8-2010

Behavioral Animation for Maya Particles Using Steering Forces

Edgar Rodriguez
Clemson University, edrodri2@vt.edu

Behavioral Animation for Maya Particles Using Steering Forces

---

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Fine Arts
Digital Production Arts

---

by
Edgar Alexis Rodriguez
August 2010

Accepted by:
Dr. Timothy Davis, Committee Chair

ABSTRACT

Maya has various built-in solutions for modeling effects. Maya does not, however, offer a built-in solution for modeling so called “steering behaviors.” While users may rely on particle expressions or scene rigs, these solutions are generally time-consuming and ultimately limiting in their cross application. Accordingly, Maya’s particle system was extended, by implementing a new force, which can be used to model a set of steering behaviors. The new force was coded in Python and was implemented using Maya’s Python Application Programming Interface (“API”). The new force can be used in conjunction with the particle system’s existing forces and Maya’s other tools.

TABLE OF CONTENTS

ABSTRACT ii
TABLE OF CONTENTS iii
List of Figures v
Chapter 1 Introduction 1
Chapter 2 Background and Research 3
2.1 Boid Model 3
2.2 Obstacle Avoidance 4
2.3 Steering Behaviors for Autonomous Characters 4
2.4 OpenSteer and Steering Creator 5
2.5 Brain Bugz 7
2.6 Proprietary Solutions 8
Chapter 3 Implementation 10
3.1 Introducing Steering Field in Maya's Particle System 10
3.2 SteeringField Plug-in Implementation 12
3.3 Individual Forces 14
3.3.1 Heading Force 14
3.3.2 Seeking Force 15
3.3.3 Arriving Force 16
3.3.4 Fleeing Force 18
3.3.5 Wandering Force 20
3.3.6 Surface Following Force 23
3.3.7 Path Following Force 28
3.3.8 Obstacle Avoidance Force 30
3.3.8.1 Ray tracing solution 30
3.3.8.2 Maya NURBS surface solution 32
3.3.8.3 Detection Capsule Solution 33
3.4 Group Forces 40
3.4.1 Separation Force 41
3.4.2 Cohesion Force 43
3.4.3 Alignment Force 45
3.4.4 Neighbor Repulsion 46
3.4.5 Leader Following Force with Offset Constraint 50
3.5 Computing Orientation for Particle Instancing 54
3.6 Steering Force Accumulator 56
3.7 Limitations 58
Chapter 4 Application and Results 59
4.1 Overview 59
4.2 Swarming Butterflies 59
4.3 Prey and Predator 63
4.4 Patrolling Fleet Grounded to Surface 67
Chapter 5 71

Conclusion and Future Work 71
5.1 Future Development 72
5.2 Texture Sampling Sensor 72
5.3 Grouping by Color 75
5.4 Curve Targeting 75
5.5 Scriptable Steering Behavior (SSB) Node 75
5.6 Spatial Partitioning 78
APPENDICES 79
Appendix A 80
Appendix B 83
Appendix C 85
Appendix D 93
Appendix E 96
Appendix F 132
Appendix G 133
Appendix H 134
Appendix I 138
Appendix J 140
Bibliography 142

List of Figures

Figure 3.1 12
Figure 3.2 19
Figure 3.3 21
Figure 3.4 22
Figure 3.5 26
Figure 3.6 29
Figure 3.7 35
Figure 3.8 37
Figure 3.9 39
Figure 3.10 41
Figure 3.11 43
Figure 3.12 45
Figure 3.13 47
Figure 3.14 50
Figure 3.15 51
Figure 4.1 62
Figure 4.2 63
Figure 4.3 67
Figure 4.4 70
Figure 5.1 74

Chapter 1

Introduction

Maya software is an industry standard for 3D animation and visual effects. Maya has many built-in tools for modeling and simulating complex effects. For example, Maya's particle system permits the modeling of natural phenomena (such as rain, sparks, fire, and dust). Although Maya's standard tool set is fairly robust, it is necessarily unable to anticipate every use-case. Instead, Maya offers users the ability to enhance the functionality of its existing tool set through custom extensions within the Maya framework.

This paper explains how Maya's particle system was extended to permit ready modeling of a set of behaviors called "steering behaviors," which permit objects to "navigate around their world in a life-like and improvisational manner" [Reyn99]. Steering behaviors are especially important for modeling hordes of creatures, permitting the creatures to appear spontaneous and natural as they respond to obstacles in their paths and the actions of their neighbors.

Maya's particle system provides users a simple way to manipulate the movement of individual particles through force fields. However, these force fields are limited (e.g., to gravity, turbulence, and air drag). Even when combined, these forces are insufficient to achieve the intended behaviors. And although particle expressions are sufficient, such expressions are a significant investment of time and labor with little return. That is, since expressions are particle shape dependent they lack scalability: for every particle shape created the expressions would need to be embedded.

Because a particle's motion is determined by the contribution of external forces, introducing a new force field is a practical expansion of Maya's existing particle system. This paper presents the additional force field, which was coded in Python, using Maya's Python API. The deliverable allows users to simulate simple to complex steering behaviors and is both scalable and intuitive: one force can act on multiple particles and allows a user to interactively modify the properties of the force while receiving instant feedback on the behavior of the particles.

The remainder of this paper is divided into the following sections: (i) Chapter 2, which outlines Reynolds' work on steering behaviors and a sampling of the implementations to-date; (ii) Chapter 3, which describes the steering behaviors implemented in the plug-in and the methods applied in doing so; (iii) Chapter 4, which presents how the steering behaviors are accessed in the plug-in and the results of using them; and (iv) Chapter 5, which summarizes the plug-in and potential extensions and improvements.

Chapter 2

Background and Research

2.1 Boid Model

Reynolds once observed that although the "aggregate motion" of hordes of creatures (such as flocks of birds, herds of land animals and schools of fish) is both a "beautiful and familiar part of the natural world," it is infrequently modeled in 3D [Reyn01]. But, as a result of Reynolds' work in the mid-1980's, hordes of creatures have become commonplace in films and games (computer and video). Specifically, Reynolds introduced an alternative to the inefficient and difficult approach of scripting such motion. At the heart of his approach is the assumption that simulated flocking creatures, which he dubs "boids," are the result of three simple behaviors: (a) separation, which steers a boid away from its neighbors; (b) alignment, which steers a boid such that its heading aligns with its neighbors; and (c) cohesion, which steers the boid toward the average position of its neighbors [Reyn87].

The boid model approximates the behaviors of a flock, herd, or school by assuming that the creatures have limited knowledge of their environment. The creatures do not have access to the "surplus information" available in the software, such as the positions, orientations, and velocities of all objects [Reyn87]. Instead, each boid "maneuvers based on the positions and velocities of its nearby flockmates" [Reyn01].

2.2 Obstacle Avoidance

Soon after developing the boid model, Reynolds informally introduced techniques for obstacle avoidance, which he presented at SIGGRAPH (and published his notes for). These techniques "abandon pre-specification and use an active navigation algorithm 'while' the object is in motion." The animator establishes the end-point, but the object is empowered to "steer" a collision-free path in the dynamic environment. Reynolds distinguishes the ability to avoid obstacles with artificial intelligence. Unlike artificial intelligence, which involves complex planning strategies and the ability to remember and reason about past actions, obstacle avoidance requires only information about the object's "local environment" [Reyn88].

2.3 Steering Behaviors for Autonomous Characters

In 1999, over a decade after introducing the boid model, Reynolds extended the body of his earlier work by presenting a "collection of simple, common steering behaviors" and strategies for "blending these simple steering behaviors together." Reynolds identifies steering behaviors as a requirement for "autonomous characters," which he defines as a category of autonomous agents that appear in computer animation and interactive media (in games they may be described as "non-player characters" or NPC). Autonomous characters generally have both improvisational and robotic characteristics [Reyn99].

Steering behaviors occupy the middle tier of the hierarchy of motion behaviors, in between "action selection" (at the top) and "locomotion" (at the bottom). Reynolds

applies these tiers to an example of a cowboy (on his horse) retrieving a stray cow upon the order of a trail boss:

> . . . The trail boss represents *action selection:* noticing that the state of the world has changed (a cow left the herd) and setting a goal (retrieve the stray). The *steering* level is represented by the cowboy, who decomposes the goal into a series of simple subgoals (approach the cow, avoid obstacles, retrieve the cow). A subgoal corresponds to a steering behavior for the cowboy-and-horse team. Using various control signals (vocal commands, spurs, reins) the cowboy steers his horse towards the target . . .The horse implements the *locomotion* level. Taking the cowboy's control signals as input, the horse moves in the indicated direction.

Each steering behavior (including seek, flee, pursuit, evasion, wander, arrival, wander, and path following) is presented as the "geometric calculation of a vector representing a desired steering force" [Reyn99].

2.4 OpenSteer and Steering Creator

Reynolds has not only established a solid theoretical and practical framework for understanding modeling steering behaviors, but has also helped set the stage for implementations of his work. Specifically, in 2002, as a part of Sony Computer Entertainment America's Research and Development Team, he developed a C++ library "to help build steering behaviors for autonomous characters in games and other kinds of multi-agent simulations" [Reyn07].

This library has been made available under the name “OpenSteer” and includes sample code and “examples of combining simple steering behaviors to produce more complex behavior” [Reyn07]. The OpenSteer library is supplemented by the OpenSteerDemo, which has a plug-in framework for users to add their own plug-ins to “quickly prototype behaviors during game design, and to develop behaviors before the main game engine is finished” [Reyn07]. Both OpenSteer and OpenSteerDemo, as their names suggest, are open source.

OpenSteer and OpenSteerDemo target game developers, giving them a “toolkit” for coding steering behaviors, as well as a platform to test the behaviors separate from the game being developed. This reflects Reynolds’ background in game development as well as the motivation for his early work: steering behaviors are important for modeling “autonomous characters” in (video and computer) games.

