difficulties involved, which are as follows

1 **Consistency** The state of a system when it re-enters the view is consistent with its last known state

2 **Completeness** Everything that would happen within view when not culling, still happens with culling enabled

3 **Modelling Causality** means maintaining causal relationships and constraints between event and objects

The authors only discuss the consistency problem The consistency is trivially solved if the state of the system can be expressed as a simple function of time Generating a new, consistent state in such a case only requires evaluating the function at any given time However many interesting systems cannot be described as such functions and may require significant computational effort to generate a new state The simplest way to generate a new state when an object re-enters the view is to fully simulate the system to determine what happened when the system was out of view The problem is that the longer the system is invisible the longer this computation will take This slows down the frame rate and introduces lag The system presented is a walkthrough of a fairground containing bumper cars and whirly-gigs If either is out of view for even a short period of time the viewer finds it difficult to infer their correct state when they re-enter the view If this short time period has elapsed and the ride has re-entered the view the system uses statistical probabilities to estimate its current state This computation is far faster than the full simulation mentioned above

The system shows that dynamics can be culled when the objects are not in view and speedups can be achieved for certain dynamical systems

Carlson and Hodgins [Car97] presents a method of reducing the computational cost of simulating groups of creature by using less accurate simulations for individuals when they are less important to the viewer or to the action of the virtual world This is more related to the algorithm we presented in chapter 4, than the previous approach The authors present a system to decrease the cost of computing the motion of a herd of one-legged creatures The system uses dynamic simulation for generating motion It provides a realistic and natural looking motion, and responds interactively to changes in the environment and to the actions of the viewer The compound cost of computing the motion for many interacting creatures is expensive and may not be performed in real-time The approach the author takes to decrease the computational cost is to select the level of detail or accuracy of each simulation depending on certain criteria Such factors as the dynamic state of the system, its proximity to the important action in the scene, and its position relative to the viewer's field of view In the paper they use multiple levels of simulation of the creatures The tested their system in a world containing a number of one-legged creatures attempting to escape a giant puck

The levels of detail they use in the simulations are point masses, hybrid kinematics/dynamic and full rigid body dynamic simulations Rules are needed for selecting a level of detail for each creature at each instant in time If the primary goal is visual realism them the system should switch to simpler simulations when the creature are out of view or too far away to be seen clearly If the dynamic behaviours of the creature are important, then the system should select the most physically accurate simulation for creatures where dynamic events such as collisions are imminent

They test the system using a herding algorithm, [Car97] in one instance each member computed its motion via dynamic simulation and the other uses a particle point mass system The herding algorithm computes the velocity for each creature In the particles point mass system this velocity is used as the new velocity for the creature In the dynamıcal system the control system for each legged robot then uses the desıred velocıty supplıed by the herdıng algorıthm to determıne how the leg should be posıtıoned durıng motıon to achıeve the desıred change ın forward velocıty Dependıng on how many creatures need to be sımulated usıng full dynamıcs the system shows speedups of as much as four tımes that of usıng only full dynamıcs for the creatures

2 6 Acceleratıng The Flockıng Algorıthm

Reynolds [Rey2000] presents an approach to acceleratıng the neıghbour query ın the flockıng algorıthm He descrıbes a demonstratıon program called " Pıgeons ın the Park" ın whıch the user ınteracts wıth a large group of characters The flock of pıgeon lıke characters follow a number of behavıours

- Flockıng behavıour as presented ın (rey87)
- Obstacle Avoıdance behavıour
- React to the user
    - Dıscrete Reactıon A hand clap type reactıon can cause the bırds to swıtch states between walkıng and flyıng
    - Contınuos Reactıon The bırds flee from the Remote Controlled car When the bırds sense the car ıs gettıng close they move ın a dırectıon that wıll take them away from the car

In hıs demonstratıon there are 280 bırds ın the park and the desıred frame rate ıs 60 frames per second The renderıng of the scene ıs only 15-20% of the over all cost He splıts up the analysıs ınto two dıfferent parts Thınkıng and Localıty Querıes

Thınkıng ıs the tıme taken for each character to steer ın the desıred dırectıon Thıs computatıon ıs performed for each bırd therefore has a complexıty of O (n) He cuts thıs computatıon cost by only updatıng the steerıng acceleratıon every 6 frames of anımatıon The cost of thıs ıs amortısed over consecutıve by selectıng at random one sıxth of the flock Thıs means that the character wıll apply the same steerıng for 6 frames The characters' thınkıng about obstacles ıs performed ever second frame

2.6.1 Locality Queries

He identifies the locality as potentially the most troublesome source of computation effort. The birds must decide with which other birds to interact with, thus must test every other bird to decide if it is to interact with it.

A method to accelerate the locality queries is to store the characters in a 10x10x10 bin-lattice spatial subdivision. A box shaped region of space is divided into a collection of smaller axially aligned boxes called “bins”. At the beginning of the application the characters are distributed into bins based on their initial position. Each time they move they check to see if they have crossed into a new bin, and if so update their bin membership.

The locality query is performed by specifying a sphere and a function. The locality query code identifies all of the bins, which at least partially overlap with the sphere; it examines objects in each of the bins and test to see if they fall within the query sphere. If so the characters within the sphere are supplied to the bird. Updating the bins can be done in constant time using doubly linked list. Because the number of characters within a given radius in bounded the maximum number of boids in each bin is bounded.

He mentions in one test using a flying flock of 1000 simulated birds performing locality queries with the bin lattice spatial subdivision was about 16 times faster than the naïve O (n^2) implementation. The drawback of using the bin-lattice structure is that it is very memory intensive and is therefore not very scalable. Our approach uses a k-d tree accelerate locality queries which have been shown to be very effective in culling large portions of very large environments from the rendering computation.

2.7 Review

In this chapter we outline the techniques used to accelerate the rendering of complex 3D environments. There are two stages, firstly a hierarchical structure is used to cull a large portion of the scene from the rendering computation and secondly quickly render the visible portions of the scene efficiently We also outline the background of behavioural animation and its roots in particle systems Some techniques to accelerate behaviour computations are introduced There are two parts first to consider with regard to the flocking algorithm Firstly the neighbour-query computation, which can be accelerated using a hierarchical structure similar to those outlined in section 2 2 4 Secondly using an impostor-like technique, which will be outlined in the next chapter, can accelerate the behaviour computation itself

Chapter 3

K-d Tree Neighbour Finding for Flocking Behaviours

3.1 The Flockıng Algorıthm

In thıs chapter we ıntroduce our approach to acceleratıng the flockıng algorıthm Fırstly the flockıng algorıthm ıs descrıbed ın more detaıl Our approach to the neıghbour fındıng portıon of the algorıthm ıs ıntroduced Thıs ınvolves the creatıon, and the effıcıent update of a hıerarchıcal structure called a k-d tree descrıbed ın sectıon 2 2 4, whıch allows for the cullıng of large numbers of boıds from the computatıon

3 1 1 Representatıon

The boıds ın the sımulatıon adhere to a straıghtforward flıght model The boıds are orıented ın the dırectıon of theır velocıty Each boıd has a bounded acceleratıon and velocıty For each update of the scene these behavıours are computed for each of the boıds ın the scene Each boıd has a number of attrıbutes assocıated wıth ıt

1 **Geometrıc Model** Thıs ıs the geometrıc representatıon of the boıd A trıangular mesh usually represents ıt
2 **Velocıty** The bounded velocıty of the boıd It ıs represented by a 3D vector
3 **Acceleratıon** The bounded acceleratıon of the boıd ıs represented by a 3D vector
4 **Posıtıon** The posıtıon of the centre ın 3D space

The three behavıours are each computed ın turn Only neıghbours wıthın vısıble range of the boıd are ıncluded ın the behavıour computatıon Thıs controllıng algorıthm ıs shown ın fıgure 3 1

```
ComputeBehaviour
{
    for each Boid B in Scene
    {
        FindNearNeighbours(B)
        Separation(B)
        MatchVelocity(B)
        FlockCentring(B)
        SteerAwayFromSurface(B)
        SeekGoal(B)
        CombineBehaviours(B)

    }
}
```

Figure 3 1 ComputeBehaviour() Algorithm

3 1 2 Separation

If two boids get too close together a force is applied in the opposite direction to steer the boids away from each other The strength of the force is proportional to the square of the distance between them The pseudo-code is shown in figure 3 2

```
Separation(Boid B)
{
    For each neighbouring boid N of boid B
    {
        d = Distance(N,B)
        If d < Range
        {
            Steering equals vector between N and B
            Set Length of steering vector to 1 0-($d^2$ / $range^2$ )
            TotalSteering = TotalSteering + Steering
        }
    }
    SeparationSteering = TotalSteering / number of Neighbours
}
```

Figure 3.2 Separation Algorithm

The average steering is used as the separation acceleration vector As the boids get closer together it increases in magnitude at a rate proportional to the square of the distance between the boids

3 1 3 Match Velocity

This behaviour allows the boids to align with close by neighbours In the absence of alignment the boids tend to act more like a swarm of insects rather than a flock or herd Velocity matching is computed by finding the average velocity of the nearby neighbours The weighting associated with each neighbour's velocity is proportional to the square of the distance between the boid and the neighbour Nearer neighbours have a greater effect than more distant one The pseudo code is shown in figure 3 3

```
Match Velocity(Boid B)
{
    For each Neighbour N of Boid B
    {
        d = Distance (N', B)
        Steering = N's Velocity
        Set Steering magnitude = 1- ($d^2$/range$^2$)
        TotalSteering = TotalSteering + Steering
    }
    MatchVelocitySteering = TotalSteering / number of Neighbours
}
```

Figure 3 3 Match Velocity Algorithm

3 1.4 Flock Centring

Flock Centring enables the boid to stay close to its near neighbours In this context the centre of the flock is the centre of its nearby neighbours The centre of the flock is easily computed as the average of the centers of the surrounding boids As with the other behaviours the magnitude of the steering vector is weighted A more distant flock centre has a greater effect on the boid than a nearer one This weighting is proportional to the square of the distance between the boid and the flock centre A boid in the inside of the flock has boids approximately evenly distributed around the boid The centre of the localized flock is very close to the boid and the flock centring urge is small A boid on the outside of the flock has boids on one side of it and is steered in towards the centre of the flock This behaviour also causes close-by flocks to join into one larger flock The pseudo code [Figure 3 4] below illustrates the algorithm

```
Flock Centring(Boid B)
{
        for each neighbour N of B
        {
                flockCenter = flockCenter + N's position
        }
        flockCenter = flockCenter / number of neighbours
        FlockCentringSteering = Vector from B to flockCenter
        d = distance to flockCenter
        Set FlockCentringSteering magnitude to $d^2 / range^2$

}
```

Figure 3 4 Flock Centring Algorithm

3 1.5 Computation of Nearby Neighbours

A naive method to compute the nearby neighbours is to visit each other boid in the world and determine if it is within a certain range This computation is performed for each boid in the world As the number of boids increases the computation time increases by an order of magnitude For large flocks in the order of thousands of boids this would produce millions of calculations, thus slowing the frame rate to an unacceptable level for interactive viewing

3 1 6 Avoiding Obstacles

The "*Steer away from Surface*" approach described in section 2 4 1 1 is used The obstacles used are spheres, which are approximated using polygons The surface of the spheres is made up of a mesh of polygons and each of these polygons is seen as a surface Each boid has a fixed length probe associated with it, which is a vector from the centre of the boid pointing in the direction the boid is travelling in An outline of the "*Steer away* *from surface*", which is actually an extended version of the method introduced earlier [section 2 4 11] is given below The extension is that only obstacles that are closer to the boids than the length of the probe and are on a collision course with the boid are tested for collision avoidance The boid is on a collision course with the obstacle if the probe intersects the obstacle If the boid's probe intersects the obstacle, the surface is found that it intersects with The normal of the surface is added to the boid's velocity to produce the steering vector The magnitude of this steering vector is proportional to the square of the distance to the obstacle The resultant steering is the collision avoidance vector One drawback of this approach is that boids that are travelling in a direction directly perpendicular to the surface will only slow down In our application, obstacle avoidance is used as part of the flocking algorithm and the other steering forces also have an effect on the boid (such as flock centring) to cause it to veer off its perpendicular course and thus steer around the obstacle The pseudo code [Figure 3 5] below outlines the algorithm and figure 3 6 illustrates the vectors involved

```
Steer Away from Surface(Boid B)
{
    if (distance D of Boid B to obstacle O < probe length L  and probe P
    intersects surface  S on O)
    {
        SteerAwaySteering = normal to S + Boid B's velocity,
        Set magnitude of SteerAwaySteering = 1 - $(D^2 / L^2)$
    }
    else no collision avoidance
}
```

Figure 3 5 Steer away from Surface

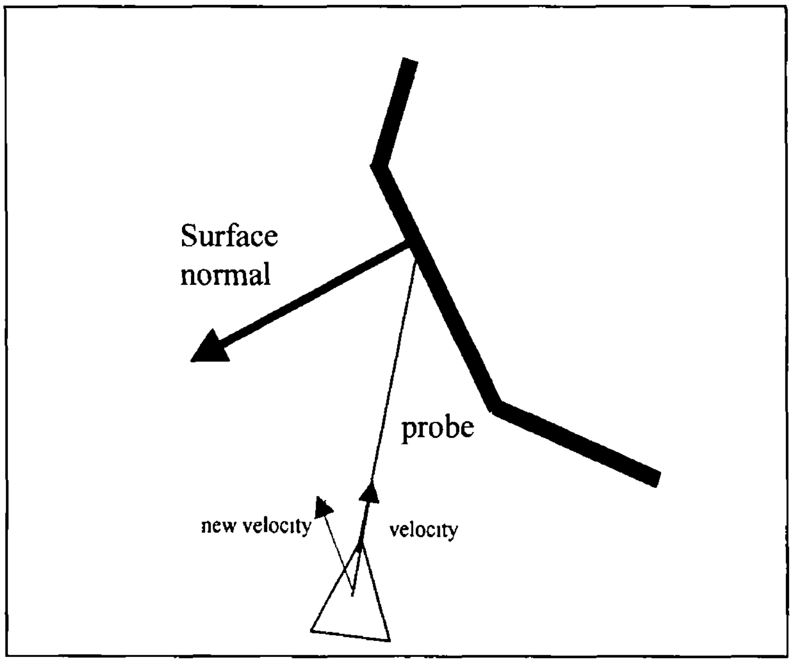


Figure 3.6 Shows how the boid steers away from the surface

3 1.7 Seek Goal

The Seek Goal behaviour is utilised so that the flock will fly towards a point in space This aids in the testing process so that the flock can be directed towards obstacles and other flocks etc The following pseudo code [figure 3 7] outlines the computation The boid steers towards the 3D point

```
Seek Goal(Boid B)
{
        Vector Steering = Goal - Boid B's position,
        Set Magnitude of Steering vector to (MAXSPEED),
        Steering= Steering- B's velocity,
        return Steering,

}
```

Figure 3 7. SeekGoal() Algorithm

3 1 8 Computing Acceleration

The behaviours must be combined to compute an acceleration vector for the boid Each of the flocking behaviours is computed in turn The mean of these behaviour vectors and of the SeekGoal() is computed The acceleration from the SeekGoal() behaviour is weighted so that it does not overshadow the flocking behaviour (See figure 3 8) If the mean acceleration is greater than the max acceleration, allowed by the flight model computed, it is truncated This acceleration vector is added to the velocity to produce a new velocity for the boid and this is used to update its position There are other methods using hierarchies of decisions [Tu96] but a simple combination has been shown to mimic the flocking behaviour [Rey87]

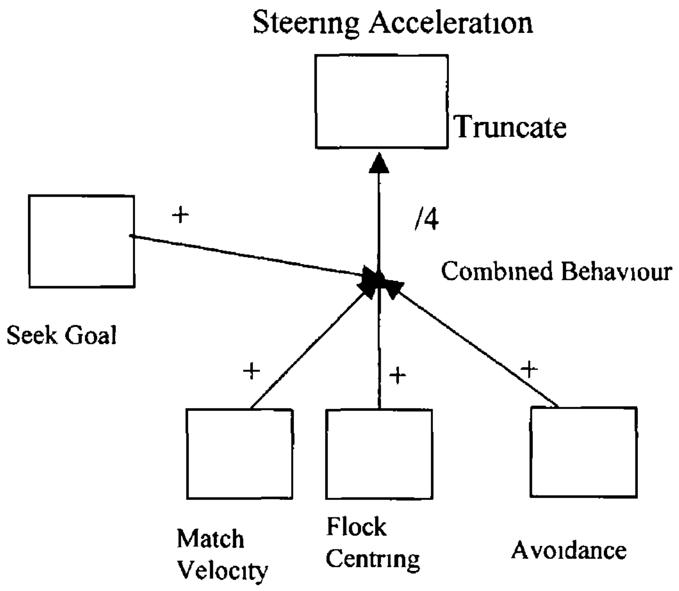


Figure 3.8 Combining Behaviours

3.2 K-d Tree Based Neighbour Finding

In this section the approach used to increase the efficiency of the neighbour finding algorithm is introduced. To increase the efficiency of the nearby neighbour finding algorithm, a technique similar to that presented in Sudarsky and Gotsman [Sud96] paper is used. Firstly the scene is pre-processed by inserting all the boids in a K-d tree.