Unfortunately, the utility of OpenSteer and OpenSteerDemo to effects artists is somewhat limited. As a work-in-progress, OpenSteer has not developed a means to integrate the library with the user’s own code (in any platform). In order to integrate the library, an effects artist would have to create an interface to bridge the library into the Maya framework. And, because Maya is used on Linux, Mac, and Windows, there would potentially be a need to support all three platforms. The same limitations apply to the also open source SteeringCreator, which is a collection of Java applets developed by Schnellhammer and Feilkas that implement Reynolds’ work and have been made available through the website: www.steeringbehaviors.de [Schn10].

2.5 Brain Bugz

In 2002, Kolve applied Reynolds' work by developing an opensource C++ plug-in for Maya's particle system called "Brainbugz." Kolve reasoned that a particle system applies forces to mass-points in order to change their position, while in "behavioural [sic] animation, mass-points are not forced from the outside, forces are *self applied* based on physical attributes and one or many behavioural [sic] rules" [Kolv09]. By attaching the self applied forces, or "*steerings desires,*" to the mass-points they become autonomous characters, which he dubs "bugs."

Unlike OpenSteer and StearingCreator, BrainBugz offers seamless integration with Maya. After loading the plug-in, a "brainbugz" menu-bar appears, permitting the user to select the appropriate command (e.g., "seek"). Because the "steering desires" are made available within Maya particle system, a user can apply Maya's built-in force fields (e.g., gravity) in conjunction with them. Within the platform, a user may also model objects to act as obstacles, or build rigs that drive a moving target.

As a C++ plug-in to the Maya particle system, BrainBugz offers many advantages to the C++ OpenSteer and is an altogether effective implementation of Reynolds' steering behaviors. But, BrainBugz has at least one inconvenience: the code must be compiled with each new version of Maya. This requires the ongoing interest of the open source community to ensure that the code has been compiled to the latest version. Fortunately, to date the code has been compiled to the latest version.

With Maya 8.5, Autodesk introduced support for Python, permitting users to write plug-ins in Python via the Maya Python API. Plug-ins in Python, unlike those in C++,

eliminates the overhead of compiling and linking code, which allows for rapid prototyping. Python is also generally easier to learn than C++, which makes it available to broader audience. In Addition, Python offers users powerful and clear syntax, a large standard library, and object-oriented programming style. Most importantly, Python interfaces with both Maya commands and Maya's API, while C++ does not. This permits the creation of scripts to mix and match Maya commands and Maya's API functions, which is advantageous because certain operations can be accomplished much more easily with Maya commands than with Maya's API calls, while other operations can be accomplished more easily with Maya's API calls than with Maya commands. Although there is the added convenience to interchange Maya API calls and Maya commands with Python, it's important to mention that Maya Python scripts will execute slower than C++ Maya API calls.

2.6 Proprietary Solutions

An example of a proprietary solution is Massive Software, which was developed for the Lord of the Rings trilogy by Weta Digital [MasA10]. Massive Software generates "realistic crowd behaviors and autonomous agent driven animation" that is scalable into the hundreds of thousands [MasB10]. Massive Software has been integrated into the pipelines of a number of studios (e.g., it is used by Pixar, Sony Pictures Imageworks, Dreamworks Animation, Rhythm & Hues) since first being introduced [MasA10].

While proprietary solutions, such as Massive Software, have been widely recognized for their ability to implement steering behaviors, they are only briefly discussed in this paper because they are (i) proprietary, and thus do not offer insight into

their approach; (ii) generally costly (out of reach for students, individual effects artists, and smaller studios); and (iii) generally have many capabilities beyond modeling steering behaviors.

Chapter 3

Implementation

3.1 Introducing Steering Field in Maya's Particle System

A standalone simulator (unrelated to Maya and similar to OpenSteerDemo) was considered as a solution for implementing steering forces. It was ultimately rejected for the reasons outlined in section 2.4 (e.g., having to build a secondary tool to import the data in a usable format that would work with Maya or some other third-party software). Instead, a solution that extends the functionality of Maya's built-in particle system was developed. This solution offers seamless integration with Maya's framework as a result of having access to Maya's core set of interfaces through the API. Unlike BrainBugz, this solution is written in Python and implemented using the Python Maya API. Accordingly, the plug-in does not need to be compiled for any version of Maya that supports Python and offers certain advantages to a C++ plug-in, as outlined above.

On a theoretical level, steering forces are a natural outgrowth of particle systems, as suggested by Kolve (i.e., because a particle system is a force-based environment) [Kolv09]. There are also practical reasons for implementing steering forces in Maya's particle system. Maya's particle system permits a number of rendering styles, such as blobbies and animated sprites. Particle attributes can be keyframed or controlled dynamically with expressions. Particles can be cached, which is important for distributed rendering. With geometric instancing, users can replace each particle with animated

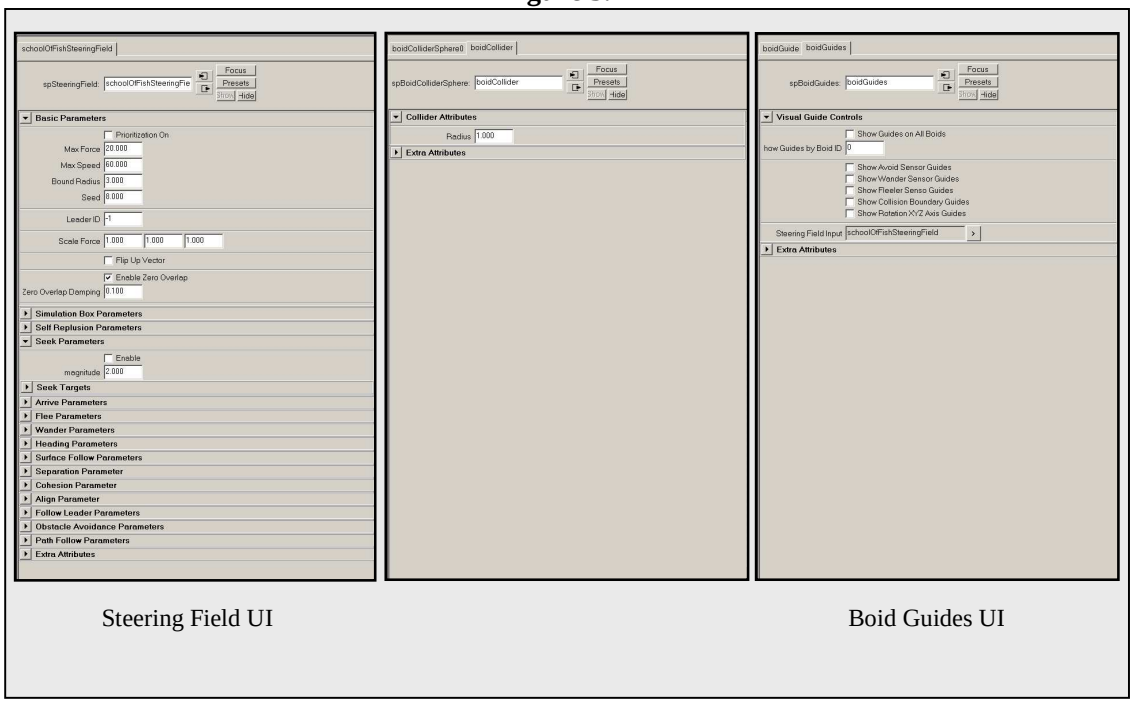
geometric shapes or include various geometric poses that can be animated and controlled by particle expressions.

The solution that was developed has three components, each of which is a separate Maya plug-in: (i) SteeringField.py (the SteeringField); (ii) boidColliderSphere.py (the ColliderSphere); and (iii) the boidGuides.py, (the BoidGuides). Each plug-in introduces a new Dependency Acyclic Graph (or DAG) node (hereafter a DAG node is referred to simply as a node to Maya's framework). The first plug-in, SteeringField, is the core plug-in and like all of Maya's built-in particle force fields, it derives from the base class MpxFieldNode. But unlike Maya's other force fields, SteeringField is a collection of steering forces that can function individually or together to produce various steering behaviors. The second plug-in, the ColliderSphere, creates a collision node, represented by a sphere, that is recognized by the SteeringField. This sphere may be translated, rotated, and scaled. As implemented, no manual association of the SteeringField with the ColliderSphere collision nodes is required. When obstacle avoidance is enabled, the SteeringField automatically queries the Maya scene graph for ColliderSphere collision nodes and resolves any collisions between the particles and obstacles, as described further below. The third plug-in, BoidGuides, provides guides for visualizing steering vectors. These guides can be activated for individual particles or for all particles.

Each plug-in has an attribute editor template file to organize the attributes associated with the plug-in. For example, the AespSteeringFieldTemplate.mel (which is the SteeringField attribute editor template file) is a MEL script that customizes the user

interface in Maya's built-in Attribute Editor when a user selects a DAG node of the type spSteeringField. Figure 3.1 is a snapshot of each custom attribute editor UI.

Figure 3.1



Steering Field UI Boid Guides UI

3.2 SteeringField Plug-in Implementation

Seamless integration into Maya's framework required that the SteeringField extend Maya's MpxFieldNode proxy class. This allows the SteeringField to inherit the basic functionality of MpxFieldNode proxy class, thereby allowing time and development effort to be focused on adding new functionality. Because the SteeringField is a DAG node, the node functions like any standard DAG node, allowing users to establish relationships to and from other nodes. This permits users to have multiple SteeringField nodes acting on one or more particle shapes (which are nodes that contain a

collection of individual particles, each particle being a component of the particle shape). A case study using multiple SteeringField nodes will be presented in Section 4.4.

The SteeringField has two types of steering forces to simulate steering behaviors: individual steering forces and group steering forces. The individual steering forces function regardless of the number of particles to which the individual steering forces are being applied, as long as there is at least one particle. While the individual steering forces are influenced by the objective selected by the user and the particle's environment, these steering forces are oblivious to the actions of neighboring particles. By contrast, the group steering forces will only function when applied to multiple particles (the minimum being two). The individual steering forces implemented include heading force, seeking force, arriving force, fleeing force, wandering force, surface following force, obstacle avoidance force, and path following force. The group steering forces implemented include neighbor repulsion force, separation force, cohesion force, alignment force, and leader following force.