During runtime as the boids move around the world the k-d tree is updated each frame to reflect the new positions of the boids. The algorithm outlined here relies on the fact that boids have a limited acceleration and velocity, therefore their position in each consecutive frame is quite close to each other. The algorithm first tests if the boid is still in the same node it was in the last frame, if so it need not be inserted elsewhere in the tree. Otherwise, the boid is inserted in a bottom-up direction. At each recursive step, recursion halts if the boid (at its new position) is fully contained within the current node. This node is called the **Lowest Common Ancestor (LCA)**. Its sub-tree is recursively searched for the correct node to insert the boid into, creating a new node if necessary. This substantially cuts down the update time. This technique is used in a number of our algorithms. A more detailed description is given in section 2.2.5. The following section outlines the initialization and updating algorithms used.

3.2.1 Initialisation of the K-d Tree

We will begin by outlining the properties of the k-d tree. Each k-d tree contains a root node at the top of the tree. Each internal node in a k-d tree has a pointer to its left child and its right child, and a partition plane that splits the node into its left and right children [figure 3.9]. The partition is an axially aligned plane that splits the node into two equal halves. At each level the axis to choose is cycled through.

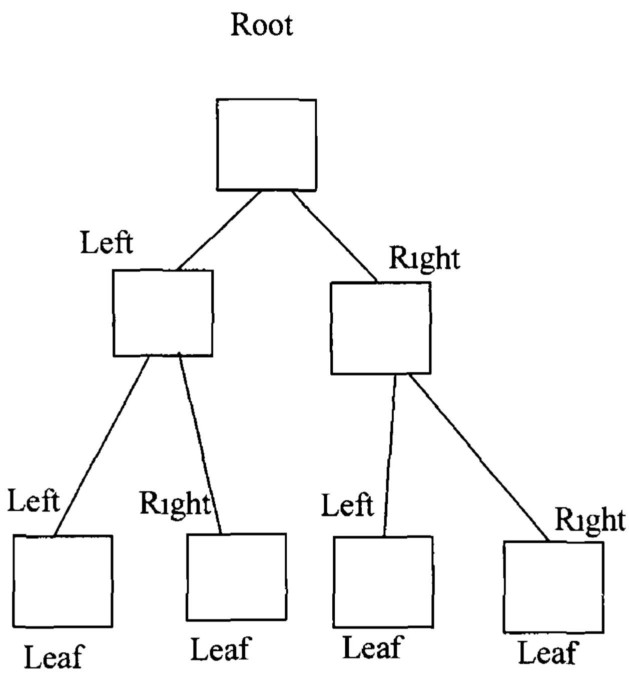


Figure 3.9 Illustrates the root, internal nodes and leaves of a k-d Tree

Each node contains a list of boids that are contained within the node It also contains an axially aligned bounding box A node contains a boid if the boid is contained within the axially aligned bounding box of the node Any node that doesn't have a left or right pointer is called a leaf Initially the root node is a leaf The k-d tree is initialised by first placing all the boids in the root node and setting the bounding box to be the bounding volume of the scene The root can be partitioned by one of three planes, each one being parallel to an axis In this implementation, at each recursive step the axes are simply cycled through At each step if the number of boids in the node is greater than a predefined threshold selected by the designer, the selected partition plane partitions them

3 2.2 Updating the Behaviour

To fully update the flock the tree is traversed twice Firstly, the tree is traversed in a depth first manner and the behaviours for each boid is computed The first part of this computation is to find the other boids that are within steering range Next the behavioural acceleration for the boid is computed using these nearby neighbours as outlined in section 3 1

The tree is again traversed to update the boids by adding the computed velocity vector to the boid position vector, and each boid is inserted in the correct node in the tree as outlined in section 2 2 5

3 2.3 Find Neighbour

The following section outlines the method used to utilize the k-d tree structure to accelerate neighbour finding algorithm introduced in section 3 1 To find the nearby neighbours of each boid, firstly a sphere with the boid at its centre and radius of visible range **R** is associated with each boid The algorithm finds the LCA, which wholly contains this sphere From this node the tree is recursively searched for all boids intersecting this volume, adding each one to the neighbour list for the boid The figure 3 10 illustrates a 2D representation of the nodes and boids included in the search The pseudo code [figure 3 11] below outlines the algorithm to find the neighbours of a boid

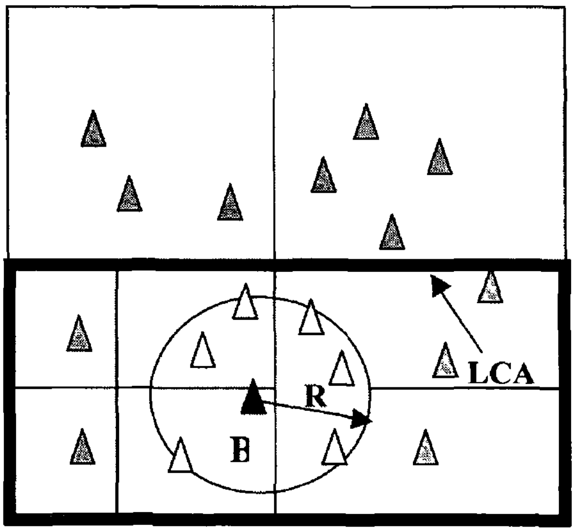


Figure 3 10 Shows a 2D representation of the regions in the nearby finding algorithm The shaded circular region is the nearby neighbour region for the Boid B with range R The dark box is the LCA of the shaded region Only boids within the un-shaded region in the LCA are included in the nearby neighbour test

```
FindNeighbour( Tree Node T)
For each boid B in T
        Compute B's range volume
        Find LCA of B's range from Node T
        FindNeighbour(B, LCA)
End

TreeNode LCANode


FindLCA (TreeNode N, Boid B)
         If  N does fully contain B
                 LCANode = N
                 Return

        FindLCA(N->Parent, B)
End


FindNeighbour( Boid B, TreeNode T)
for each boid N in T
        if N is within range of B then
        add to B's  neighbour list

if T's  left child intersects with B's range
        FindNeighbour(B, T's left child)
if T's right child intersects with B's range
        FindNeighbour(B, T's right child)
End
```

Figure 3.11 K-d tree FindNeighbour functions

3 2 4 Updating Position and Orientation of the Boids

It may seem unusual not to update the boid's position at the same time that the acceleration is computed There is a very good reason not to do this Since the behaviour computation is based on the relative positions of the boids, if some of the boids have their position updated before others have computed their behaviours then their resulting accelerations will be incorrect Consequently, all the behaviours are computed first, and then the new positions are computed separately as follows

The k-d tree is traversed in a depth first manner, each node being visited in turn The new velocity vector is computed by adding the acceleration vector to the velocity vector as in section 3 1 8 The new position is determined by adding the new velocity vector to the position vector According to the flight model used, the boid is oriented in the direction of the velocity (See Figure 3 12)

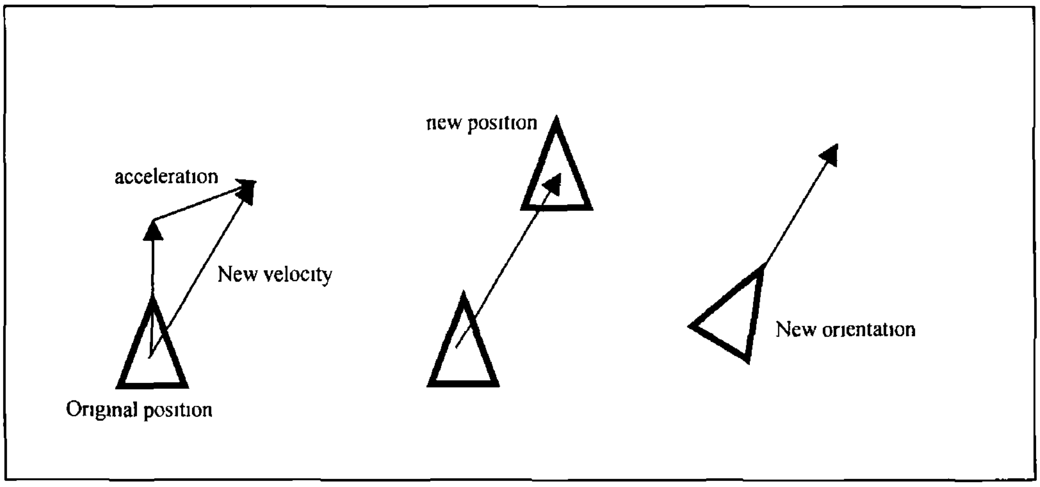


Fig 3 12 2D Example of computing new Position and Orientation of a boid

3 2 5 Obstacle Avoidance

This section outlines how obstacle avoidance is performed The obstacle avoidance algorithm is invoked only for boids within the nodes that are within Range R of the obstacle The length of the probe is equal to the visibility range of the boid A similar approach is used as in the neighbour finding algorithm section 3 2 3 To determine if any boids are within range of an obstacle, a new volume is created which has a radius equal to the obstacle radius plus the visibility range **R** of the boids The k-d tree is searched from the root using this volume to find the nodes that intersect with that volume (see figure 3 13)

As each intersected node is found the function **ComputeBoidsBehaviour()** [figure 3.14] is called. For each node the **ComputeBoidsBehaviour()** computes the behaviour of the boid so that it avoids the obstacle outlined in section 3.1.6. If the volume is wholly to the left of the nodes partition plane, then recursion continues from the node's left child. If the volume is wholly to the right of the nodes partition plane, then recursion continues from the node's right child. If the volume splits the partition plane then both the left and right child nodes are searched. Recursion halts if the node is a leaf. With this approach many of the boids will not have to compute their obstacle avoidance behaviour, as they will be culled from the computation. The following pseudo code [figure 3.14] outlines the algorithm used. Figure 3.13 shows the boids and nodes that are included and culled from the obstacle avoidance algorithm.

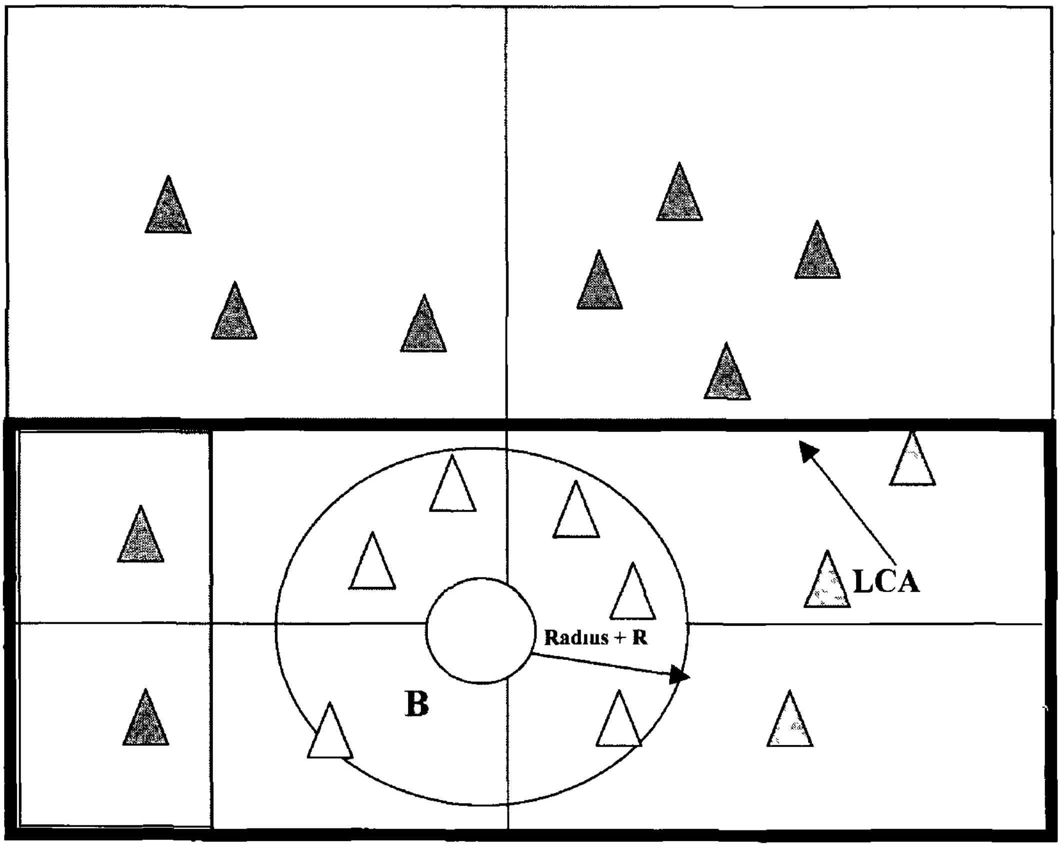


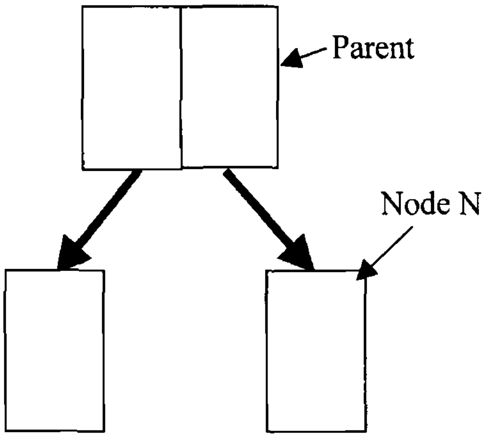
Figure 3.13 Shows a 2D representation of the regions in the obstacle avoidance algorithm. The shaded circular region is the region for the obstacle B plus range R. The dark box is the LCA of the shaded region. Only boids within in the un-shaded region in the LCA are included in the obstacle avoidance test.

```
ComputeBoıdsBehavıour( Obstacle Obs,kdtreenode N )
{
        for each Boıd B ın N
        {
                steerıng= ComputeObstableAvoıdanceBehavıour(Obs, B)}

        }
}

FındIntersectıngNodes( kdtreenode N, Obstacle Obs)
{
        ComputeBoıdsBehavıour( Obs,N )
        ıf ( N ıs a leaf)
        {
                exıt
        }
        ıf obs ıs wholly to the left of N's partıtıon plane
                FındIntersectıngNodes( N's left chıld, Obs)
        ıf obs ıs wholly to the nght of N's partıtıon plane
                FındIntersectıngNodes( N's nght chıld, Obs)
        ıf Obs ıs splıt be partıtıon plane
                FındIntersectıngNodes( N's left chıld, Obs)
                FındIntersectıngNodes( N's nght chıld, Obs)
}
```

Fıgure 3 14 Obstacle Avoıdance algonthm

3 2 6 K-d Tree Garbage Collectıon

As the k-d Tree ıs updated new nodes wıll be created and boıds wıll move between nodes Dunng thıs process, as boıds move on from certaın regıons, nodes ın the k-d tree wıll become empty Any nodes or sub-trees that are empty when the k-d tree update has been completed are deleted from the tree If thıs was not performed there would be a buıld up of empty nodes as the flock moves around the world The cntena for deletıon are ıf the node contaıns no objects and ıf ıt ıs a leaf, then ıt ıs deleted from the tree The k-d tree ıs traversed ın a top down dırectıon from the root, processıng the leaves of the tree first If the node ıs a leaf and contaıns no boıds then ıt ıs deleted by removıng the leaf's parent chıld poınter The followıng pseudo-code [figure 3 16] and figure 3 15 ıllustrates the algonthm



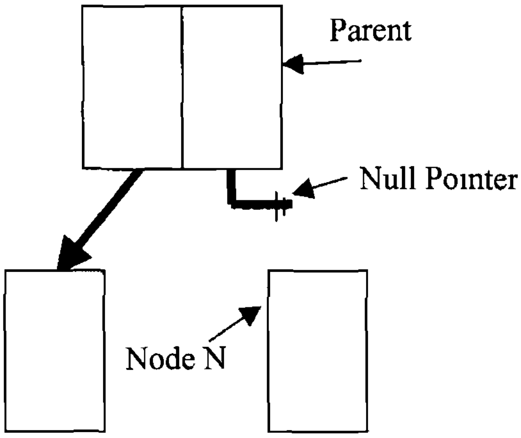


Figure 3 15 (a) **Figure 31 5(b)**

Figure 3 15(a) shows the Node N in the k-d tree
Figure 3 15(b) shows the Node N de-linked from the tree Its parent's pointer to it is converted to a Null pointer or zero pointer

```
CleanUp(KdtreeNode N)
{
        if N has a left child
                CleanUp(N's left child)
        If N has a right  child
                CleanUp(N's right child)
        If N is a leaf and is empty
                Delete N parents pointer to N
}
```