The algorithms presented in the next section of this paper do not explicitly alter a particle's velocity. The algorithms instead, compute a desired velocity that is used in the calculation of the steering force vector. The steering force vector is managed internally by MPxField node, which hands off the force vector to Maya's built-in solver. Maya's built-in solver is responsible for accumulating the input force vectors from all force fields acting on a particle, calculating the acceleration of the particle from the forces applied, and updating the particle's velocity and position. The Steering Field uses the updated velocity and position to calculate the steering force vector for the next simulation step.

For purposes of this paper the up axis is defined along the Z axis and hereto will be refer to as the Z direction.

3.3 Individual Forces

3.3.1 Heading Force

The heading force—the simplest of steering forces implemented in this plug-in—directs the particle towards a desired velocity, without a predefined destination target. In other words, when the particle's current velocity reaches the desired velocity, it will remain at the desired velocity indefinitely.

In this context, the desired velocity is a user-defined vector scaled by the particle's maximum reachable speed (which is a global parameter, hereafter referred to as maximum speed). The steering force is then calculated by subtracting the particle's current velocity from the desired velocity. Both formulas are provided below:

*desired velocity = normalize(user-defined direction vector) * maximum speed*
steering force = desired velocity – particle's velocity

A portion of the heading force code is shown below:

```
def _heading(…):
…
…
…
    for i in range(positions.length()):
        headingHandle = block.inputValue(SteeringField.aHeading)
        heading = headingHandle.asFloatVector()
        curVel = velocities[i]
        if heading.length() == 0.0:
            desiredVel = OpenMaya.MVector(curVel)
        else:
            desiredVel = OpenMaya.MVector(heading)

        steeringForce = self._seekHelper(block, desiredVel, curVel)
        if leaderID > 0:
            if i != leaderID:
                steeringForce = OpenMaya.MVector(0.0,0.0,0.0)
        outputForce.append(steeringForce)
```

A seeking steering force moves a particle towards a specific destination target. To arrive at the desired velocity (i.e., the velocity necessary for the particle to reach the destination target), the position of the particle is subtracted from the position of the target, and then normalized and scaled by the particle's maximum reachable speed. The seeking force is then calculated using the same steering force formula provided for the heading force. The desired velocity formula is provided below:

*desired velocity = normalize(destination target – particle position) * maximum speed*

An important feature of the seeking force is that when applied, a particle will always overshoot and pass the destination target (although the amount overshot depends

on the speed at which the particle is moving). In this situation, the particle will turn around and move towards the destination target again.

A portion of the seeking force code is shown below:

```
def _seekTarget(…):
    …
    …
    …
    for i in range(positions.length()):
        targetIndex = random.randint(1,targetSize) - 1
        desiredVel = targets[targetIndex] - pos[i]
        curVel = vel[i]
        steeringForce = self._seekHelper(block, desiredVel, curVel)
        if leaderID>0:
            if i != leaderID:
                steeringForce = OpenMaya.MVector(0.0,0.0,0.0)
        outputForce.append(steeringForce)


def _seekHelper(self, block, desiredVel, currentVel):
    maxSpeed = self._attrDoubleValue(block, SteeringField.aMaxSpeed)
    maxForce = self._attrDoubleValue(block, SteeringField.aMaxForce)
    dist = desiredVel.length()
    steeringForce = OpenMaya.MVector(0.0,0.0,0.0)
    if (dist > 0):
        desiredVel.normalize()
        desiredVel = desiredVel * maxSpeed
        steeringForce = desiredVel - currentVel
        steeringForce = self._limit(steeringForce, maxForce)

    return steeringForce
```

3.3.3 Arriving Force

The arriving force is similar to the seeking force, in that both forces steer a particle towards a specific destination target. However, the arriving force is different from the seeking force in that it keeps the particle from overshooting the destination target, causing the particle to gradually decelerate until it comes to a complete stop at the destination target.

The desired velocity (i.e., the velocity necessary for the particle to arrive at the destination target) is calculated by subtracting the position of the particle from the destination target and then scaling it by the speed required to reach the destination target. The speed is calculated by dividing the particle's distance to the target by a controllable variable called the deceleration scalar, which dictates how quickly the particle will decelerate as it approaches the target. The speed is capped at the maximum speed. The arriving force is then calculated using the same steering force formula provided for the heading force (and the seeking force). Below are the formulas for calculating the speed and the desired velocity:

speed = (distance to target)/(deceleration scalar)

but if *speed* > *maximum speed*, then *speed* = *maximum speed*

*desired velocity = (destination target – particle position) * speed*

A portion of the arriving force code is shown below:

```
def _arriveHelper(…):

    maxSpeed = self._attrDoubleValue(block, SteeringField.aMaxSpeed)
    maxForce = self._attrDoubleValue(block, SteeringField.aMaxForce)

    decelerationMult = self._attrDoubleValue(block,
                                    SteeringField.aDecelerationMult)
    '''
    deceleration damping factor
    '''
    decelerationDamping = self._attrDoubleValue(block,
                                SteeringField.aDecelerationDamping)

    dist = desiredVel.length()

    steeringForce = OpenMaya.MVector(0.0,0.0,0.0)

    if ( dist > 0.0 ):
        '''
        calculate the speed required to reach target given the
        desiredVel
        '''
        speed = dist/(decelerationMult*decelerationDamping)

        '''
        make sure the current speed does not exceed the maxSpeed
        '''
        speed = min(speed, maxSpeed)

        desiredVel = desiredVel * speed

        steeringForce = desiredVel - currentVel
        steeringForce = self._limit(steeringForce, maxForce)
```

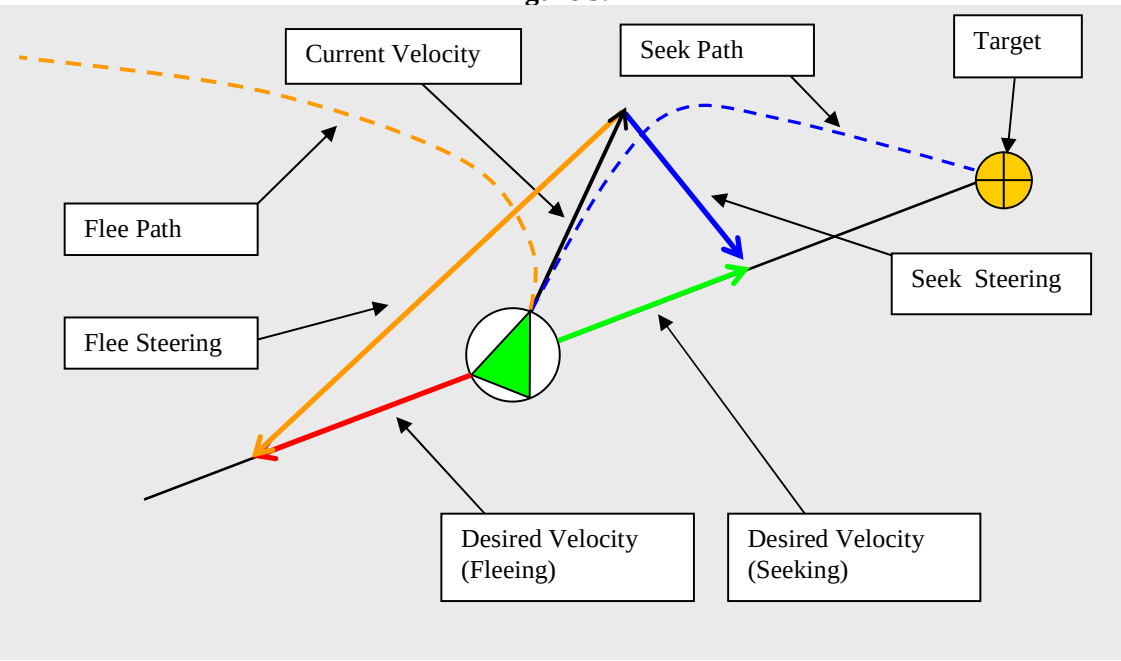
3.3.4 Fleeing Force

While the seeking force moves a particle towards a specific destination target, the fleeing force moves a particle away from it (and therefore is the opposite of the seeking force). Figure 3.2 illustrates the distinction between the fleeing force and the seeking force. The desired velocity (i.e., the velocity necessary for the particle to flee from the

target) is calculated by subtracting the target position from the position of the particle, and then normalizing and scaling it by the particle's maximum speed. The fleeing force is then calculated using the same steering force formula provided for the seeking force. The formula for calculating the desired velocity is provided below:

*desired velocity = normalize(particle position-target) * maximum speed*

Figure 3.2



A proximity test is calculated, such that the particle will not begin to flee until the target is within a specific range. The following is a portion of the fleeing behavior code:

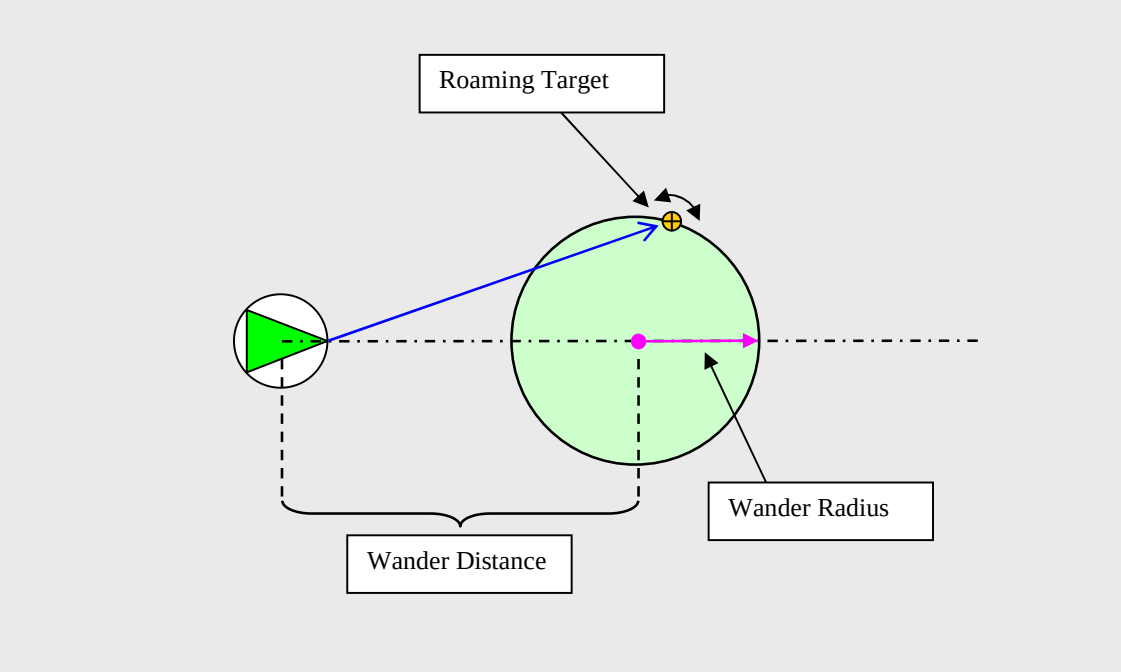
```
def _fleeTarget(…):
…
…
…
    for i in range(positions.length()):
        targetIndex = random.randint(1,targetSize) - 1
        desiredVel = positions[i] - targets[targetIndex]
        DistanceSq = desiredVel.x*desiredVel.x + \
        desiredVel.y*desiredVel.y + \
        desiredVel.z*desiredVel.z
        # proximity test, distance is in squared-distance space
        if(DistanceSq > panicDistanceSq):
            steeringForce = OpenMaya.MVector.zero
        else:
            desiredVel.normalize()
            desiredVel = desiredVel * maxSpeed
        steeringForce = desiredVel - velocities[i]
        steeringForce = self._limit(steeringForce, maxForce)
        outputForce.append(steeringForce)
```

3.3.5 Wandering Force

The wandering force moves the particle in seemingly random directions, with no end goal. Specifically, the particle's velocity changes every simulation step giving the illusion of stochastic mobility. The wandering force is implemented by creating a virtual wander sphere in front of the particle that has a roaming target contained within its boundary. The particle's desired velocity is arrived at using the same method applied in the seeking force, although here, the roaming target is randomly displaced by some factor (within the wander sphere) every simulation step (see figure 3.3). The wandering force should be viewed as an extension of the seeking force because the wandering force is

continuously computing a new target that becomes a new specific target for the seek force.

Figure 3.3



The wandering force is calculated by following *Steps 1-4*, below:

Step 1: A wander distance, wander radius, and displacement jittering factor are given as inputs.

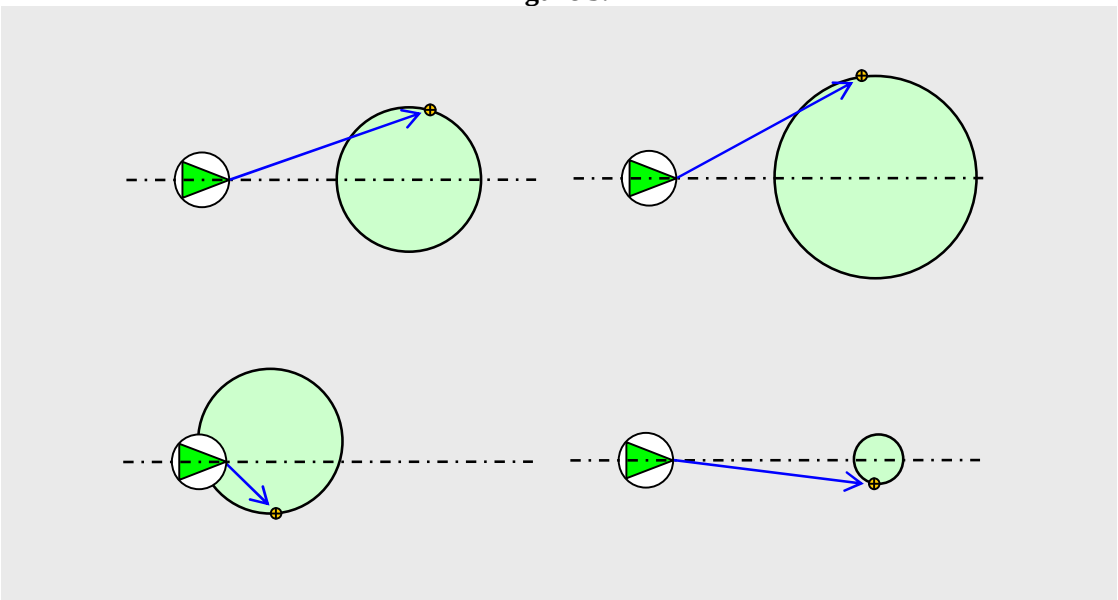
Step 2: For each simulation step, a virtual wander sphere is defined at the particle's world space position. The wander sphere is then projected in front of the particle by the amount equal to the wander distance. This is accomplished by first normalizing the particle's velocity to determine the direction. The direction is then scaled by the wander distance and added to the particle's position, ensuring that the center of the wander sphere is always in front of the particle and parallel to the particle's velocity.

Step 3: For each simulation step, the roaming target's position is randomly displaced by a small fraction. This is accomplished by confining the roaming target to the boundary of a unit sphere located at the origin (0,0,0). A small random displacement (ranging from -1 to 1), scaled by the displacement jitter factor, is added to the roaming target. The roaming target, which is initially defined at the origin, is then scaled by the wander radius and projected into world space by adding the world position of the wander sphere as defined in *Sep 2*.

Step 4: The roaming target is used as the destination target and the desired velocity is arrived at using the same method applied in the seeking behavior.

An important feature of the wandering force is that the distance between the wander sphere and the particle's position affects the degree of the particle's turns (see figure 3.4). When the wander sphere is farther away, the turns are smaller, while a closer wander sphere results in larger turns [Buck05].

Figure 3.4



A portion of the wandering force code is shown below:

```
def _wander(…)
…
…
…
    for i in range(positions.length()):
        wanderCircleOffset = OpenMaya.MVector(velocities[i])
        wanderCircleOffset.normalize()
        wanderCircleOffset = wanderCircleOffset*wanderDistance
        wanderCircleOffset = wanderCircleOffset + positions[i]

        outWanderTargetArray[i].x = (outWanderTargetArray[i].x +
                                     randomClamped()*wanderJitter)*wanderScale.x
        outWanderTargetArray[i].y = (outWanderTargetArray[i].y +
                                     randomClamped()*wanderJitter)*wanderScale.y
        outWanderTargetArray[i].z = (outWanderTargetArray[i].z +
                                     randomClamped()*wanderJitter)*wanderScale.z
        outWanderTargetArray[i].normalize()
        outWanderTargetArray[i].x = outWanderTargetArray[i].x * wanderRadius
        outWanderTargetArray[i].y = outWanderTargetArray[i].y * wanderRadius
        outWanderTargetArray[i].z = outWanderTargetArray[i].z * wanderRadius

        targetWorldPosition = outWanderTargetArray[i] + wanderCircleOffset
        desiredVel = targetWorldPosition - positions[i]

        outWanderSpherePositionArray[i].x = wanderCircleOffset.x
        outWanderSpherePositionArray[i].y = wanderCircleOffset.y
        outWanderSpherePositionArray[i].z = wanderCircleOffset.z

        outTargetPositionArray[i].x = targetWorldPosition.x
        outTargetPositionArray[i].y = targetWorldPosition.y
        outTargetPositionArray[i].z = targetWorldPosition.z

        steeringForce = self._seekHelper(block, desiredVel,velocities[i])
        if leaderID > 0:
            if i != leaderID:
                steeringForce = OpenMaya.MVector(0.0,0.0,0.0)
        outputForce.append(steeringForce)
```

3.3.6 Surface Following Force

The surface following force steers a particle towards the closest point on a non-uniform rational basis spline (NURBS) surface given the particle's position. An input ground offset controls how far from the surface the particle must always remain relative to the closest point on the surface. By itself, this force results in the particle resting on the surface without advancing. Thus, in order for the particle to move along the surface,

the surface following behavior must be used in conjunction with a steering behavior that provides for forward movement, such as a wandering force, seeking force, arriving force, or heading force. An illustration of the surface following force is provided in figure 3.3.6-1.

The surface following force also takes into account whether the particle is moving uphill or downhill. When the particle is moving uphill, it slows down by a controllable factor. Conversely, the particle speeds up by a controllable factor when moving downhill.

The desired velocity (i.e., the velocity necessary for the particle to reach the closest point on the surface) is arrived at by following the steps outlined below:

Step 1: A future position of the particle is predicted based on the particle's trajectory and speed. (An alternative method for predicting the particles' future position is to use an adjustable forward vector whose length determines the particle's future position relative to its position and heading, where heading refers to a particle's normalized velocity).

Step 2: The predicted particle position and the surface are used as inputs to compute the closest point on the surface. Given the closest point, Maya's utility functions are used to derive *u* and *v* values associated with the closest point on the surface. The *u* and *v* values are then used to compute the surface normal at the closest point.

Step 3: An overshoot length is derived by finding the length between the closest point and the forecast particle position. The overshoot is used for checking if the particle will be above the surface, right on the surface, or beneath the surface on the next

simulation step. If the overshoot length is less than or equal to the ground offset, then the desired velocity is set to the surface normal. Otherwise the desired velocity is calculated by subtracting the particle's position from the closest point.