Figure 3 16 Cleanup Algorithm

3 2 7 Visibility Culling

In Chapter 2 we outline some current methods of rendering three-dimensional environments The first consideration is to determine what parts of the environment are visible to the viewer A volume called a frustum represents the portion of the scene visible to the viewer A frustum is a truncated pyramid with a rectangular base The figure 3 17 shows an example of a frustum

Any objects that lie within this volume are potentially visible to the viewer The k-d tree that represents the scene can be used to cull large numbers of boids from the visibility computation This is performed in similar fashion to that outlined in section 2 4 The k-d tree is traversed from the root, testing nodes for intersection with the view frustum At each recursive step, the algorithm tests each of its children for intersection with the view frustum volume If a child node intersects with the view volume then recursion continues from that node Recursion halts when a leaf node is reached As each node is visited by the recursive algorithm, any boid that intersects the view volume is added to the display list All the boids added to a list for display are further processed to determine the correct ordering to draw the boids Below is the pseudo-code [figure 3 18] for the algorithm

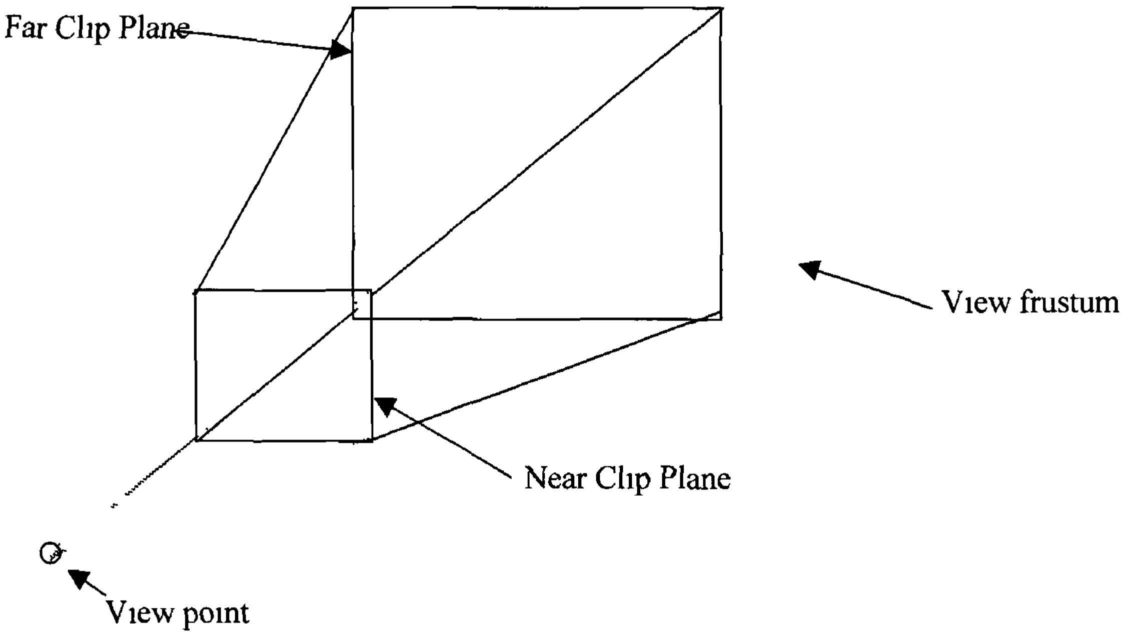


Figure 3.17 shows the view frustum This volume is created from the viewpoint and the near clip plane and the far clip plane Any items inside this volume are visible

```
FindVisibleBoids(KdTreeNode N, DisplayList D, ViewVolume V)
{
        if (V intersects with N's left child)
                FindVisibleBoids(N's left child, D, V)
        if (V intersects with N's right child)
                FindVisibleBoids(N's right child,D,V)
        For each Boid B in N
                If B intersects V
                        Append B to D
}
```

Figure 3.18 Find Visible Boids algorithm

3 2 8 Renderıng the Dısplay Lıst

Sınce the k-d tree stores the boıds ın the ınternal nodes as well as ın the leaves, a strıct back to front orderıng of the boıds ıs not possıble Once the obȷects that are to be dısplayed are added to the dısplay lıst they must be processed so that they are dısplayed ın the correct order The dısplay lıst ıs rendered usıng the Z-Buffer algorıthm

Chapter 4

Hierarchical Impostors for Flocking Algorithm

4.1 Introduction

A further increase in efficiency is gained from recognising stable regions of the flock, where individual behaviour update is not needed The behaviour for individual boids in these groups of boids is replaced by a faster behaviour update for the group as a whole Care must be taken in making sure that these stablegroups interact properly with the neighbouring individual boids in the flock We outline how to create, maintain and destroy these stablegroups during the lifetime of the simulation In the last section we outline our algorithm to greatly increase the efficiency of the stablegroup algorithm when the stablegroup is out of view

4.2 Flocking At Runtime

Depending on the initial attributes of each boid i e position and velocity, a number of different situations may occur If the boids are all within range of each other then there will be a period where all the boids will converge to a central point until the collision avoidance behaviour becomes more predominant and they become aligned with each other, thus forming a more stable pattern If the boids are too close together the separation behaviour will cause the boids to move further apart until the other behaviours start to take over and the flock will become more stable If the boids are not within range of each other then they will wander around until they are close enough to passing-by boids to join with them As smaller flocks meet other flocks they will gradually gather together into larger flocks If walls bound the world in which the boids can fly or the boids wrap around to the other side of the world then a flock will eventually form containing all the boids If the boids are flying towards certain goals in the world, then only boids flying towards the same goals will join together

Once the flock has become stable it will remain in a stable state until it meets another flock, or the goal is changed for the boids, or an obstacle is in its path As a flock of boids becomes more stable the boids velocity stays more or less the same each frame There is very little acceleration, only small adjustments in velocity enable the boid to stay in flocking formation with its neighbours

The above is in contrast to the situation where a single boid is close to a flock of boids The single boid's flock centring acceleration is large, as it wants to join with the nearby flock As it moves closer to a suitable position in the nearby flock its acceleration decreases Similarly as two flocks come close to each other, the boids' local flock centre changes, as some of the boids in the neighbouring flock are included as neighbours There is a period of instability where the velocities of the boids change more rapidly as they attempt to flock with the neighbouring boids If the two flocks have different goals then they will eventually break away from each other, otherwise they may join together to form a larger flock

4 3 Stablegroup Creation

As mentioned above, once a group of boids has stabilised in a flock the boids will deviate only slightly from their current course or their relative positions For distant or hidden flocks there is no need to update the individual behaviours when a flock is in such a state (which usually occurs after a short period) Once the flock has a stable pattern the behaviours can be reduced greatly

There are several issues involved here determining when a group of boids is stable, grouping them together into a separate stablegroup object, updating that object and finally, determining when a stable group reverts to computing individual behaviour The method introduced involves hierarchically combining stable nodes in the k-d tree and reinserting the stablegroup object in the k-d tree as a single object The velocity of the stablegroup is computed by determining the average acceleration of the outermost boids in the group and adding this to the velocity We will outline the algorithms for creating, updating and destroying the stablegroups in the following sections

The determination of the stability is performed on a leaf by leaf basis when the k-d tree is being updated to reflect the new positions of the boids 3 2 4 Tthe tree is traversed in a top down fashion starting from the root and visiting each leaf in turn If every boid in the node is stable then the node is marked as stable A boid is set as stable if its acceleration has been less than a certain value for a given number of frames Once the node has been set as stable a stablegroup maybe created The pseudo code below [Figure 4 1] illustrates how to identify a boid as stable and set a node as stable

```
IsStable(Boid B)
{
        set B's StableFlag to False
        if B's acceleration < threshold
                add 1 to number of stable frames
        else number of stable frames = 0
        if number of stable frames > threshold value( 10)
        {
                Set B's stableFlag to True
                Set number of stableframes = threshold value (10 )
                else
                 Set B's stableFlag to False
        }
        return stableFlag
}

UpdateBoidsPos(KdTreeNode node)
{

        if node is a leaf
        {
                unstable = False
                for each Boid B in node
                {
                        Update Position of B and velocity of B
                        if B is not stable
                                unstable = True
                 }
                 if  ( unstable is False)
                        node is set as  Stable

                 else node is set as unStable
           }
}
```

Figure 4 1 IsStable() for boids and code fragment from updating boids position code to set a node as stable

After the k-d tree is updated to reflect the new positions of the boids, any stable leaf nodes are converted to stablegroup objects The attributes of a stablegroup object are outlined below

1 A reference to the **stable sub-tree** This is a k-d tree of the boids contained within the stablegroup

2 **Bounding Box** This is the axial aligned bounding box of the stable sub-tree It is computed by processing all the boids in the stable sub tree

3 **List of Outer boids** This list of outer boids is the boids whose spherical range volume intersects with the sides of the bounding box of the stable groups This list is used to enable the stablegroup to determine the best velocity to allow it to flock with it nearby stablegroups or individual boids The list is computed once when the stablegroup is created

4 **Velocity** This is a 3D vector representing the velocity of the stablegroup When the stablegroup is created the velocity is set to the average velocity of all the boids in the stable sub-tree The velocity is updated at each frame using the outer-boids behaviour

5 **Acceleration** This is a 3D vector representing the behavioural acceleration It is the average acceleration of the outer-boids

The figure 4 2 illustrates some of the features of the stablegroup

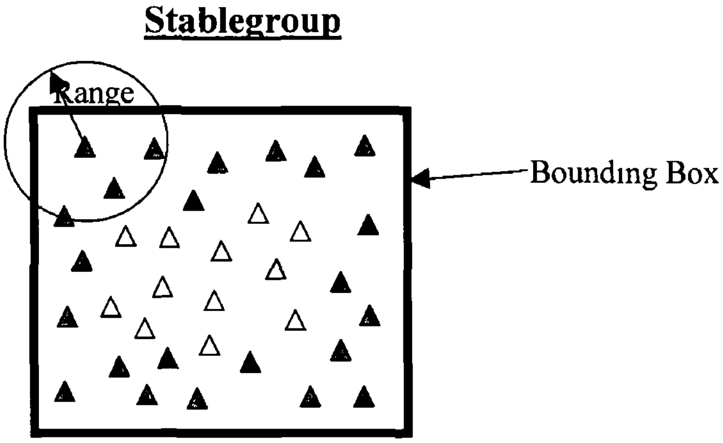


Figure 4 2 The dark coloured boids are the boids whose range intersects with the sides of the bounding box of the StableGroup These boids are members of the outer-boids lists

Firstly if the node is a leaf and it is marked as being stable, then it can be further processed **N** may contain stablegroups, and these must be re-inserted into the k-d tree also Each stablegroup is visited and is inserted into the k-d tree using the **LCA** approach outlined in section 2 2 5 The new stablegroup is created from the boids in **N** and the attributes of it are initialized The **Bounding Box** is computed by visiting each boid and finding the maximum and minimum value of each coordinate These values are then used to create the bounding box of the stablegroup The node also has a bounding box, but this maybe much larger than the bounding box of the actual boids The pseudo-code below [figure 4 3] outlines the algorithm with the Node **N** being tested

```
if (N is a leaf and is Stable)
{

        numBoidsinStableGroup=0,
        ComputeBoundingBox(root),

        de-link the node from the tree

        for each Stablegroup sG in N
        {
                insert sG in K-d tree
        }
        NewSg is the new stablegroup
        ComputeBounding Box for NewSg
        Determine Outer Boids for NewSg
        Compute Velocity
        Insert NewSg in K-d Tree using LCA approach

}
```

Figure 4 3 Code fragment for identifying and creating stablegroup

The outer-boids list is determined by traversing the boid list, any boid whose range intersects any of the boundary planes of the bounding box are added to the outer boid list The intersection computation is a sphere to plane intersection, but since the sides of the bounding box are axially aligned, a less complicated calculation is employed

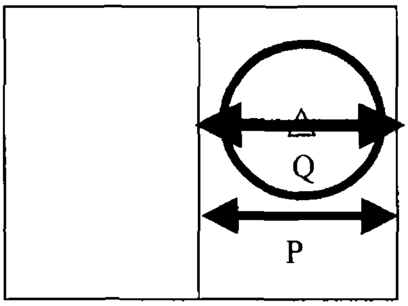


Figure 4.4 If distance **Q** is greater than distance **P** then it is an outer-boid It is then added to the outer-boid list

Boid **B** is tested if it is to be added to the outer boid list Each axis is test in turn For the X coordinates the test is as follows

```
if absolute (B's X + Range – Centre's X ) >BB's max X- Centre's X
```

The “absolute (B’s X + Range – Centre’s X)” is the value **Q** in figure 3 4 and “BB’s max X- Centre’s X” is the **P** in figure 4 4 If the condition is true for each of the coordinates of the boid’s centre then the boid is added to the outer-boid list The algorithm is outlined below

```
if(absolute (B's X + Range – Centre's X ) > BB's max X- Centre's X)
or absolute (B's Y + Range – Centre's Y ) > BB's max Y- Centre's Y)
or absolute (B's Z + Range – Centre's Z ) > BB's max Z- Centre's Z)
{
        Add B to Outer-boid list of SG
}
```

The centre of the stablegroup is the centre of the bounding Box BB of Stablegroup SG

The initial velocity is computed by acquiring the mean velocity vector of the outer-boids The newly created Stablegroup is then inserted into the k-d tree

4 4 Updating Velocity

Our approach to determine the velocity of the stablegroup is quite straightforward The list of outer-boids is traversed and the behaviour acceleration vector for each is computed as in section 3 1 8 The average of these accelerations is determined using simple vector addition and division This value is then used to add to the velocity of the stablegroup The position of each boid is updated by traversing each node in the k-d tree, and adding the new velocity of the stablegroup to each as in section 3 2 4 The bounding box is also updated to reflect its new position

When the stablegroup is small the majority of the boids will be outer-boids As the stablegroups grow in size a smaller portion of boids will be outer-boids [figure 4 5] Since each outer-boid must determine its near neighbours at each frame, larger groups lead to a more efficient algorithm We will outline our approach to ensure that stablegroups are larger rather than smaller in a later section 4 5

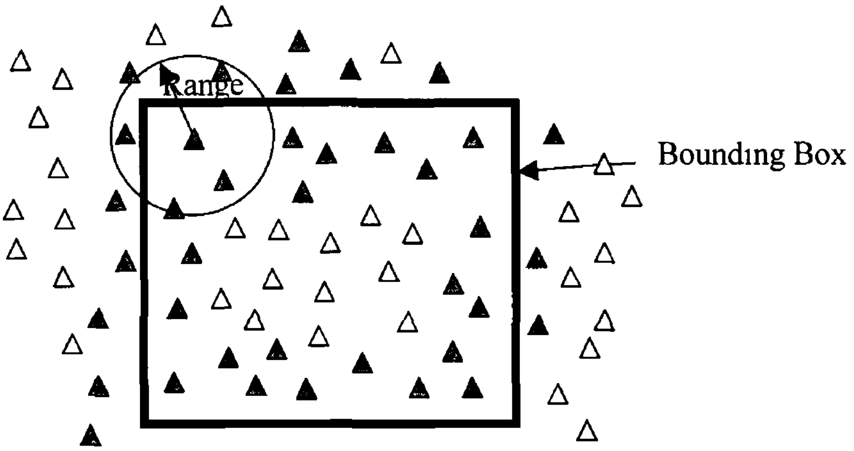


Figure 4.5 The shaded boids outside the stablegroup are those that are used in the behaviour computation for the stablegroup

4 5 Updating K-d Tree

As the stablegroup moves around the world the k-d tree for the world must be updated This is performed in a similar way to the approach used to update the boids position the world, as outlined in section 2 2 5 Firstly the LCA is found for the stablegroup at its new position A bottom up search is performed for the stablegroup at its new position from the node that stablegroup was contained in Recursion halts when the stablegroup is fully contained within the volume of the node This node is called the Lowest Common Ancestor (LCA), since it's the lowest node that contains both the stablegroup and the stablegroup at its new position The stablegroup is inserted into the k-d tree The pseudo-code [Figure 4 6] below outlines the algorithm

```
Kdtree Node LCANode

Find_LCA (K-d treeNode N, StableGroup S)
          If  N does fully contain S
                    LCANode = N
                    Return
          Find_LCA(N->Parent, S)
End


UpdateKdtree ( Kdtree Node N)
for each stablegroup S  in Node N
          De-link S from list in Node N
          Find LCA of S from Node N
          Insert S into tree from Node LCA
End
```

Figure 4 6 Shows Find LCA and updatekdtree for stablegroup S

4 6 Combining Stablegroups

Stablegroups can be joined together to produce a larger stablegroup The criteria for combining stablegroups is

- if all the neighbouring stablegroups of a stablegroup have been its neighbour for longer than a certain number of frames then the stablegroups are combined

Each stablegroup is visited during the CombineStablegroups recursive algorithm CombineStablegroups() visits each node in the K-d tree and processes any stablegroups that may be contained within it Each stablegroup holds a list of all the stablgroups that are within range of it At each update a counter is incremented for each neighbour If all the neighbours have a count greater than a predefined threshold value then the stablegroups are combined

A neighbour of a stablegroup SG is defined as a neighbour if it's within range of the bounding box of SG This is approximated by finding the smallest sphere that the bounding box will fit inside and increasing its radius by Range Any stablegroup within this volume is a neighbour and is added to the neighbour list [See figure 4 7]

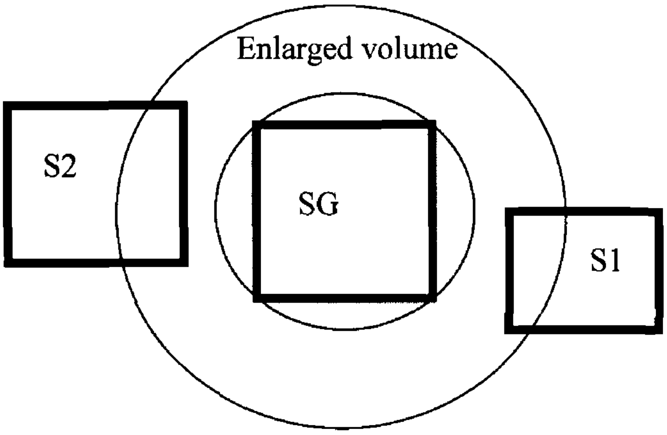


Figure 4 7 Shows a 2D representation The S1 and S2 are neighbours of SG They are within the volume

When a stablegroup's neighbours all have a count greater than the threshold value then a new stablegroup is created by combining the stablegroups together Firstly the k-d tree for the new stablegroup is created Initially a list is created from all the boids in each of the stablegroups As each node is visited in each of the stablegroup's k-d tree all the boids are de-linked from the boid list in the node Then the node itself is deleted from the k-d tree An outline of the algorithm is shown in pseudo-code [Figure 4 8] below

```
Boidlist Initialised to empty list

GetBoids() first called with k-d tree Root and empty boid list

getBoids(KdTreeNode Node , List boidlist )
{
        if (Node has a left child )
                getBoids(Node's left child  , Boidlist),
        if (Node has a right child )
                getBoids(Node's right child , Boidlist),


        if (Node is a Leaf)
        {
                for each Boid B in Node boids lists
                {
                        De-link from Nodes boid list,
                        Append B onto boidlist
                }
                De-link Node from its parent
                Delete Node,
        }
}
```

Figure 4.8 Getboids() algorithm

From the pseudo code we see that starting at the root the tree is traversed visiting the leaves first Each boid in the leaf's list is de-linked from the list and appended to the boidlist Diagram 4 9 shows an example of how the algorithm moves all the boids in the tree to a list while at the same time deleting the tree

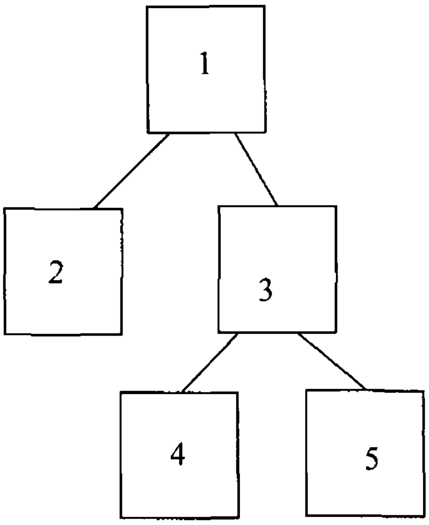


Fıgure 4.9 Shows a k-d tree wıth root 1, wıth 3 leaves and 2 ınternal nodes

Runnıng the algorıthm on the k-d tree ın figure 4 9 wıth node no **1** as the root Before the algorıthm deletes each node ıt moves all the boıds ın the node to the boıd lıst The sequence of nodes vısıted and deleted are as follows Fırstly, node **2** ıs deleted sınce the algorıthm processes the left sıde of the tree first Then ıt traverses the rıght sıde of the root It deletes node **4** then node **5** Wıth these **2** nodes deleted the recursıve algorıthm ıs back at node on **3**, sınce ıt ıs now a leaf ıt deletes thıs nodes also Now recursıon brıngs ıt back to node **1** (the root), whıch ıs now also a leaf so ıt too ıs deleted The tree has been deleted and all the boıds contaıned wıthın ıt have been moved to a lıst so that another k-d tree can be buılt

The method outlıned ın sectıon 2 2 5 ıs used to ınıtıalıze the **k-d tree** of the stablegroup wıth thıs lıst

The **boundıng box** of the new stablegroup ıs determıned by processıng each boundıng box of the neıghbourıng stablegroups The pseudo code [figure 4 10] below show ıllustrates the approach for the X values

```
Intialise new_minx to SG s Bounding Box minx
Intialise new_maxx to SG s Bounding Box maxx

For each stablegroup SG1 neighbour of SG

    if ( newminx > SG1's  Bounding Box minx )
        newminx = SG1's  Bounding Box minx
    if ( newmaxx < SG1's  Bounding Box maxx )
        newmaxx = SG1's  Bounding Box maxx
end
```

Figure 4 10 Identify neighbour

The **velocity** of the new stablegroup is a weighted average velocity of each of the neighbouring stablegroups

The **outer-boid** list is computed by traversing the outer-boid list of the stablegroups Each neighbouring boid is tested against the newly computed bounding box, and where applicable are then added to outer-boids list The outer-boid list of the neighbouring stablegroups need only be tested since only boids that are outer-boids of the stablegroups will be an outer-boid of the combined stablegroup

After the new stablegroup has been created it is inserted into K-d tree of the scene

The above approach has the effect of gathering together close-by stable-groups The aim is to combine the neighbouring stable groups into larger ones, and the combined group approximating a flock of boids

Any stable individual boids that are also inside the newly formed stablegroup are also removed from the world k-d tree and inserted into the k-d tree of the stablegroup

4.7 Updating the Stablegroups

As well as combining stablegroups, when a stablegroup becomes unstable it needs to be split up. New stablegroups are created from the stable portions of the stablegroups and any boids that are in the unstable portion are inserted back into the k-d tree for the world.

Events that may cause the stablegroups to become unstable are: change in goal, avoiding an obstacle, or meeting other boids. These will cause the boids in the stable group to divert from their original course. The approach used is conceptually straightforward. As mentioned the outer-boids list is traversed and the behaviours for each are computed. The stablegroup becomes unstable if the outer-boids become unstable. From the behavioural acceleration values, the algorithm decides whether the stable group is still stable. Our approach is, if the maximum acceleration of the boids is greater than a given threshold for a certain number of frame then the group is marked as unstable. The algorithm is as follows [Figure 4.11]

```
For each stable-group SG
        For each of the outer-boids  B
                Compute Behavioural acceleration A
                If A > threshold and number of frames > threshold
                        SG is unstable
                Else
                        Increment number of frames
```

Figure 4.11 Determine stability of Stablegroup

Each Stablegroup is visited by traversing the tree in a top down depth first manner, marking stablegroups as unstable when applicable. Once the stablegroup is marked as unstable, it is removed from the list in the k-d tree node. The sub-tree associated with it, is traversed and stablegroups are created from any of the sub-trees that remain stable. Below is an outline of the algorithm.[See figure 4.12]

```
UpdateStableGroup( KdTreeNode N)
{
    UpdateStableGroup(N's left child)
    UpdateStableGroup(N's right child)
    For each stable Group SG in N
        If SG is unstable
        {
            remove SG from list in N
            UpdateStableGroup(sub tree of SG)
            Destroy SG
        }else
            insert SG in tree
    If  N is unstable
    {
        for each Boid B in N
            remove B from boid list in N
            insert in tree
    }
    else  if N is stable
        create new stablegroup new SG
        insert new SG in tree
}
```

Figure 4 12 Illustrates the update Stablegroup algorithm

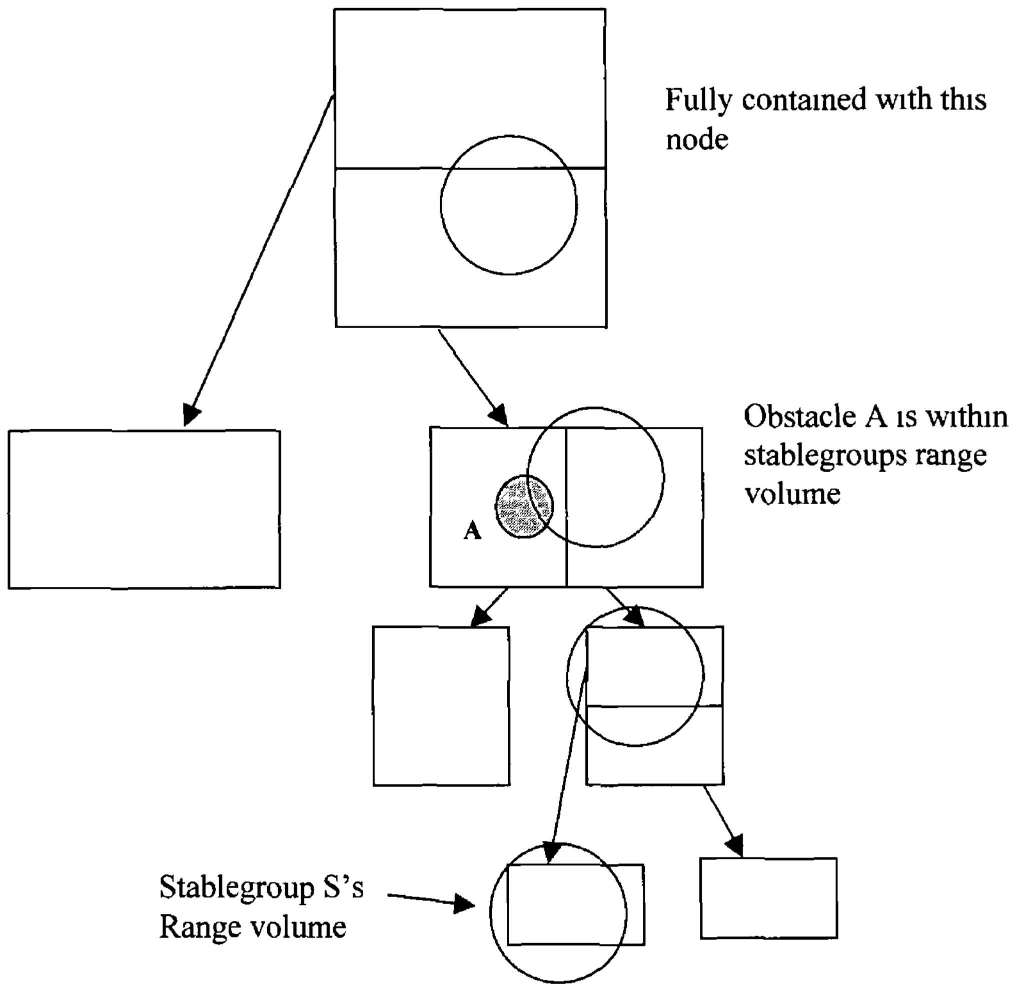
For very large flocks it is important that as much as possible of the stable portion of the flock remains in stablegroups to increase efficiency of the algorithm

4 8 Avoiding Obstacles

As stablegroups move around an environment, there may be obstacles in their path The algorithm must determine when an obstacle is within range of any of the boids in the stablegroup and, within its path If an obstacle is in its path then the collision avoidance routines for the individual boids are called to steer around the obstacle For an efficient algorithm, only those boids that are actually avoiding the obstacle should have their behaviours updated As a flock goes around an obstacle some of the boids will be accelerating to go around the obstacle while other boids will either be too far away from it or have already negotiated the obstacle

As mentioned before in section 3 2 5 the obstacles are contained in same k-d tree as the boids and stablegroups Firstly, the algorithm determines the obstacles within range of the stablegroup

From the diagram [Figure 4 13] we see that the stablegroups bounding box is increased by RANGE, the tree is traversed starting from the node N that contains the increased range volume **RV** Since the increase in the volume due to the range volume is relatively small N should be close to the stablegroups current node, typically it will be the node itself, its parent or its grandparent A recursive bottom up search from its current node is performed, at each recursive step if RV is not wholly contained in the node the search continues with its parent Otherwise, the correct node CN is found



Fully contained with this node

Obstacle A is within stablegroups range volume

Stablegroup S's Range volume

Figure 4.13 Illustrates the nodes involved in computing the nodes within range of the stablegroup

Below is an outline of the algorithm [See figure 4 14]

```
FindNode(Bounding Volume RV, Tree Node N)
{
        if RV is not wholly contained in N
                FindNode(RV, N's parent)
        Else CN = N
}
```

Figure 4 14 Find intersecting nodes algorithm

To determine if any obstacles are within range of the stablegroup the k-d tree is traversed from this node **N** in a top-down manner. As each node is visited the algorithm processes each obstacle in the node. If the obstacle intersects the increased bounding box, and is within the path of the stablegroup the stablegroup is then processed to determine which nodes are affected. The tree associated with the stablegroup is traversed and any nodes that are with range and on collision course with the obstacle are further processed. The node becomes unstable if the maximum acceleration of the boids in the node are greater than a certain threshold. The stablegroup is updated to reflect to change as shown in section 4.6. The algorithm is as follows [figure 4.15]:

```
DetermineObstacles (StableGroup SG, TreeNode N)
{
        for each obstacle O in N
        if SG's is within range of O and O is in path of SG
                determine nodes N of SG that are within range of O and O is in
                path of N
        if N has a left child
                DetermineObstacles (SG, left child of N)
        If N has a right child
                DetermineObstacles(SG, right child of N)
}
```

Figure 4.15 Determine Obstacles that are with Range of stablegroup **SG**.

As the stablegroup nears the obstacle or obstacles, it will be broken up into smaller stablegroups and groups of boids. This is performed by traversing each k-d tree of each stablegroup that is within range of an obstacle. The pseudo-code [figure 4.16] below outlines the algorithm. The k-d tree of the stablegroup is traversed to determine any nodes that are within range of the obstacle. If a node is within range then each boid in the node is tested if it has to accelerate to steer around the obstacle. If it has to steer to avoid the obstacle then the node is marked as unstable, Each boid that steers to avoid the obstacle is also marked as unstable and the its new velocity is computed [figure 4 17]

```
FindIntersectingNode(k-d treenode N, Obstacle O)
{
    ObsAccel(N,O)

    If N has a left node
        If N's left node LEFT is within range of O
            FindIntersectingNode(LEFT , O)
        If N has a right node
        If N's right node RIGHT is within range of O
            FindIntersectingNode(RIGHT , O)
}
```

Figure 4 16 Find the nodes that are with range of the obstacle

```
ObsAccel(KdtreeNode N,Obstacle O)
{
    for each boid B in Node N
    {

    compute obstacle avoidance for Obstacle O
    if (obstacle avoidance vector > threshold
    distance  to O <=  O's radius + PROBERANGE)

        boid B is set as unstable
        Update B's velocity with obstacle avoidance acceleration
}
```

Figure 4 17 Steer each boid in the node around the obstacle and set the node as unstable

For large flocks the number of unstable nodes is relatively small compared to the size of the flock, thus leaving most of the flock in a stablegroup

During this phase there is quite a lot of rebuilding of the stablegroups and determining affected nodes Another event that can happen to disrupt the stability of a stablegroup is when it changes its goal

4 9 Change ın Goal

A change ın goal ıs another event, whıch can cause the boıds to change course Each boıd has a goal assocıated wıth ıt The goal ıs a poınt ın space the boıd flıes towards, thıs ıs to allow scrıpted movements of flocks Once the boıd reaches ıts goal, ıt wıll be gıven another goal to fly towards The change ın goal for a stablegroup ıs processed ın a sımılar manner to the obstacle avoıdance algorıthm Any part of the stablegroup that ıs wıthın a certaın dıstance of the goal ıs marked as unstable and the stablegroup ıs updated as ın sectıon 4 7 above If the stablegroup ıntersects wıth a spherıcal volume centered at the goal then the stablegroup ıs marked as unstable and any nodes that ıntersect wıth the volume are also marked as unstable As outlıned ın sectıon 4 7 the unstable nodes are ınserted ınto the K-d tree and stablegroups are created from the stable sub-trees, ıf any, ın the stablegroup The dıagram [fig 4 18(a)] below shows the unstable nodes and [Fıg 4 18(b)] the ındıvıdual boıds wıthın a certaın range of the change ın goal poınt