Step 4: The particle needs to take into account the undulations of the surface and adjust its speed accordingly. If it is moving uphill, it should slow down; likewise it should speed up if it is moving downhill. To perform this check, an infinite plane is defined with its normal pointing in the negative Z direction. The resulting sign of the dot product between the plane's normal and the particle's normalized velocity indicates whether the two vectors are pointing roughly in the same direction [Leng02]. If the dot product is greater than zero, then the particle's normalized velocity lies on the same side of the plane, and hence, the particle is going downhill. Likewise, if the dot product is less than zero, then the particle's normalized velocity lies on the opposite side of the plane, and hence, the particle is going uphill. To give the illusion of moving uphill or downhill, the particle's maximum speed is scaled by an adjustable multiplier: an upward multiplier (if moving uphill) or a downward multiplier (if moving downhill).

The formula for computing the desired velocity is provided below:

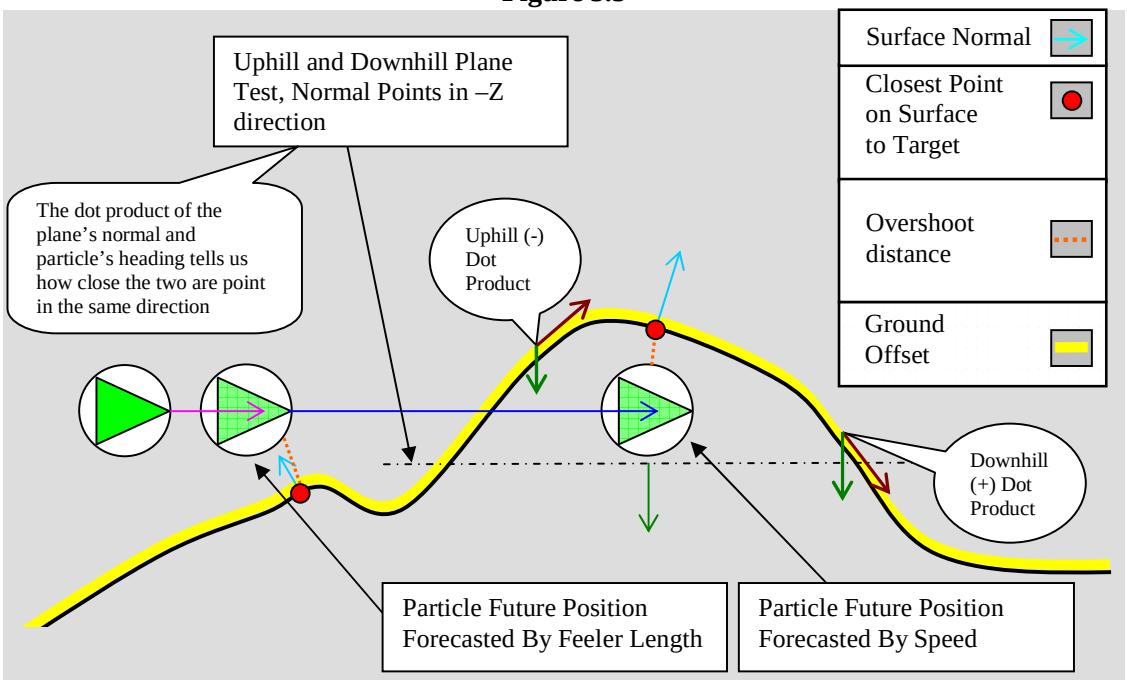
if(overshoot <= ground offset)
desired velocity = normalize(surface normal)
else:
desired velocity = normalize(closest surface point – particle's position)

direction = normalize(particle velocity) dot plane(0,0,-1)

if direction > 0:
*desired velocity = desired velocity * maximum speed * downward multiplier*
else:
*desired velocity = desired velocity * maximum speed * upward multiplier*

steering force = (desired velocity – particle's velocity)

Figure 3.5



A portion of the surface following code shown below:

```
def _surfaceFollow(…)
…
…
…
    for i in range(positions.length()):
        unitVelocity = OpenMaya.MVector(velocities[i])
        mag = unitVelocity.length()
        unitVelocity.normalize()
        forecastPosition = None

        # perdict position of boid
        if groundforecastBySpeedOn:
            # by speed
            forecastPosition = OpenMaya.MPoint(positions[i] + (velocities[i]))
        else:
            # by feeler length
            forecastPosition  = OpenMaya.MPoint(positions[i] +
                                            (unitVelocity*(feelerLength)))

        connections = OpenMaya.MPlugArray()
        g_plug.connectedTo(connections, True, False)
        groundPlug = connections[0]
        nNode = groundPlug.node()

        mfnNurb = OpenMaya.MFnNurbsSurface(nNode)
        uIP = OpenMaya.MScriptUtil().asDoublePtr()
        vIP = OpenMaya.MScriptUtil().asDoublePtr()
        closestSurfacePoint = mfnNurb.closestPoint(forecastPosition,uIP,vIP)

        overshoot = OpenMaya.MVector(forecastPosition-closestSurfacePoint).length()

        u = OpenMaya.MScriptUtil().getDouble(uIP)
        v = OpenMaya.MScriptUtil().getDouble(vIP)

        uvPerpend = mfnNurb.normal(u,v)
        …
        …
        …
        if(overshoot <= groundOffset):
            outputForce.append(self._surfaceFollowHelper(block, uvPerpend,
                               velocities[i]))
        else:
            desiredVel = OpenMaya.MVector(closestSurfacePoint - positions[i])
            outputForce.append(self._surfaceFollowHelper(block, desiredVel,
                               velocities[i]))
```

3.3.7 Path Following Force

A path following force moves a particle along a series of waypoints that define a path. The particle's desired velocity is arrived at using the same method applied in the seeking and arriving behaviors, although here, there are sequential targets defining a path, and every arrival to a waypoint is immediately followed by a seeking or arriving behavior to the next waypoint in the sequence. The path can either be open or closed (see figure 3.6); in an open path, the particle decelerates as it gets closer to the final waypoint and eventually stops, while in a closed path the particle heads back to the first waypoint after reaching the final waypoint and starts over again.

There are a number of useful applications for the path following force. These include guards patrolling secure areas on a map, creatures avoiding natural obstacles in an open terrain, or cars navigating a difficult path [Buck05]. The implementation steps are as follows:

Step 1: Set the current waypoint to the first waypoint in the list.

Step 2: Steer the particle towards the current waypoint using the seeking behavior until the particle is within the waypoint proximity radius.

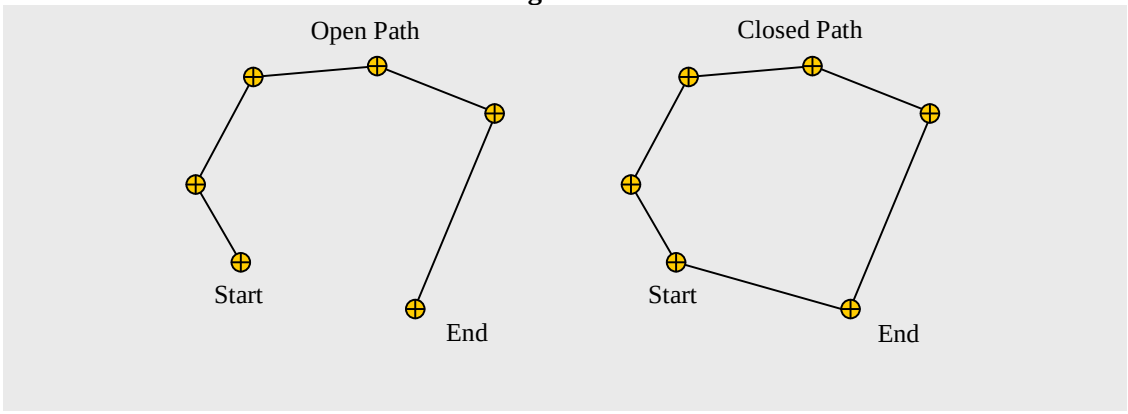
Step 3: Set the current waypoint to next waypoint in the list, continue *Step 2* until the final waypoint is reached.

Step 4: If it is an open path, steer towards the final waypoint using the arriving behavior. Otherwise, steer towards the final waypoint using the seeking behavior method and jump to *Step 1*.

A portion of the path following code shown below:

```
def _pathFollow(…)
…
…
…

    numWayPoints = wayPoints.length();

    if numWayPoints == 0:
        return

    for i in range(positions.length()):

        if self._currentWayPoint[i] >= numWayPoints:
            if pathFollowLoopOn:
                self._currentWayPoint[i] = 0
            else:
                self._currentWayPoint[i] = int(numWayPoints - 1)

        wayPoint = wayPoints[self._currentWayPoint[i]]

        desiredVel = wayPoint - positions[i]
        distToWayPoint = desiredVel.length()
        steerFunc = self._seekHelper
        if(distToWayPoint<=wayPointProximityRadius):
            if self._currentWayPoint[i] < numWayPoints:
                self._currentWayPoint[i] = int(self._currentWayPoint[i] + 1)
            else:
                steerFunc = self._arriveHelper

        currentVel = velocities[i]
        steeringForce = steerFunc(block, desiredVel, currentVel)

        if leaderID>0:
            if i != leaderID:
                steeringForce = OpenMaya.MVector(0.0,0.0,0.0)

        outputForce.append(steeringForce)
```

Figure 3.6



3.3.8 Obstacle Avoidance Force

The obstacle avoidance force steers a particle away from objects in its path. The algorithm to detect and resolve collisions went through three iterations. The first iteration, the ray tracing solution, involved projecting rays onto an implicit sphere (where the sphere is a ColliderSphere node), and then testing for ray intersections. The second iteration, the Maya NURBS surface solution, involved projecting rays onto any arbitrary NURBS surface. This method made use of Maya's built-in function sets to calculate intersections. The third iteration, the Detection Capsule Solution, involved the use of a sphere-swept volume to compute the intersection between a capsule and a sphere (where the sphere is a ColliderSphere node). Each of these methods is discussed in more detail below.