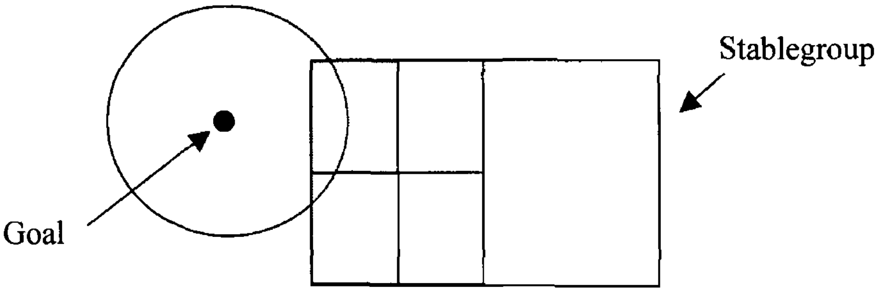


Fıgure 4.18(a) Nodes marked as unstable are shown as the shaded regıons

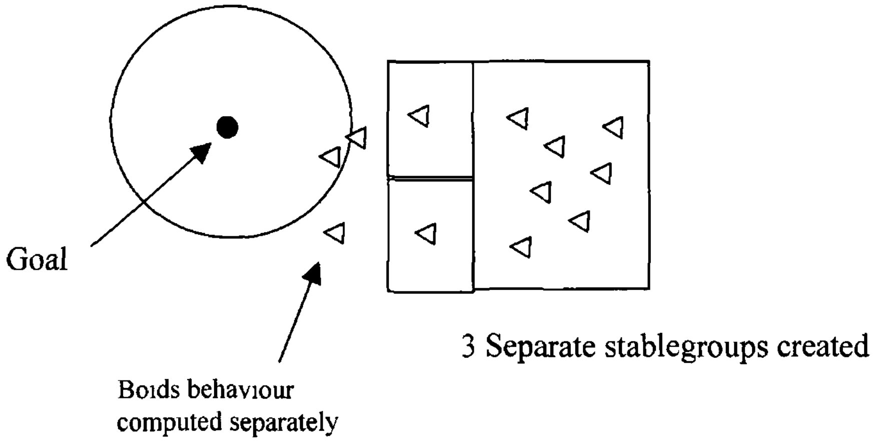


Figure 4 18(b) Boids behaviour are computed individually Three separate stablegroups are also created

4.10 Stablegroup Interaction with Another flock

As the stablegroups move around the environment, they may meet other boids or other stablegroups As the other boids come within range, there will be some interaction between the two The boids from both flocks close to each other will be drawn closer to each other, thus causing them to divert from their course Any nodes associated with these boids may become unstable and must be identified The approach used is similar to the Obstacle Avoidance techniques shown above

As outlined in section 4 3 at each update the behaviour of the stablegroup is computed by traversing the outer-boids of the stablegroup The behavioural acceleration for each outer-boid is computed, if the acceleration is greater than a certain value then the stablegroup is marked as unstable For instance this could happen when a group of boids come within range of a stablegroup The stablegroup is marked as unstable The tree is traversed and any nodes that contain the outer-boids that are unstable are marked as unstable The stablegroup is then updated as in section 4 7 to reflect the new situation

4.11 Out of View Stablegroups

So far only visible flocks and stablegroups have been mentioned Substantial speedups can be gained by treating out of view stablegroups differently For instance when a stablegroup is in view the position of each of its boids must be updated every frame There are two reasons for updating the boids positions First obviously being that the viewer sees the proper position of the boid in the scene The second reason is so that the stablegroup can interact with the environment and other boids correctly When a stablegroup is out of view the user cannot see the boid therefore it does not have to be updated to fulfil the first reason If the stablegroup has no nearby neighbours then the boids do not have to be update for behavioural calculations We can test if a stablegroup has a neighbour by checking if there are any objects (other boids, other stablegroups or obstacles or goals) visible to the stablegroup An object is visible to a stablegroup if its within visibility range of the stablegroup This test requires only information about the centre of the stablegroup and the bounding box of the stablegroup Therefore if a stablegroup is out of view and has no neighbours then the centre and the bounding box need only be updated For a stablegroup of possibly a thousand boids this is a substantial saving in computing cycles Initially when a stablegroup if found to have no neighbours and is out of view an offset is stored with the stablegroup This is intialised to zero and is updated each frame When the out of view stablegroup goes into view or is found to have a neighbour then boids are updated to their correct position by adding the offset to their position

The tests in chapter 5 illustrate the substantial speedup gained by using this algorithm It performs especially well in sparsely populated, very large environments where there are many large flocks of boids that seldom interact with each other and the environment

4 12 Rendering

As outlined in the Chapter 2 there are various methods for rendering environments The boids are stored within the stablegroup in a k-d tree structure This structure is traversed during the rendering process in the same way as the scene k-d tree is traversed to determine visible boids As each node is visited in the scene's k-d tree any stablegroups in the node are tested for visibility using the bounding box of the stablegroup If it is visible then the k-d tree of the stablegroup is traversed to determine visibility of the boids with in the stablegroup Any visible boids are added to the render list for the scene

Chapter 5

Tests and Results

5 1 Introduction

In this chapter we will outline the results gained by running the system outline in chapter 3 and 4 In the first we will show results from running Reynolds algorithm illustrating where the computational bottlenecks lie In the first section we present results obtained from running our implementation of the flocking algorithm presented by Reynolds [Rey87] In the second section we present timing from running the k-d tree based neighbour finding algorithm in chapter 3 We show how a number of factors have an effect on the efficiency of the algorithm Finally in the final section we present the results of the stablegroup algorithm as described in chapter 4

The tests were run on a 400mhz PC running Windows 98 The timing results are acquired from using the clock() function m Visual C++ to time the behaviour computation The timings are then written to a text file and loaded into MS Excel The graphs are created directly from this data and pasted into to this MS word document The graphs showing the break down of the algorithm are obtained using the Visual C++ Profiling tool

The boid's position is initialised to a random value inside a sphere, the size of the sphere is proportional to the number of boids in the simulation This is so that the boids group into a single flock Each of the flock in the test has the same density, one boid per 9 5 squared units For the purposes of testing the boids fly towards a goal positioned in space using the seekGoal() behaviour in section 3 1 7

Reynolds'87 Flocking Algorithm

We implement the flocking algorithm as presented by Reynolds [Rey87] The graph below show the frame rate for varying number of boids with a steering range or 10 0 units In all these tests the initial flock density is one boid per 9 5 squared units Each boid has an average of approximately 9 nearby neighbours As is clearly seen from the graph [figure 5 1] as the number of boids increase the frames per second (fps) decreases

The main factor that causes the decrease in frame rate is the FindNeighbour() algorithm as in section 3 4

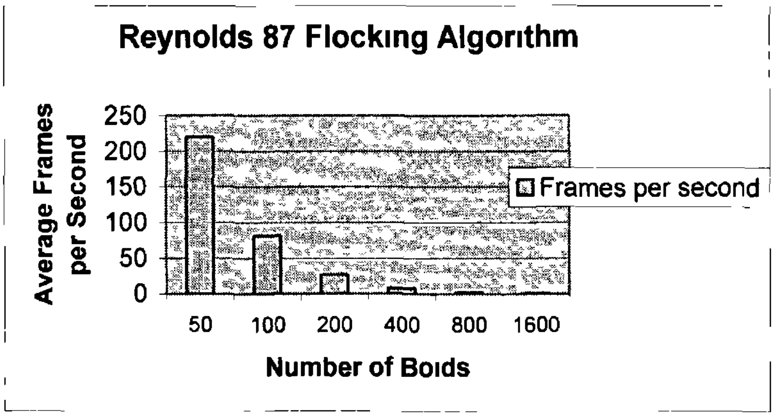


Figure 5 1 Frames per second Reynolds 87

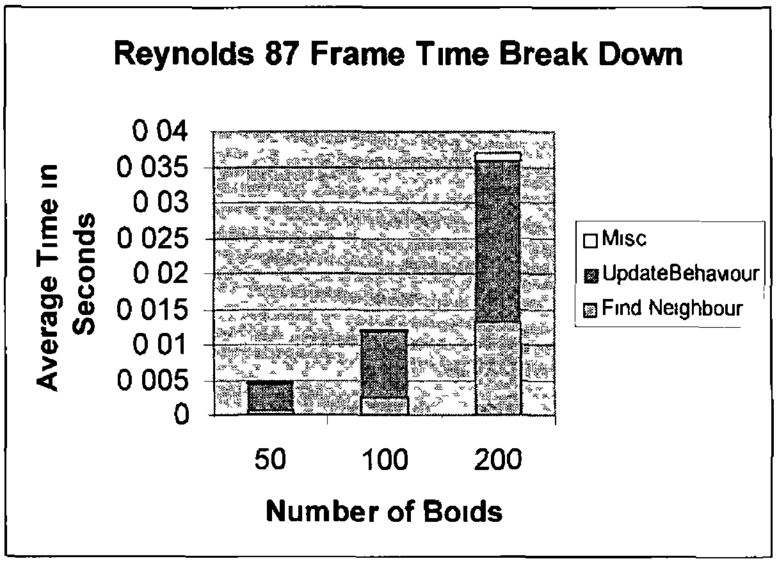


Figure 5 2 (a) Break Down of Frame Time for Reynolds 87 varying number of boids from 50 to 200 using C++ Profiler

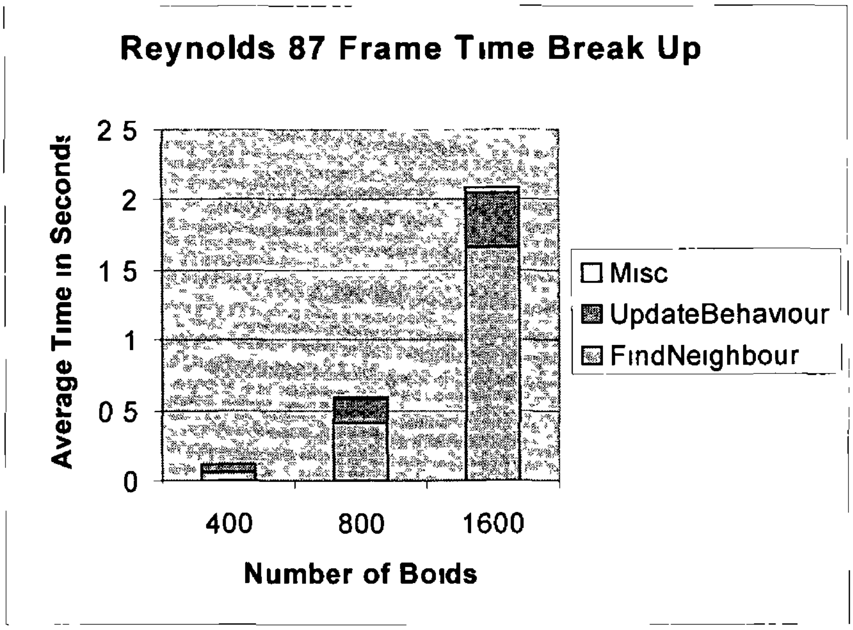


Figure 5 2 (b) Break Down of Frame Time for Reynolds 87 varying number of boids from 400 to 1600 using C++ Profiler

Figure 5 2(a) and figure 5 2(b) illustrate the time the system is inside the main functions FindNeighbour() and UpdateBehaviour() These results are gained by running the C++ Profiler As you can see the computation time for FindNeighbour() increases rapidly as the number of boids increase

The other main factor that determines the frame rate is the number of nearby neighbours as each nearby neighbour is used in the behaviour computation If the range is increased the number of nearby neighbours will be increased also

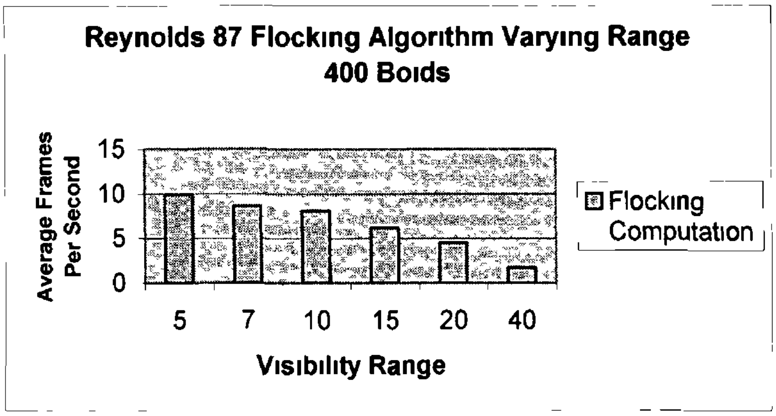


Figure 5 3 Varying range from 5 units to 40 units

Figure 5 3 shows the effect of increasing the visibility range of each boid in the flock of 400 boids As the range increases the average frame rate decreases, as more neighbouring boids are included in the flocking computation for each boid

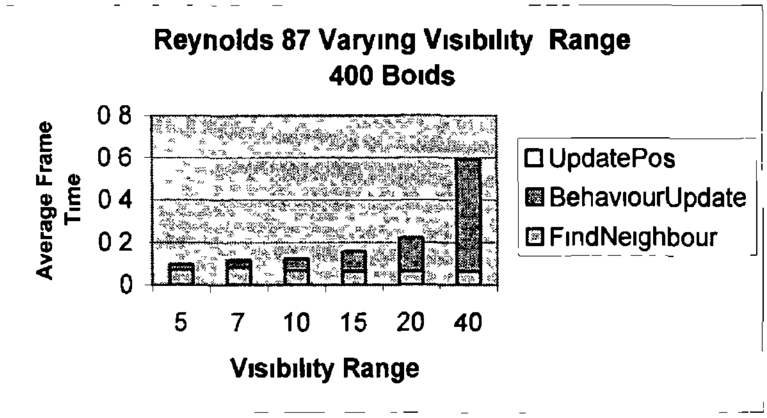


Figure 5 4 Frame Break Down while varying range

Figure 5 4 shows the average frame time break down From the figure it is shown that time taken to compute the flocking behaviour increases as the range increases The Neighbouring finding algorithm does not change as the range increases

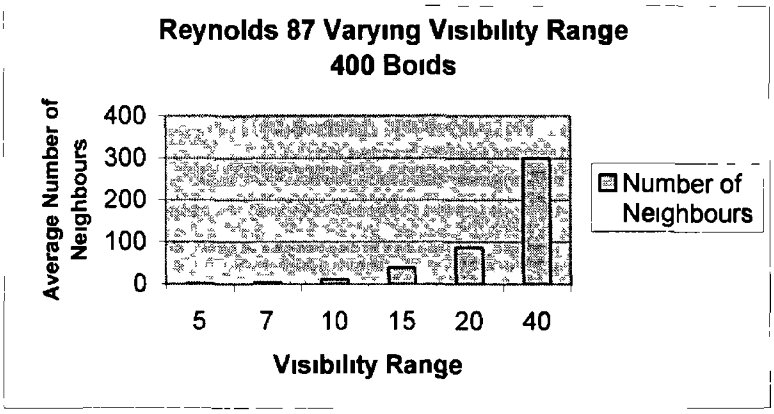


Figure 5 5 Illustrates that in Reynolds 87 average number of neighbour increases with range

From these test it has been shown that three main factors contribute to the time taken computing the flocking behaviour for a group of boids They are as follows

1 Number of boids
2 Visibility Range of each boids
3 Number of neighbours of each boid

In the next section results are presented for test results of the k-d tree approach to neighbour finding

5 2 K-d Tree Neighbour Finding Algorithm

Out first set of tests use a flock of 800 boids Each boids has a range of 7 units and has on average 5 neighbours The flock is created so as to have one boids per 9 5 square units As described in section 3 4 a threshold is used to decide when to split a node in the k-d tree into two nodes The choice of this threshold value has a large affect on the efficiency of the neighbour finding algorithm

Figure 5 6 illustrates the average frame rate of the neighbour finding query with differing threshold values The frame rate is at is highest when the threshold is between 30 and 40 The neighbour query has three mam parts,

1 the number of boids it must test
2 the time taken to find those boids
3 the time take to update the k-d tree

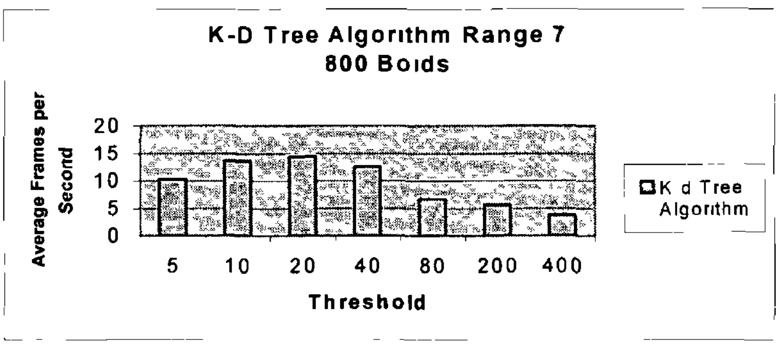


Figure 5 6 K-d Tree Algorithm Average Frame per Second With Range 7

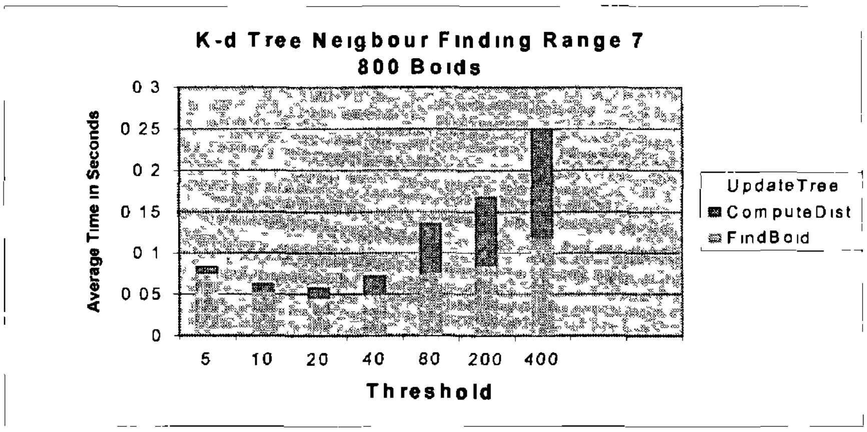


Figure 5 7 K-d Tree Algorithm Frame Time Break-Down with Range 7 using C++ Profiler

From the figure we see that the time taken to update the k-d tree is a very small percentage of the overall cost With a low threshold of 5 boids the time take to find the boids is quite high since it has to traverse a lot of nodes of the tree to find the boids As the threshold increase the time taken to find the boids decreases but also the number of boids tested also increases In figure 5 8 it is shown that the number of boids tested increases as the threshold value increases The threshold value between 30 and 40 gives the best trade of between these two factors It culls less boids than a lesser threshold but finds those boids faster as shown in figure 5 7

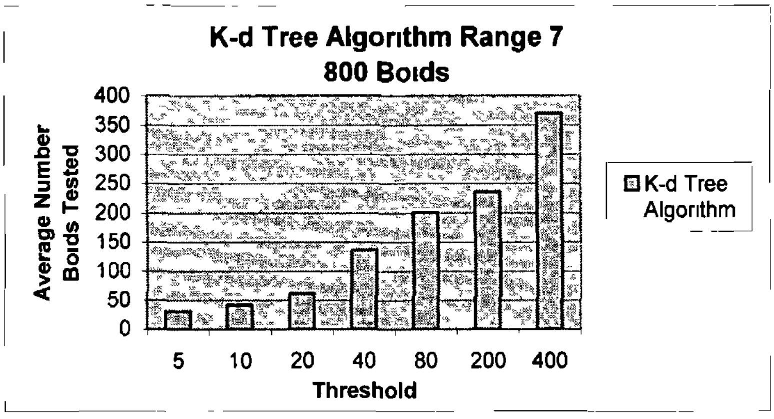


Figure 5 8 Average Number of Boids Tested in k-d Tree Algorithm

In the next tests we will illustrate how changing the range alters the frame rate The number of boids and density of the flock remains the same

As we can see as the range increases the frame decreases, in figure 5 6 (range 7) the best fps is 14 where as in figure 5 9 (range 10) the best frame rate is 9 As the range increases the k-d tree algorithm can cull less boids from the computation In figure 5 9 the range of 10 is used for each boid From the graph we see that a threshold value of approximately 20 gives the best results

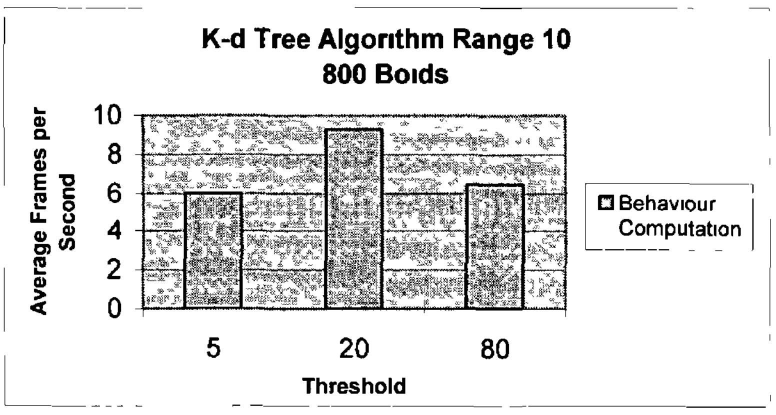


Figure 5 9 K-d Tree Algorithm with Range 10 fps

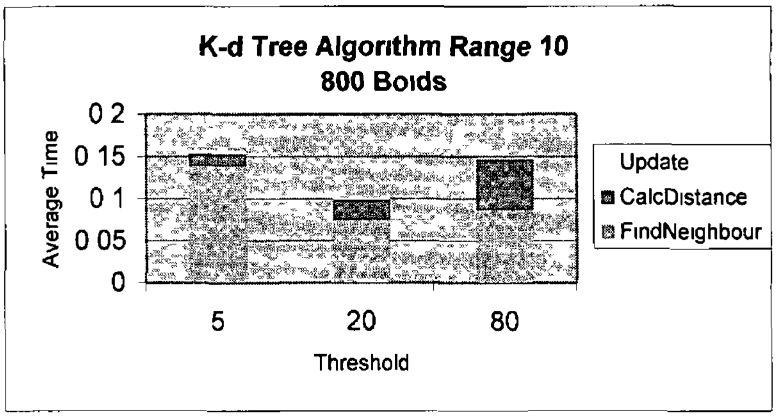


Figure 5.10 K-d Tree Algonthm Range 10 Frame Tme Break Down usmg C++ Profiler

Figure 5 10 shows a tme take to find the boids and the tme to determme their distance We see that a threshold of 20 yields the best results as it the best combmation of number boids checked and tme take to find those boids Figure 5 11 shows the average frame rate for 800 boids while varymg the threshold from 5 to 160 Threshold 40 has been shown to produce the fastest frame tmes

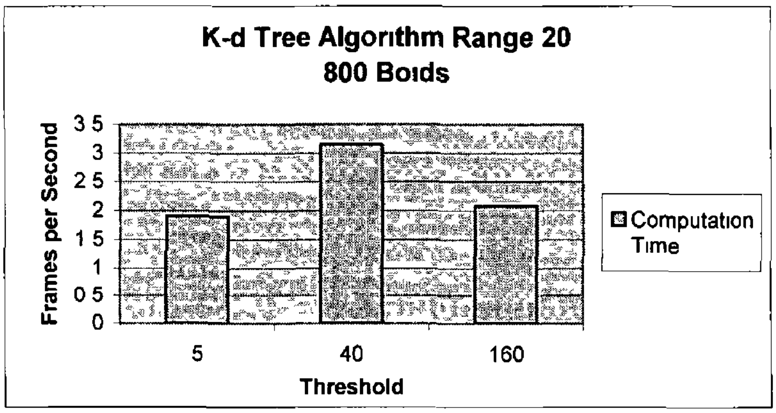


Figure 5 11 K-d Tree Algonthm Range 20 Fps

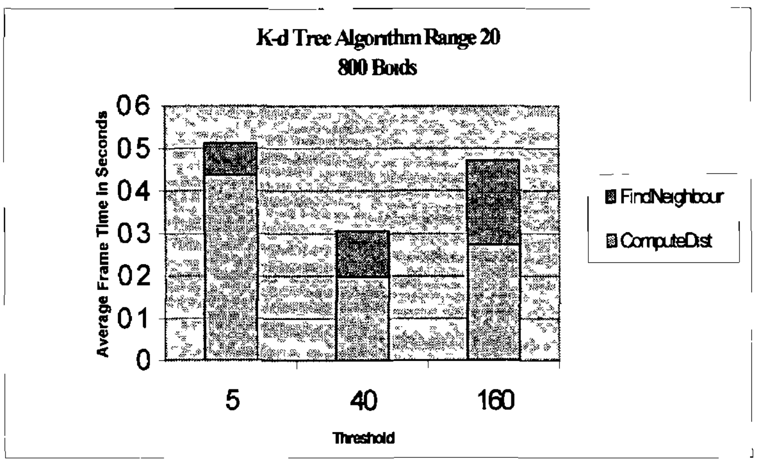


Fıgure 5 12 K-d Tree Algorıthm Frame Tıme Break Down usıng C++ Profiler

As ıs ıllustrated ın figure 5 12 the threshold 40 yıelds the fastest combınatıon of number of boıds tested and tıme take to find those boıds As the range ıncreases two factors cause the k-d tree neıghbour query to decrease

1 As the range ıncreases, the algorıthm traverses more nodes to find all the boıds that are potentıally ıncluded as nearby neıghbours

2 As the range ıncreases, more boıds are ıncluded as nearby neıghbours thus cullıng a smaller percentage of the overall flock than a lesser range

5 3 Comparıson Neıghbour Fındıng wıth Reynolds'87 and K-d Tree

In thıs sectıon we compare our k-d tree approach to Reynolds '87 method of ımplementıng hıs flockıng algorıthm

Fıgure 5 13 compares the average frame rate of the k-d tree approach to Neıghbour findıng versus the approach by Reynolds '87 There ıs a marked ımprovement as the number of boıds ıncreases, the k-d tree approach ıs far more efficıent than the brute force search ımplıed by Reynolds '87 As the number of boıds ıncreases the k-d tree culls a larger percentage of the boıds ın the entıre flock

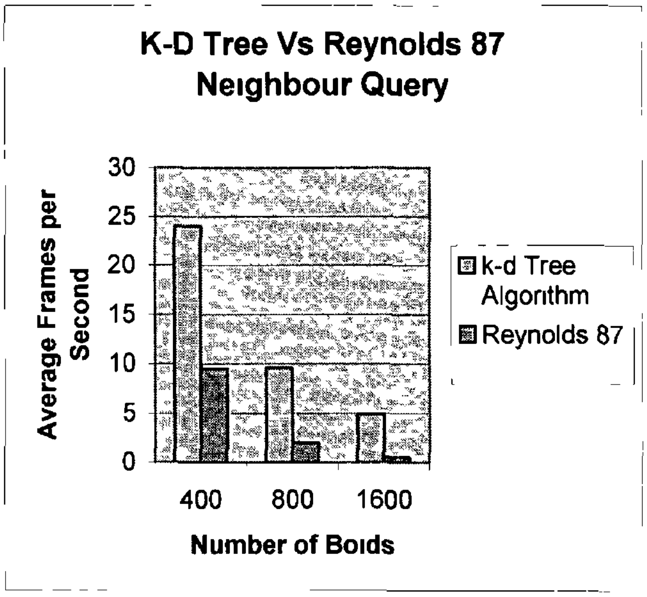


Figure 5.13 K-d Tree Vs Reynolds 87 Neighbour Query Frame rates

The neighbour query is only a part of the flocking computation, figure 5 14 and 5 15 illustrate the comparison between the k-d tree approach and Reynolds'87 The tests are preformed with each boids having a range of 10 units The average frame rate relates to the computation of flocking algorithm As the number of boids increase the k-d tree approach proves to be more efficient due to the faster neighbour query computations For instance for 1600 boids our approach is 10 times faster that Reynolds'87

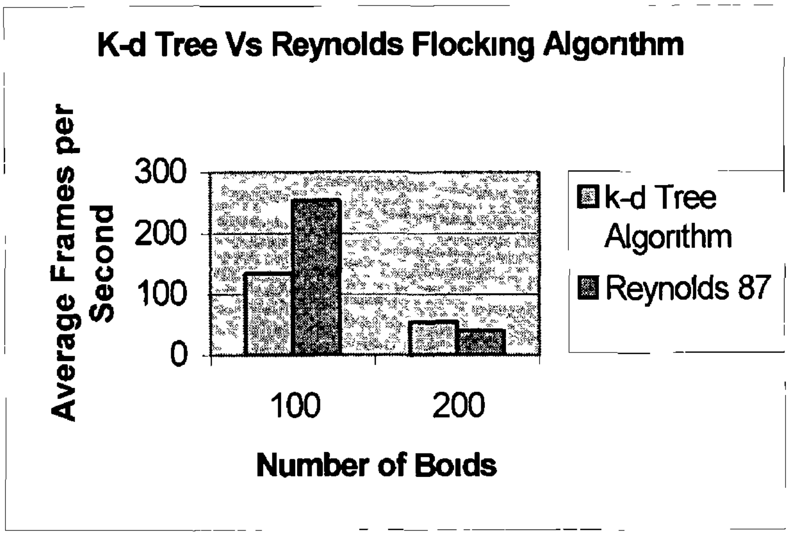


Figure 5.14 K-d Tree Vs Reynolds Frame rates

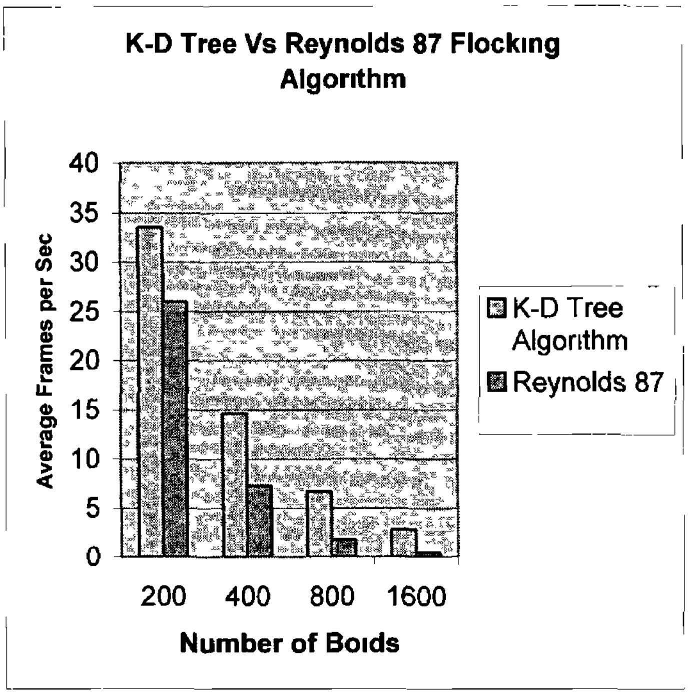


Figure 5 15 K-D Tree Algonthm Vs Reynolds Frame rates

Avoiding Obstacles and Changing Goal

In the test environment there are only a few obstacles and a couple of goals So only a small percentage of the cost is used in computing these two steering acceleration compared to the cost of the flocking behaviour Neither have a marked effect on the frame rate

5 4 Stablegroup Algorithm

This section investigates the fps of the stablegroup algorithm varying the number of boids There are three tests containing 100, 200 and 400 boids respectively, each test contains a single obstacle and one change in goal There is one boid per 9 5 units squared and each boid has a visibility range of 10 units The obstacle is spherical in shape with a radius of 12 units and is in the direct path of the boids

Figure 3 16 to 3 18 all have similar features The frame rate initially is approximately the same as the k-d approach then as the stablegroups are created the frame rate jumps considerably When the flock reaches the obstacle the frame rate reduces to a value similar to the k-d tree approach Once the flock has cleared the obstacle the frame rate increases once again as the stable group is created When the flock reaches a goal and is steered toward the next goal the frame rate once again is reduce to the same speed as the k-d tree approach

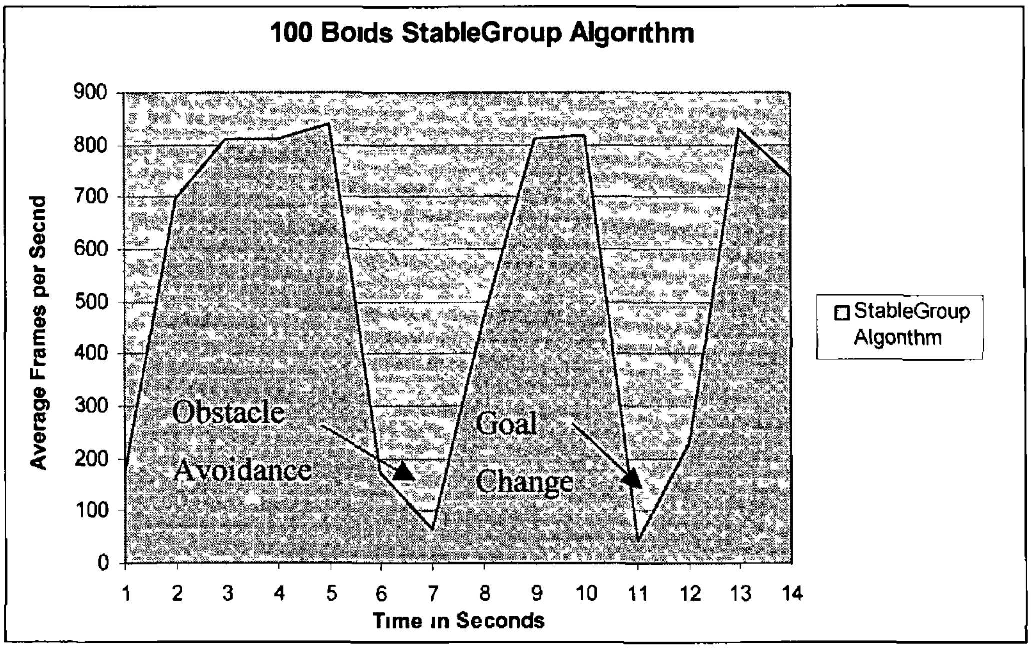


Figure 5 16 Shows average frame rates of stablegroup algorithm with a flock of 100 boids as it moves around obstacle and changes goal