3.3.8.1 Ray tracing solution

The ray tracing solution uses ray tracing elements for detecting collisions. The implementation resembles a light-weight ray tracing engine. In the scene, obstacles are approximated by implicit spheres. For each simulation step, the algorithm iterates through each particle, performing an intersection test for each ray fired against each obstacle. Each particle has three sensors which are used to derive the rays that are fired. A ray by definition has an origin and a direction. The primary ray is derived from the particle's position and velocity. The two secondary rays are derived from the primary ray. Each secondary ray is adjacent to the primary ray and is rotated along the Z axis by 45 and -45 degrees, respectively.

For each detected intersection, the intersection test routine returns the point of intersection and the normal vector at the point of intersection. The distance between the origin of the active ray and the point of intersection is used to test if the current point of intersection corresponds to the closest object. This test is required because the ray can intersect more than one object. Once the closest point of intersection has been determined, a steering force is computed using the surface normal, the active sensor tip point, and the intersection point.

Using the sensor tip point and the intersection point, a vector is constructed, referred to in this section as vector *A*. Vector *A* is used to determine if the particle's active sensor tip penetrates the obstacle by taking the dot product of vector *A* and the normal vector at the point of intersection. A zero or positive dot product indicates that the sensor tip is touching or penetrating the obstacle while a negative dot product indicates that the sensor tip is not close enough to the obstacle. If there is penetration, a steering force is deduced by creating a force in the direction of the normal vector at the point of intersection with a magnitude equivalent to the length of vector *A*. The magnitude is essentially an overshoot amount and therefore affects how fast the particle reacts. The normal vector describes the direction the surface is facing, and which direction to steer away from.

The ray tracing solution was rejected due to (i) slow running time resulting from the ray tracing calculations needed to test for intersections; and (ii) inaccurate collision detection, resulting from the bounding radius of the particle not being taken into account.

The slow running time is attributed to the computationally expensive operations executed in pure Python. In an ideal scenario all computationally expensive tasks would be off-loaded to efficient pre-compiled libraries that interface with Python. Refer to Appendix A for the complete source code.

3.3.8.2 Maya NURBS surface solution

This solution uses Maya's API to perform intersection tests. Unlike the previous solution, this method does not confine the user to use implicit surfaces as obstacles, which are defined in this iteration as any object that can be approximated by a NURBS shape. Instead, it allows the user to use an arbitrary shaped surface as an obstacle, as long as it is a NURBS surface. The execution speed is improved (as compared to the ray tracing solution) with this method because the underlying intersection function is bound to a Python Maya API function call (if the ray tracing solution described previously had off-loaded all expensive computations to a C++ library, the execution speed would have been considerable faster than using Python Maya API function calls to compute ray-NURBS intersections). In general, a ray-sphere intersection test is much cheaper to compute than ray-NURBS intersection test. For each simulation step, the algorithm performs a ray intersection test for each NURBS surface defined in the scene graph against each particle and each ray fired.

The rays fired by the particle are derived in the same manner as the previous method. When an intersection is detected, the distance is set to the closest intersection point, and the *u* and *v* values corresponding to the intersection point are stored for future determination of the surface normal. When the distance corresponds to the closest

intersection point, the algorithm checks if the distance is less than or equal to the length of the current sensor. This ascertains whether the tip of the senor is away from the surface or penetrating the surface. An overshoot factor is calculated by taking the difference between the active sensor and the current distance. The *u* and *v* values are passed as parameters to another Maya API function that returns a surface normal corresponding to the point of intersection. The resulting steering force is deduced by creating a force in the direction of the surface normal at the point of intersection with a magnitude equivalent to the overshoot.

The running time with this iteration is better than the previous iteration since the ray tracing computations are handled with optimized API calls. While there is greater flexibility in having arbitrary shapes as obstacles, the collisions have the same inaccuracies noted in the previous iteration. Refer to Appendix B for the complete source code.

3.3.8.3 Detection Capsule Solution

This method uses a capsule and sphere intersection test to maneuver the particle such that the detection capsule is always free of collisions. The particle's velocity controls the direction of the capsule. The radius of the capsule is equivalent to the particle's bounding radius, and the length of the capsule is proportional to the particle's speed. As the particle moves faster, the detection capsule gets longer, allowing it to detect collisions early in time and give the particle enough time to adjust its trajectory [Buck05].

Choosing a capsule as a bounding volume as opposed to a cylinder has some advantages. First, a cylinder intersection test is expensive to compute mathematically [Eric05]. A capsule, on the other hand, can be viewed either as a sphere-swept volume or a cylinder with the caps fitted with spheres [Eric05]. When the capsule is viewed as a sphere-swept volume, its calculation is simply the (squared) distance between two inner primitives, compared against the (squared) sum of the combined radii.

The following steps explain the algorithm used to compute the capsule to sphere intersection:

Step 1: The capsule is generalized to a line segment and the sphere is generalized to an arbitrary point in 3D space.

Step 2: By defining A and B as the endpoints of the line segment, and point C as a point in 3D space, the problem is reduced to determining the point D on AB that is closest to C. A viable strategy for determining the point D is to project point C onto the line passing through points A and B. There are three possibilities for how C projects onto AB (in finding the closest point to C):

C projects outside AB but is closest to A. Hence, A is closest to C, so D = A

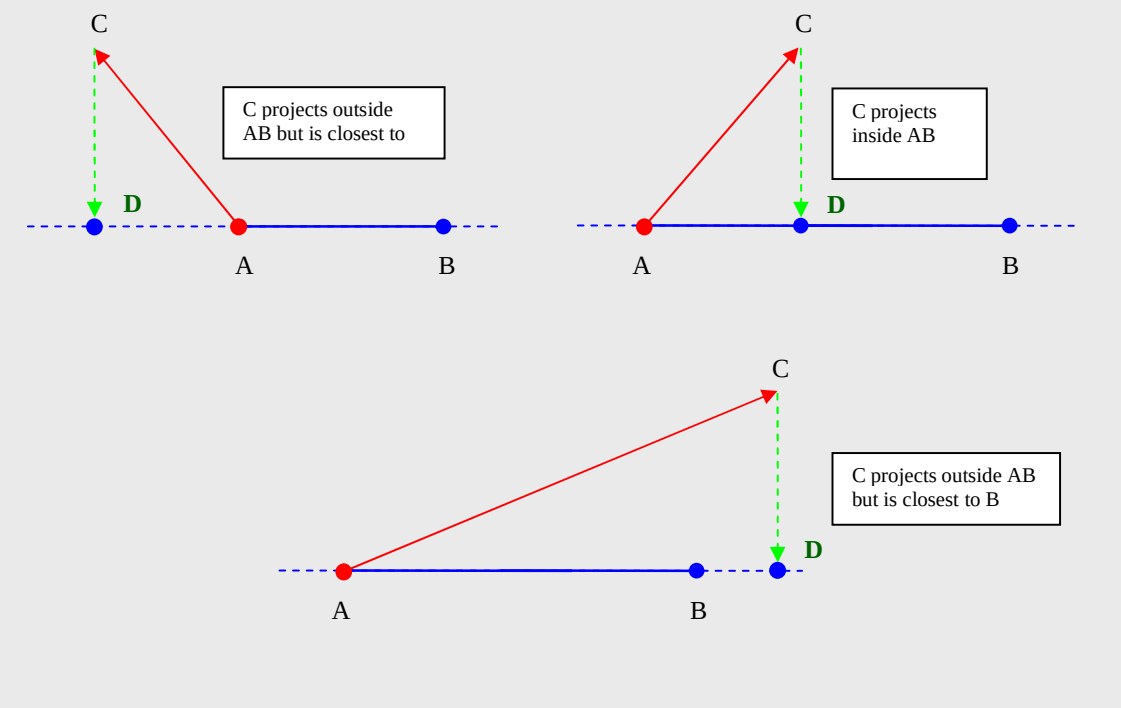
C projects inside AB, D = A + t(B-A), 0<=t<=1, t = (C-A) dot (B-A)/|(B-A)|^2

C projects outside AB but is closest to B. Hence, B is closest to C, so D = B

Figure 3.7 shows the three possibilities, outlined above. In the figure, line segment AB (in blue), vector AC (in red), and the dotted line (in green) correspond to C projecting onto line segment AB. The figure tells us that if the projection point is D, and D lies

within the line segment AB, then D is the closest point to C. If, however, D lies outside the line segment, then the endpoint closer to C is the closest point, rather than D [Eric05].

Figure 3.7



Step 3: The method applied to project point C onto line segment AB is streamlined by calculating the projection of one vector onto another. By constructing a vector Q from points A and B and vector P from points A and C, the following vector projection formula is applied to project P onto Q:

$$proj_Q P = \frac{P \bullet Q}{\|Q\|^2} Q$$

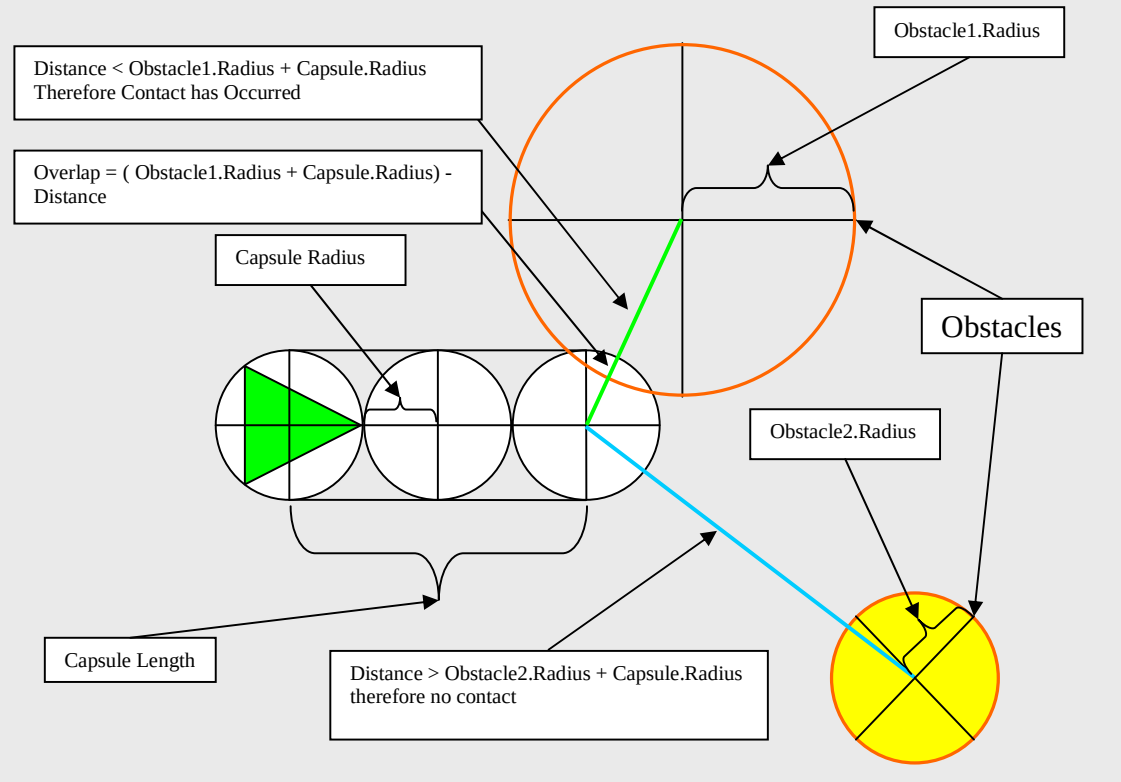
Given that any point on the line AB can be expressed as a parametric function, specifically as $P(t) = A + t(B - A)$, the vector projection equation is modified such that $t = \frac{P \bullet Q}{\|Q\|^2}$. The value of t is clamped to the interval $0 \le t \le 1$, which ensures that the closest point lies within the line segment. The value of D is obtained by substituting t into the parametric equation [Eric05]. The function was implemented as follows:

```
def ClosestPtPointSegment(self, c, a, b):

    ab = b - a
    ca = c - a
    ab_dot_ab = ab*ab

    # only compute closest point to segment when particle
    # is moving

    if ab_dot_ab > 0:
        t = (ca*ab)/ab_dot_ab
        unclamped_d = a + (ab*t)

        if t < 0.0:
            t = 0.0
        if t > 1.0:
           t = 1.0
        d = a + (ab*t)
        return t,d, unclamped_d

    # zero velocity

    else:
        return 0, a, a
```

Step 5: Once D is obtained, it is used to compute the distance *d* from D to C. The capsule and sphere only overlap if the distance *d* is less than or equal to the sum of their respective radii. In the implementation an unclamped point D is preserved for computing the lateral force. If penetration is detected, a lateral force and brake force are calculated, the sum of which determine the steering force. The lateral force steers the particle away from the obstacle, while the brake force steers the particle in the direction opposite its

velocity [Buck05]. These forces are then scaled by the amount of overlap between the detection capsule and the obstacle, enabling the particle to respond faster as it gets closer to the obstacle [Buck05]. An illustration of the capsule sphere intersection test is shown in figure 3.8.

Figure 3.8



The lateral direction vector is obtained by first finding a vector that is perpendicular to the line AB. The vector perpendicular to AB is then normalized and scaled by the amount of overlap, the resulting vector in the lateral direction vector. The following formula is used to find the vector perpendicular to the line AB.

$$perp_Q P = P - proj_Q P$$

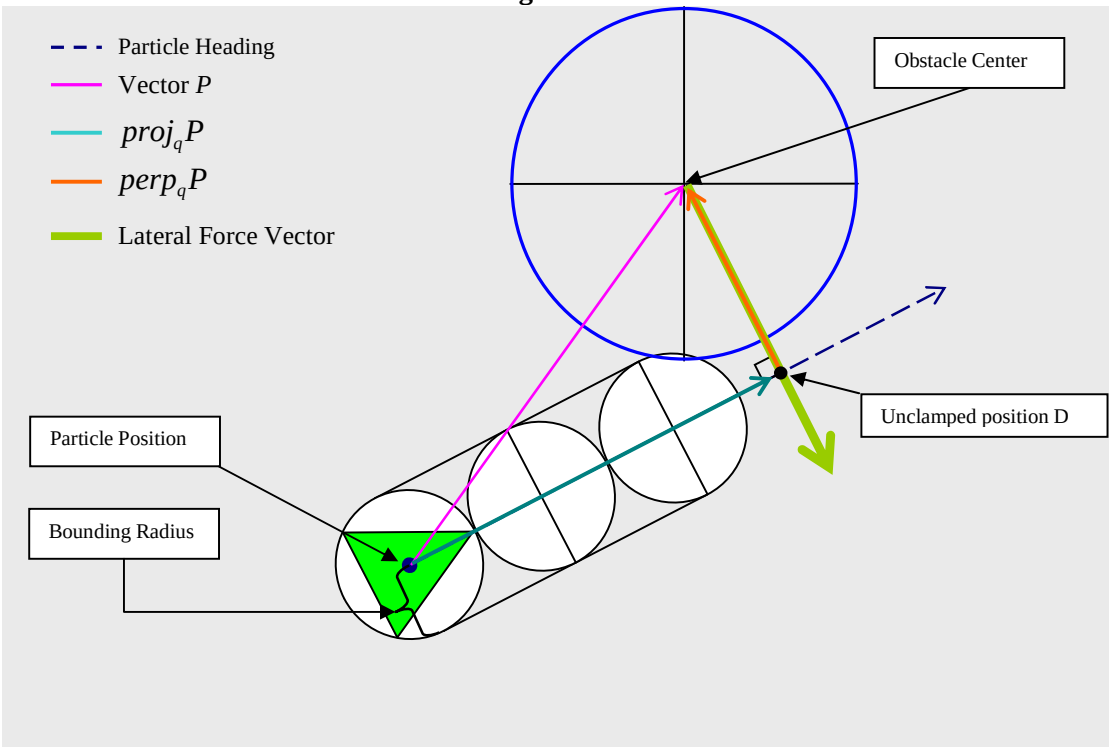
In the formula above, vector *P* is obtained by subtracting A from C, where A and C are the positions of the particle and the center of the obstacle, respectively. The vector $proj_Q P$ is obtain by subtracting A from the unclamped point D, where A is the center of the particle and D is the (unclamped) point intersecting the line passing through AB. The vector from A, the position of the particle, to D is always parallel to the particle's heading, while the vector from D to C is perpendicular to vector *AD* (refer to figure 3.9).

The resulting lateral force is deduced by creating a force in the direction of the lateral vector scaled by the overlap amount.. The overlap amount is calculated as follows:

Distance = (length(D-C)

Overlap =abs((capsule bounding radius + obstacle bounding radius) – Distance)

Figure 3.9



The implementation of the capsule to sphere collision test is shown below.

```
def SphereCapsuleIntersection(self, sc, sr, capsule):

    ca = capsule.a
    cb = capsule.b
    cr = capsule.radius

    t,d, ud = self.ClosestPtPointSegment(sc,ca,cb)

    dist =  (sc - d).length()

    radius = sr + cr

    if (dist  <= radius):
        overlap = math.fabs(radius - dist)
        return True, t, ud, overlap
    else:
        return False, -1.0, None,0
```

The brake force is derived by taking the inverse of the particle's velocity and multiplying it by the overlap amount. An extra brake multiplier is provider so the user can manually control the swiftness at which the particle brakes. The following calculations are used to derive the desired steering force:

*lateral force = (normalize(D-C) * -1.0) * overlap*

*brake force = velocity * -1.0 * overlap * braking weight*

steering force = brake force + lateral force

This iteration has several advantages to the previous two. The running time with this solution is faster because the intersection test is essentially performing a sphere-to-sphere intersection test. The accuracy is also improved because this iteration takes into account the bounding radius of the particle.

3.4 Group Forces

Group behaviors, like individual behaviors, are influenced by a particle's objectives and environment. However, group behaviors also take into account the existence of other particles, specifically other particles within the same particle shape [Buck05].

Neighbor tagging is a common characteristic of group behaviors. Prior to determining the group steering force, a particle considers other particles within a predefined circular area of the particle shape, sometimes referred to as its "neighborhood

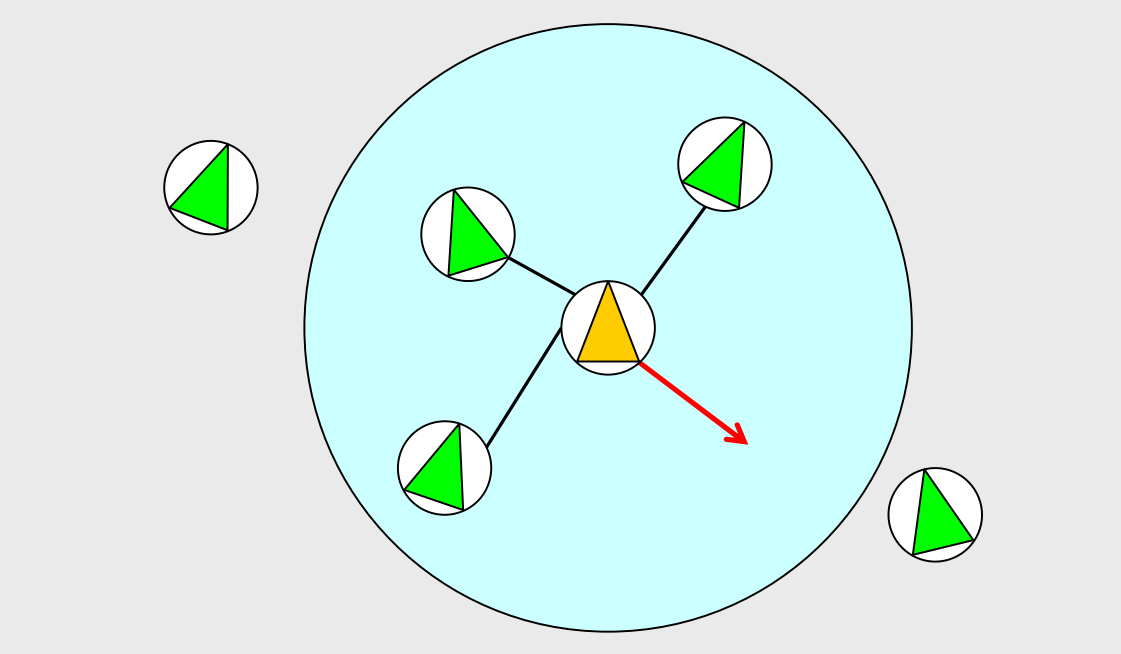
radius" [Buck05]. Those particles that are inside the neighborhood radius are stored in a list for further processing and later use.