Figure 5 16 illustrates the average frame rates as the flock moves around the world Initially the frame rate is approximately 65 fps, which is approximately the same as the k-d tree approach As the stablegroup is created the frame rate quickly jumps to 800 fps It remains at 800 fps until it is meets an obstacle or changes goal The frame rate returns to 65 fps during those periods, but quickly increases to approximately 800 fps as the stablegroup is created again

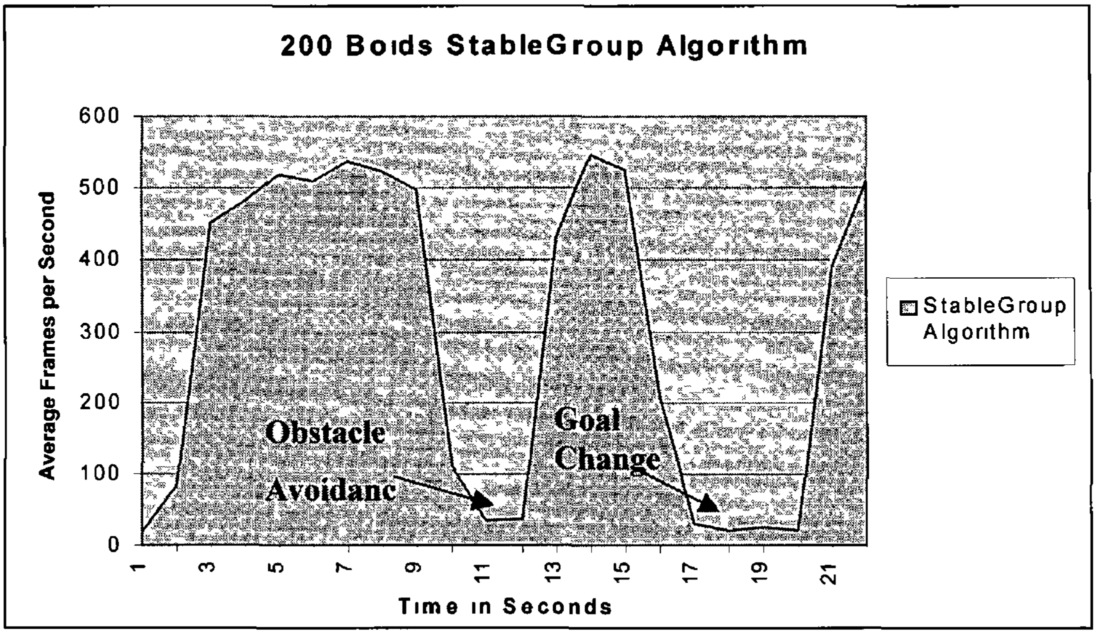


Figure 5 17 Average frame rates over a time period as flock of 200 moves around an obstacle and changes goal

Figure 5 17 illustrates the average frame rates as the flock moves around the world Similarly to figure 5 16 the frame rate is high as the stablegroup is created and decreases to that of the k-d tree approach while the flock avoids an obstacle or changes goal In this case the frame rate increases to 510 fps as the stablegroup is created and decreases to 34 as the individual boid behaviours are computed

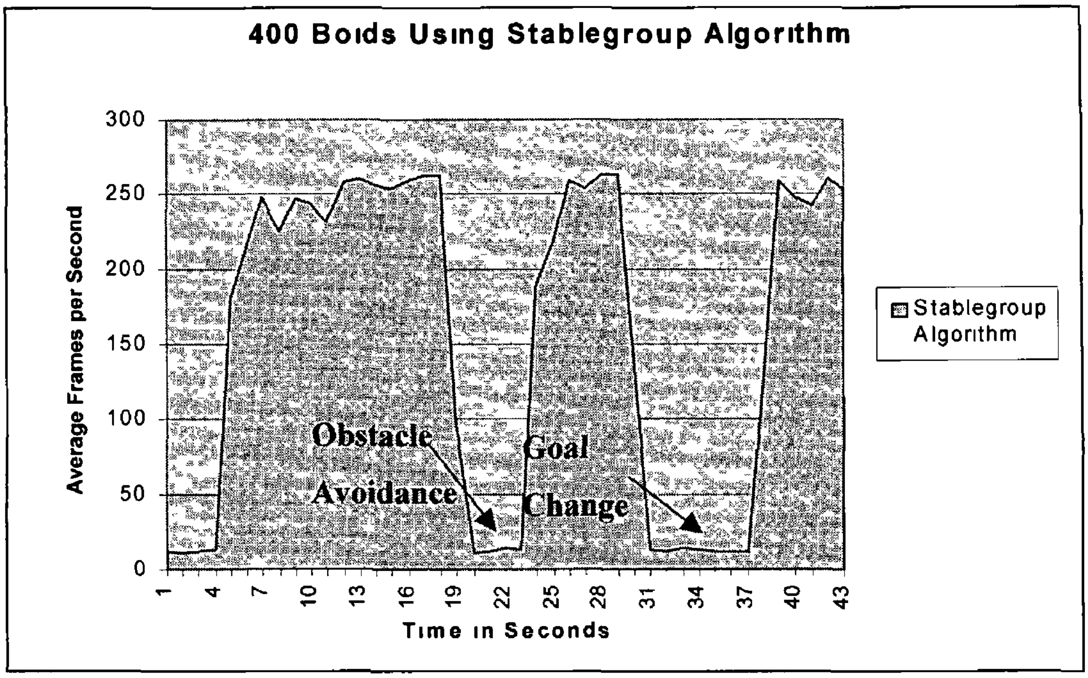


Figure 5 17 Average frame rates over a time period as flock of 400 moves around an obstacle and changes goal

Above (Figure 5 17) the fps for a flock of 400 boids is shown As the stablegroup is created the fps increases to 250 fps, the frame rate drops 14 fps as the flock steers avoids an obstacle and changes goal

Meeting another flock

The test contains two flocks of 200 boids each, each having a different goal The two flocks collide with each other and shortly there after veer away from each other Figure 5 18 shows the fps change as two flocks meets The frame rate decreases as each individual boid's behaviour is computed The fps increases again as the two flocks move away from each other The frame rate dips to the k-d tree frame rate as the flock meets another flock As soon as the flocks start to depart from each other they become stable again the frame rate increases

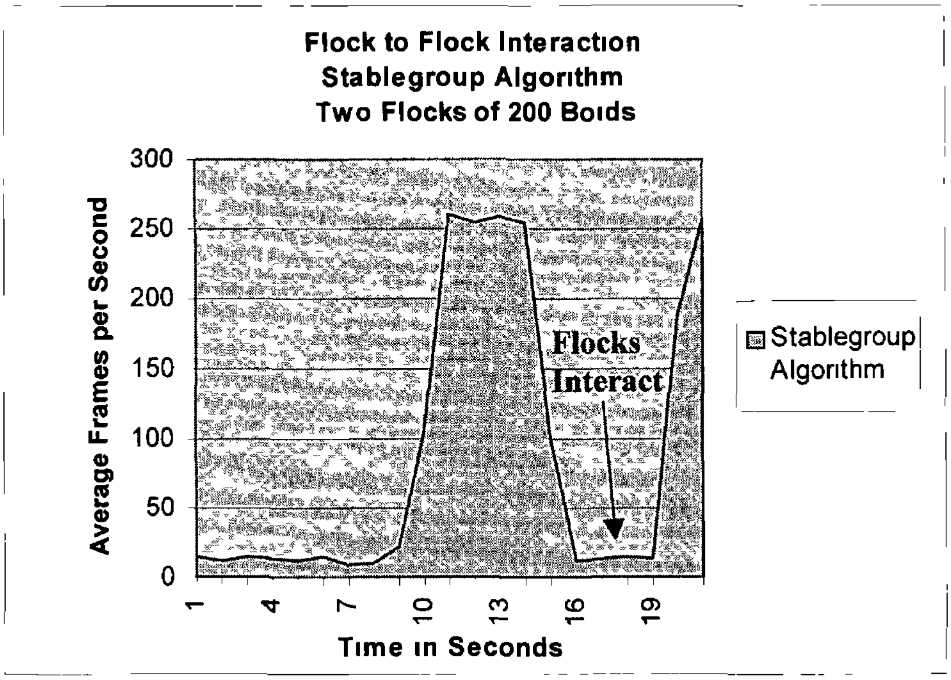


Figure 5 18 Average fps as two flocks of 200 boids meet each other then depart

5 5 Out of View Stablegroup

As described in section 4 11 when a stablegroup goes out of view there is no need to update each boids position in the stablegroup Thus accelerating the frame rate dramatically The test illustrated in figure 5 19 contains 200 boids, which move out of view During that time the fps increases to approx 1600 fps, the stablegroup then moves into view and the frame rate decreases to approximately 510 fps As the stablegroup again moves out of view the frame rate increase once again to 1600 fps

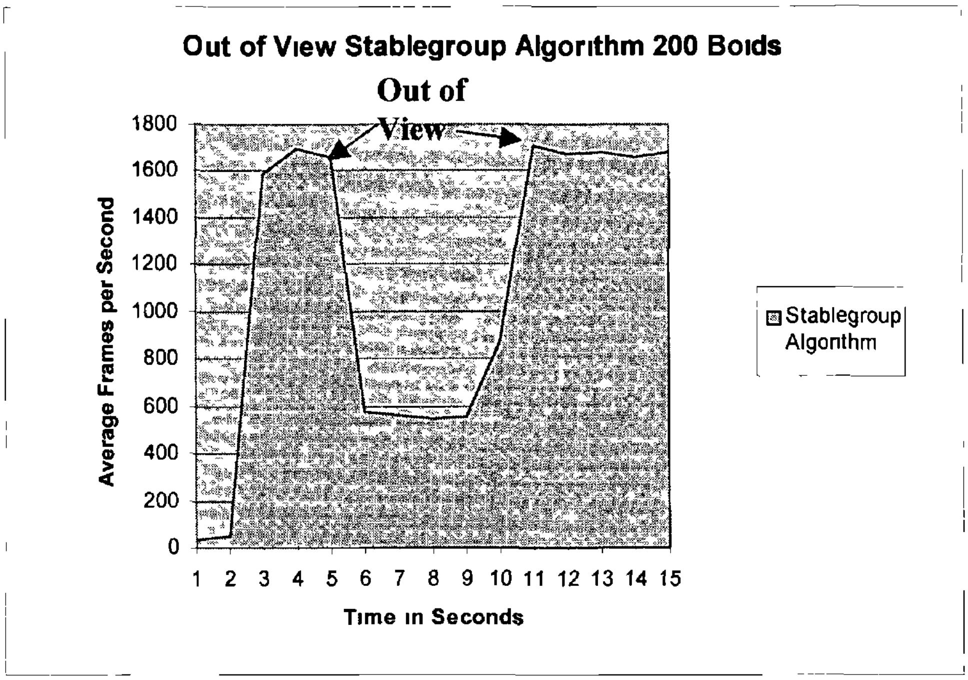


Figure 5 19 Out of View Stablegroup containing 200 boids

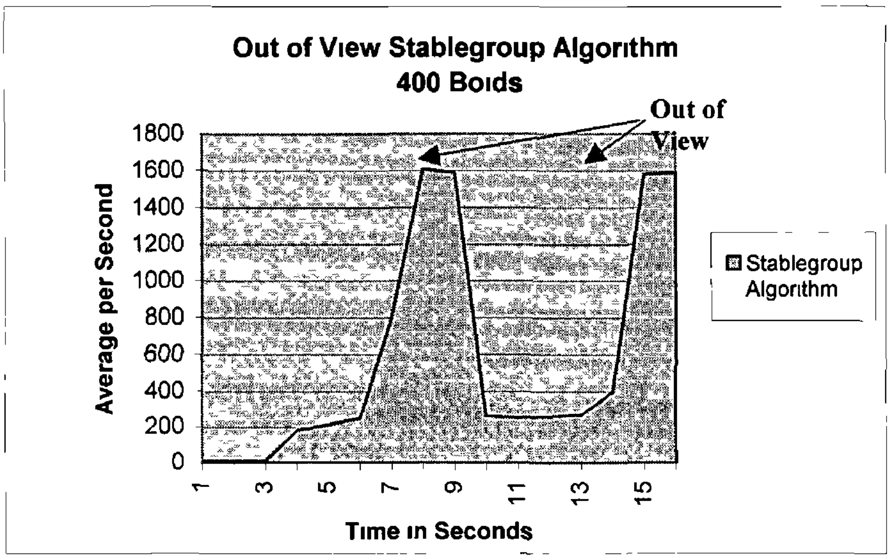


Figure 5 20 Out of View Stablegroup containing 400 boids

The test illustrated in figure 5 20 contains 400 boids which move out of view During that time the fps increases to approx 1600 fps, the stablegroup then moves into view and the frame rate decreases to approximately 250 fps As the stablegroup again moves out of view the frame rate increase once again to 1600 fps

5 6 Viewer Trials

This viewer trial was conducted to test if viewers of the stablegroup animation could notice a difference to the flocking animation where each boids behaviour is computed separately The trial consisted of 2 demos in Appendix A Demo1 is the *kdtreedemo* and demo2 is the *stablegroupdemo* Each demo was shown side by side on 400mhz PC The participant looked at the demos for a short period and then commented on the two demos There were fifteen people involved in the trial Of those 15, 12 people felt that the only difference was that demo2 ran at a faster rate when between the goals Three people felt that the flock in demo2 accelerated between the goals but also on more careful scrutiny noticed that demo2 has slightly less relative motion of each of the boids when it accelerated None of the participants felt that there was a significant difference in the demos and they both acted in a fish like manner

Chapter 6
Conclusions and Future Work

In the preceding chapters we outline our approach to increasing the efficiency of the flocking algorithm introduced by Craig Reynolds I introduce the naive algorithm to compute the flocking behaviour for each boid in the flock For each individual boid the naive algorithm tests each other boid to find its nearest neighbours The flocking behaviour is computed each animation frame computing the boids new position and orientation I outline our approach in detail, on how we increase the efficiency of the flocking algorithm There are a number of steps involved, firstly the k-d tree is created and the boids in the scene are inserted into it The k-d tree is used to cull much of the boids from the neighbour finding computation The tests we run shows that our approach is more than the Reynolds original algorithm The next step is to decrease the time taken for behaviour computation for the boids

Our new approach is to group together a number of boids into a single group and to compute the behaviour for the group as a whole The behaviour is determined by computing the individual behaviour for the outermost boids in the stablegroup object The behavioural vector for the stablegroup is obtained by computing average acceleration vector for the outer-boids We outline the creation, update, combining and destruction of stablegroups in the scene In chapter 5 our tests show that this method allows for much larger flocks to be updated at interactive rates on a PC Our tests illustrate large decreases in the cost behaviour computation for the flock We also outline how obstacle avoidance and flying to a goal is performed for individual boids and Stablegroups We then present our novel algorithm for updating out of view stablegroups This novel algorithm reduces possibly millions of computations to a few dozen in many situations There are a number of avenues of future direction

6.1 Amortize Behaviour Computation

Reynolds presents a method of [Rey2000] of amortizing the cost of the behaviour computation across 6 frames Each boid only updates its behaviour every 6 frames At each frame the algorithm chooses one sixth of the boids at random to update their behaviour The other boids use the acceleration from the previous frame This technique could be applied to our algorithm This would result in a large increase in frame rate when the boid's behaviour is being individually computed For instance 400 boids are currently updated every frame at a rate of 15 fps Using the above approach this could be increased to 90 fps The behaviour computation of the stablegroup could also utilise the same technique by only computing the behaviour every sixth boid in the outer-boids list Indeed since stablegroup acceleration is small only one boids in ten or twenty could be updated depending on the acceleration of the stablegroup

6 2 Adaptive Algorithm

Funkhouser and Sequin [Fun93] present an approach to rendering a 3D scene within a bounded frame rate Their research is only concerned with the visual appearance of the object on the screen

To bound the frame rate of flocking behaviour computation both the benefit and cost of updating individual boids and the stablegroups in the scene need to be determined The benefit of individual boids depends mainly on its acceleration, higher acceleration yielding higher benefit Increased acceleration is caused by

- Meeting other boids
- Steering around obstacles
- Changing goal

Boids that are performing the above behaviours have increased benefit to the environment These boids should be updated at a higher rate than other boids Benefit also depends on the distance of the boids to the user, closer boids having a higher benefit The time since last update is used in the benefit computation also, the benefit increases as the time since last updated grows This ensures all boids do eventually get updated Boids that have not been updated use their acceleration from the previous frame

The cost of computing the behaviour can be estimated from the number of neighbours that boid has

A list of candidates boids to be updated sorted by benefit is created N number of boids are chosen to be updated from the candidate list N is chosen so that cost of the first N boids does not exceed the threshold frame rate We compute the behaviour of these boids We then update the position and orientation of the boids in the environment We render the scene and then compute the new benefit and cost of each boid and create a newly sorted list

Stablegroups are updated adaptively in a similar manner Since the acceleration of a stablegroup is low the main factor in determining the benefit is the size of it on screen, and number of boids contained within it As outlined in 6 1 only a percentage of the outer-boids behaviour need be computed each frame More distant stablegroups would be updated less frequently and when updated would compute a smaller percentage of its outer-boids to determine its acceleration As with individual boids the benefit increases as the time since last update increases The cost is estimated from the cost of updating the outer-boids that are to be updated These stablegroups are inserted in the sorted list of candidates to be updated

This adaptive algorithm will yield a bounded frame rate and update boids based on their behavioural importance to the scene

6 3 Rendering

There are many methods of rendering distant objects in computer graphics More efficient rendering may make use of stablegroups, which can be treated as a single object Impostor selection is one approach suited to more efficient visualisation of distant flocks The rendering of individual boids in stablegroups may be replaced by a texture mapped impostor as in [Sha96] or multiple impostors as in [Aub98]

6 4 Obstacle Avoidance

The algorithm used for obstacle avoidance is quite straightforward More complex obstacle avoidance could be employed One such avoidance technique is silhouette avoidance as outlined in section 2 4 11 There are a number of possibilities for caching of information For instance as boids move to avoid the obstacle they find a point on the silhouette of the object to steer towards Many of the boids behind in the flock will take very similar avoidance measures

Ideally stablegroups that are out of view for a considerable amount of time need not have their behaviours updated Take for instance a stablegroup that is out of view, which avoids an obstacle and gather back together into a single stablegroup again If the above happens while it is out of view and the obstacle avoidance has little effect on the appearance of the stablegroup then the obstacle avoidance routines need not be computed An out of view stablegroup can merely ignore the obstacle There are a number of issues involved, how to determine the effect of an obstacle on a stablegroup There are a number of criteria, the size of the stablegroup, the size of the obstacle, the flocking behaviour characteristics Each of which have to be included in determining the effect an obstacle will have on a stablegroup Secondly the stablegroup must be out of view until it has regrouped at the other side of the obstacle If not then the user must be shown a consistent view of what the stablegroup would look like if the full obstacle avoidance algorithm had been used There are many issues involved, which warrants further research

6.5 Animation

Currently the flock is not animated Obviously for a more realistic simulation animation would be advantageous In a large flock of birds, the animation of boids maybe duplicated in the flock Once a boid is rendered the resulting rendered image maybe used to render other boids in the flock more efficiently

6.6 Temporal Bounding Volumes

As outlined in Section 2 2 5 [Sud96] introduced the idea of temporal bounding volumes (TBV) Dynamic objects are replaced by a static TBV The TBV is inserted into the tree until it becomes visible or the time limit is up on it The TBV could be extended to replace stablegroups with a TBV Determining the volume of the TBV is complicated because the stablegroups interact with each other and the environment The TBV created cannot overlap other stablegroups or their TBV's or indeed obstacles in the scene This approach may work well for sparsely populated environments or in conjunction with obstacle avoidance techniques outlined in Section 2 4 11

6 7 Extension to Multi-body Animation.

The only algorithm described in this thesis is the flocking algorithm The approach may be extended to other group behaviour systems where a stable region of the group maybe identified and the behaviour computation replaced by a simpler more efficient calculation Flocking algorithms are not confined to the air they can also be used for earth bound creatures, such as sheep, wildebeest For virtual animals that follow a terrain the stablegroup would have to be augmented to account for the terrain following behaviour When the stablegroup is created the stablegroup should follow the terrain also Instead of each member of the herd or flock computing its terrain hugging acceleration and flocking behaviour, the stablegroup would compute one value for the terrain following and flocking behaviour

Crowding behaviour is very similar to flocking behaviour and the stablegroup approach could be adapted to accelerate crowding behaviour using the acceleration of each member to determine when to convert to a stablegroup

The stablegroup approach could be also investigated to acceleration real-time fluid dynamics and soft body simulations

Appendix A

Demonstration of K-d Tree and Stablegroup Algorithm

Fıg A 1 ıllustrates the 3D envıronment for the k-d tree and stablegroup algonthm There ıs one obstacle, three goals and one flock contaımng 50 boıds The demo programs are on the CD accompanyıng thıs thesıs The demo programs are based on the clıent server model, each demo program contaıns a server and a clıent To run each demo, the server program ıs executed then the user runs the correspondıng clıent demo

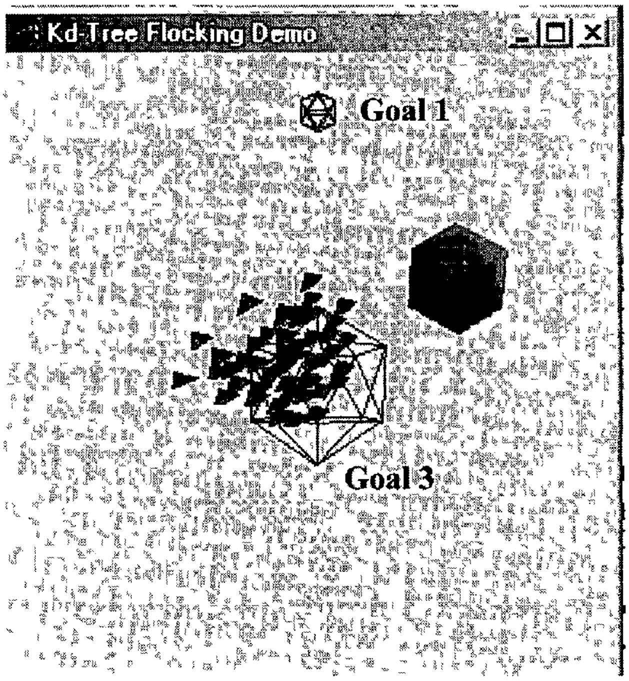


Fıg A 1

The boıds are ıntıalısed near goal 3 The boıds first head towards goals 1 then 2 then back to 3 As the boıds change goal from 3 to goal 1 ıt passes by the obstacle Any boıds on collısıon course wıth the obstacle steer to avoıd ıt The boıds contınue to travel from goal to goal untıl the wındow ıs closed by clıckıng on the top nght of the wındow

K-d Tree Demo

This demonstration application illustrates the k-d tree finding algorithm as outlined in section 3 2 The frame stays relatively constant throughout the lifetime of the program To run the application double click on the *KdTreeDemoServer exe* followed by *KdTreeDemoClient,exe* in windows Explorer When closing the demo, close both the client and the server

Stablegroup Demo

This demonstration application illustrates the Stablegroup algorithm as outlined in section 4 1 The boids follow the same path as in the k-d tree algorithm (illustrated in Fig A 1) To run the application double click on the *StableGroupDemoServer exe* followed by *StableGroupDemoClient exe* in windows explorer When closing the demo, close both the client and the server The stablegroup algorithm causes the frame rate to increase dramatically when the entire flock becomes stable This is especially noticeable as the flock travels from Goal 1 to Goal 2

Out of View K-d Tree demo

This demo illustrates the k-d tree algorithm with the first goal much further away (three times further than the first demo) To run the application, double click on the *OutOfViewKdTreeServer exe* followed by *OutOfViewKdTreeDemoClient exe* in windows explorer When closing the demo, close both the client and server

Out of View Stablegroup Demo

This demo illustrates the speedup gained by applying the out of view stablegroup algorithm outlined in section 4 11 This demo differs from the previous two in that the first goal is further away (three times further) To run the application, double click on the

OutOfViewStableGroupServer exe followed by *OutOfViewStableGroupClient exe* in windows explorer When closing the demo, close both the client and server/

Bibliography

[Abr96] M Abrash *Inside Quake Visible Surface Determination* Dr Dobb's Sourcebook January/February 1996 pp 41-45

[Air91] John M Airey, John Rohlf, Frederick Brooks Jnr *Towards Image Realism with Interactive Update Rates in Complex Virtual Environments SIGGRAPH* '91 pp 41-50

[Ali96] Aliaga, D G *Dynamic Simplification Using Textures*, UNC Technical Report No TR96-007, Deptartment of Computer Science, University of North Carolina, Chapel Hill, NC 1996

[Aub98] Aubel, R Boulic, D Thalmann *Animated Impostors for Real-time Display of Numerous Virtual Human*, Proc Virtual Worlds '98, Paris, France, 1998, pp 1428

[Bat92] Batman Returns (1992), *Motion Picture*, Warner Bros , 1992

[Blu95] Bruce Blumberg and Tinsley Galyean *Multi-level control for animated autonomous agents Do the right thing oh, not that* Creating Personalities for Synthetic Actors, pages 74--82 SpringerVerlag Lecture Notes in Artificial Intelligence, 1997

[Can87] John Francis Canny *The Complexity of Robot Motion Planning* PhD Thesis, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, May 1987

[Car97] Deborah A Carlson and Jessica K Hodgins *Simulation Level of Detail for Real-time Animation* Georgia Institute of Technology Graphics Interface 97

[Cha95] Chamberlain B , Derose T , Lischinki D , Salesin D , Snider J *Fast Rendering of Complex Environments Using a Spatial Hierarchy.* SIGGRAPH '95

[Cha87] David Chapman *Planning for Conjunctive Goals* Artificial Intelligence volume 32, number 3 , July 1987 pp 333-378

[Che96] Han-Ming Chen, Wen-Teng Wang, *The Feudal Priority Algorithm on Hidden-Surface Removal* SIGGRAPH '96 pp 55-64

[Che93] Shenchang Eric Chen and Lance Williams *View Interpolation for image synthesis* SIGGRAPH '93 pp 279-288

[Che95] Shenchang Eric Chen, *QuickTime VR- an Image Based Approach to Virtual Environment Navigation,* SIGGRAPH '95, August 1995, pp 29-38

[Che97] Stephen Chenney *Culling Dynamical Systems in Virtual Environments* Symposium on Interactive 3D Graphics April 27-30, 1997 Providence, RI

[Coo97] Satyan Coorg Seth Teller SIG MIT.*A Spatially and Temporally Coherent Object Space Visibility Algorithm* Symposium on Interactive 3D Graphics, Providence, Rhode Island, April 27-30, 1997

[Dav94] Mark Davies and Patrick Green *Perception and Motor Control in Birds* Springer-Verlag 1994 P 258

[Duc97] M Duchaineau, M Wolinsky, D E Sigeti, M C Miller, C Aldrich, and M B Mineed-Weinstein *Roaming terrain Real-time optimally adapting meshes* In Proceedings IEEE Visualization'97, pages 81--88, 1997

[Fuc79] Fuchs , Kedem and Naylor *Predetermining Visibility Priority in 3D Scenes* SIGGRAPH '79, pp-175-181

[Fun93]Thomas A Funkhouser , Carlo H Sequin UNC at Berkeley *Adaptive Display Algorithm for Interactive Visualisation of Complex Virtual Environments* SIGGRAPH '93 pp 247-252

[Gre94] Ned Greene and Michael Kass. *Error-Bounded Antialised Rendering of Complex Environments* SIGGRAPH '94 pp 59 - 66

[Gre96] Ned Greene *Hierarchical Polygon Tiling with Coverage Mask* SIGGRAPH '96 pp 65-72

[Gre93]Green N , Kass M , Miller G , *Hierarchical Z-Buffer Visibility* In Computer SIGGRAPH '93, pp 231-238

[Hop98] Hugues Hoppe *Smooth view-dependent level-of detail control and its application to terrain rendering* In David Ebert, Holly Rushmeier, and Hans Hagen, editors, Proceedings Visualization '98, pages 35--42 IEEE Computer Society Press, October 1998

[Hub 82] Hubschman H and Zucker S *Frame to Frame Coherence and the Hidden Surface Computation Constraints for a Convex World* ACM Trans on Graphics 1982, pp129-162

[Kuf99] J J Kuffner and J C Latombe *Fast Synthetic Vision, Memory and Learning Models for Virtual Humans* Computer Animation '99 , pages 118-127

[Lin 96] Lindstrom P ,Koller D , Hodges L , Ribarsky W , Faust N ,Turner G *Level-Of-Detail Management for Real-Time Rendering of Phototextured Terrain.* SIGGRAPH 96

[Lio94] The Lion King (1994), *Animated Motion Picture*, Walt Disney Productions, 1994

[Lue95] Davıd Luebke and Chrıs Georges *Portal and Mırrors Sımple, Fast Evaluatıon of Potentıally Vısıble Sets* 1995 Symposıum on Interactıve Graphıcs, pp 105-106

[Mac95] Paulo W C Macıel and Peter Shırley *Vısual Navıgatıon of Large Envıronments Usıng Textured Clusters* 1995 Symposıum on Interactıve 3D graphıcs ACM

[McM95] Leonard McMıllıan and Gary Bıshop *Plenoptıc Modellıng An Image-based renderıng system* SIGGRAPH ‘95 pp 39-46

[Mea82] D Meagher *Effıcıent Synthetıc Image Generatıon of Arbıtrary 3-D Objects* Proc, IEEE Conference on Pattern Recognıtıon and Image Processıng, pp 473-278, June 1982

[Mıc95] Mıcrosoft, *DırectDraw API Specıfıcatıon* and *Dırect3D API Specıfıcatıon.*, Mıcrosoft Corporatıon , Redmond WA, 1995

[Mol92] Molnar, S , J Eyles , J Poulton *PıxelFlow Hıgh Speed Renderıng Usıng Image Composıtıon* SIGGRAPH ‘92, pp 231-240

[Qua96] Quake, *ıd Software*, Mesquıte, TX, 1996

[Reg94] Regan, M , and Pose, R *Prıorıty renderıng wıth a vırtual realıty address recalculatıon pıpelıne* SIGGRAPH '94 pp 155--162

[Ree83] Reeves, W , T *Partıcle Systems-A Technıque for Modellıng a Class of Fuzzy Objects* ACM Transactıons on Graphıcs, V2 #2, Aprıl 1983

[Rey87] Craıg Reynolds *Flock, Herds and Schools A dıstrıbuted Behavıoural Model,* SIGGRAPH ‘87, pp 25 34

[Rey2000] Craig Reynolds *Interaction with Groups of Autonomous Characters* Game Developer's Conference 2000

[Sch69] Schumaker, Brand, Gillard and Sharp *Study for Applying Computer Generated Images in Visual Simulation* AFHRL-TR-69-14, US-AF Human Resources Lab, 1969

[Sch96] Gernot Schaufler and Wolfgang Sturzlinger *A Three Dimensional Image Cache for Virtual Reality* Eurographics 96, pp c-227 - c-235

[Sha84] Shaw, E *Fish in Schools* Natural History 84, no 8 (1975), pp 4046

[Sha96] J Shade, D Lischinski, D Salesin, T DeRose, and J Snyder *Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments* SIGGRAPH '96

[Sims94] K Sims *Evolving Creatures* SIGGRAPH'94 pp 15-22

[Sud96] Oded Sudarsky and Craig Gotsman *Output-Sensitive Visibility Algorithms for Dynamic Scenes with Applications to Virtual Reality* Proceedings of Eurographics 96

[Sud99] O Sudarsky and Craig Gotsman, *Dynamic Scene Occlusion Culling*, IEEE Transactions on Visualization and Computer Graphics Vol 5, No 1 1999

[Suth74] Sutherland, I E , R F Sproull, and R A Schumaker *A Characterisation of Ten Hidden Surface Removal Algorithms* ACM Computing Surveys, 6(1), March 1974, 1-55

[Tel91] Seth J Teller Carlo H Sequin UNC at Berkeley *Visibility Pre-processing for Interactive Walkthroughs* SIGGRAPH '91 pp 61-69

[Tel92] Teller S J (1992 October) *Visibility Computations in Densely Occluded Polyhedral Environments Ph D Thesis* Computer Science Division (EECS) , UC Berkeley, Berkeley, California 94720 Available as report No UCB/CSD-92-708

[Ter94] D Terzopoulos, X Tu, R Grzeszczuk *Artificial fishes Autonomous Locomotion, Perception, Behaviour, and Learning in a Simulated Physical World,* Artificial Life '94, 327-351

[Tha2001] D Thalmann, *The Role of Virtual Humans in Virtual Environment Technology and Interfaces,* In Frontiers of Human-Centred Computing, Online Communities and Virtual Environments, Springer, London, pp 27-38 (invited paper)

[Tor90] E Torres, *Optimization of the Binary Space Partition Algorithm (BSP) for the Visualization of Dynamic Scenes* Eurographics 1990, (Montreux, Switzerland) pp 507-518, Elsevier Science Publishers B V (North-Holland), Sept 1990

[Tor96] Jay Torborg and Jim kajiya *Talisman Commodity Real-time 3D Graphics for the PC* SIGGRAPH '96

[Tu96] Xiaoyuan Tu and Demetri Terzopoulos *Artificial Fishes Physics, Locomotion, Perception, Behaviour* Deparment of Computer Science, University of Toronto, Ontario, M56 1A4 1996

[War69] J Warnock, *A Hidden Surface Algorithm for Computer Generated Halftone pictures* Computer Science Dept Univ of Utah, TR 4-15, June 1969