The following section will require making a distinction between the particle for which a steering force is being calculated (the "current particle") and the particle's neighbors (the "neighbor" or "neighbors") that will be used in calculating the particle's steering force. All particles are assumed to be components of the same particle shape.

3.4.1 Separation Force

The separation force directs the current particle away from its immediate neighbors. The observed behavior can be described as particles "spreading out, each trying to maximize their distance from every other particle" [Buck05] (see figure 3.10).

Figure 3.10



The implementation of this force is explained in the following the steps:

Step 1: Find the neighbors of the current particle.

Step 2: For each neighbor compute a vector *V* by subtracting the position of neighboring particle from the position of the current particle. Obtain the distance from the neighbor to current particle by calculating the length of the vector *V*. Nomalize vector *V*.

Step 3: Scale the vector *V* inversely proportional to the current particle's distance from its neightbor and add it to the desired velocity accumulator. This operation will ensure that the force will be stronger when the distance between the two particles is small, and weaker when the distance is larger.

Step 4: After iterating through each neighbor, normalize the accumulated desired velocity and scale it by the current particle's maximum reachable speed.

Step 5: Compute the steering force by subtracting the current particle's velocity from the desire velocity. Repeat the steps for the next particle.

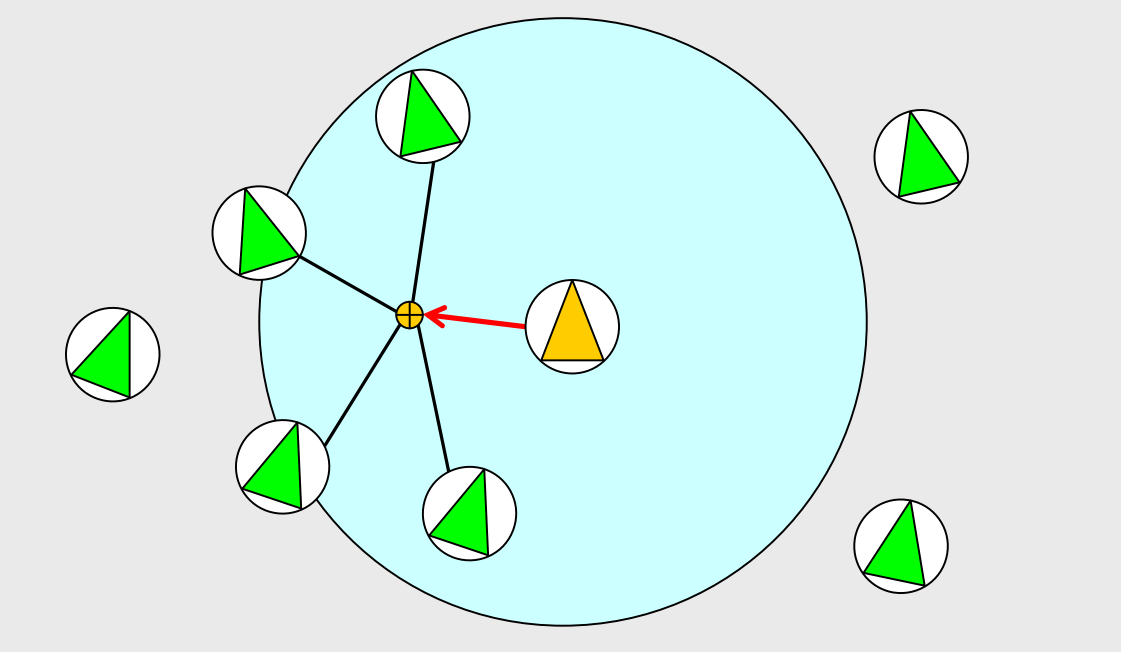
The code for separation behavior is presented below:

```
def _separate(self, block, positions, velocities, neighbors, outputForce):
    if positions.length() != velocities.length():
        return
    outputForce.clear()

    maxSpeed = self._attrDoubleValue(block, SteeringField.aMaxSpeed)
    maxForce = self._attrDoubleValue(block, SteeringField.aMaxForce)
    for i in neighbors.keys():
        neighborList = neighbors[i]
        pos = positions[i]
        desiredVel = OpenMaya.MVector(0,0,0)
        for j in neighborList:
            toVector = pos - positions[j]
            dist = toVector.length()
            if dist > 0.001:
                toVector.normalize()
                toVector = toVector * (1.0/dist)
                desiredVel = desiredVel + toVector

        steeringForce = OpenMaya.MVector(0,0,0)
        if desiredVel.length() > 0:
            desiredVel.normalize()
            desiredVel = desiredVel * maxSpeed
            steeringForce = desiredVel - velocities[i]

        steeringForce = self._limit(steeringForce, maxForce)
        outputForce.append(steeringForce)
```

3.4.2 Cohesion Force

The cohesion steering force directs the current particle towards the centroid of its immediate neighbors, where centroid refers to the average position (see figure 3.11). The centroid is used as a target for the seeking force which will steer the current particle towards it. The cohesion force results in the particles' appearing to stay together as a group.

Figure 3.11



The implementation of this behavior is explained in the following the steps:

Step 1: Find the neighbors of the current particle.

Step 2: Obtain the centroid by summing all the positions of neighbors and taking the average.

Step 3: Use the centroid as the target for the seeking behavior to obtain the steering force. Repeat the steps for the next particle.

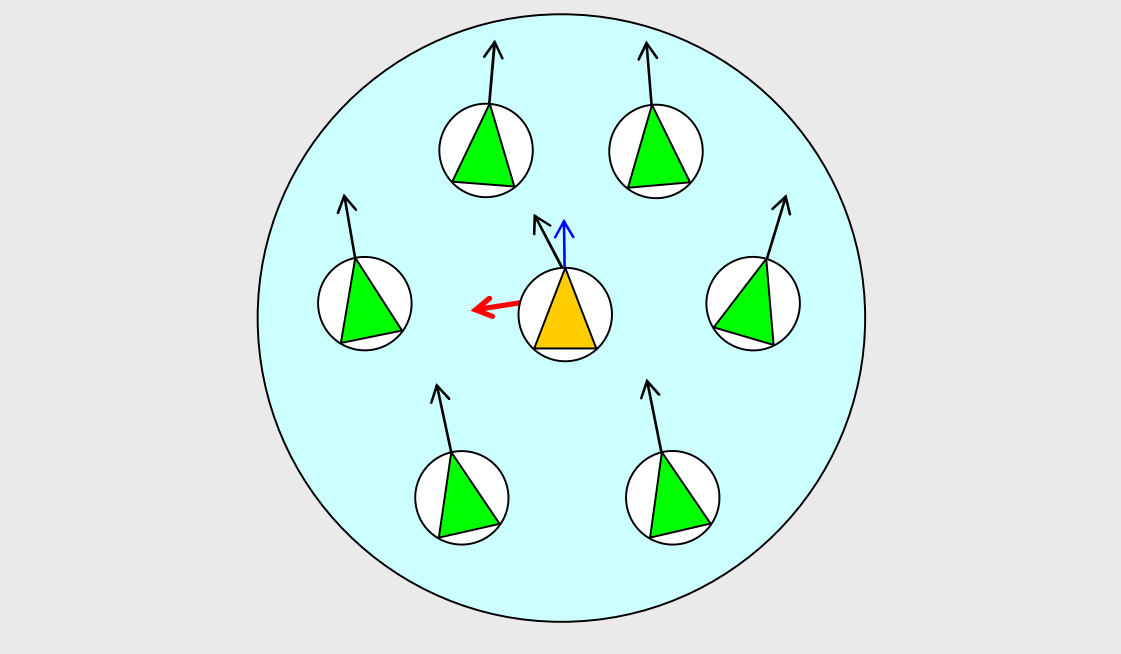
The code for the cohesion force is presented below:

```
def _cohesion(self,block, positions, velocities, neighbors, outputForce):
    if positions.length() != velocities.length():
        return
    outputForce.clear()

    for i in neighbors.keys():
        neighborList = neighbors[i]
        steeringForce = OpenMaya.MVector(0,0,0)
        centerOfMass = OpenMaya.MVector(0,0,0)
        for j in neighborList:
            centerOfMass = centerOfMass + positions[j]

        count = len(neighborList)
        if count > 0:
            centerOfMass = centerOfMass * (1.0/count)
            desiredVel = centerOfMass - positions[i]
            steeringForce = self._seekHelper(block, desiredVel, velocities[i])

        outputForce.append(steeringForce)
```

3.4.3 Alignment Force

The alignment behavior steers the current particle such that its heading aligns with its neighbors' (see figure 3.12). Application of the alignment force results in the particles trying to maintain the same heading as they move together.

Figure 3.12



The implementation of this behavior is explained in the steps outlined below.

Step 1: Find the neighbors of the current particle.

Step 2: Obtain the desired heading vector by summing all the heading vectors of the neighbors and taking the average.

Step 3: After iterating through each neighbor, normalize the desired heading vector and scale it by the current particle's maximum reachable speed.

Step 4: Compute the steering force by subtracting the current particle's velocity from the desired heading vector. Repeat the steps for the next particle.

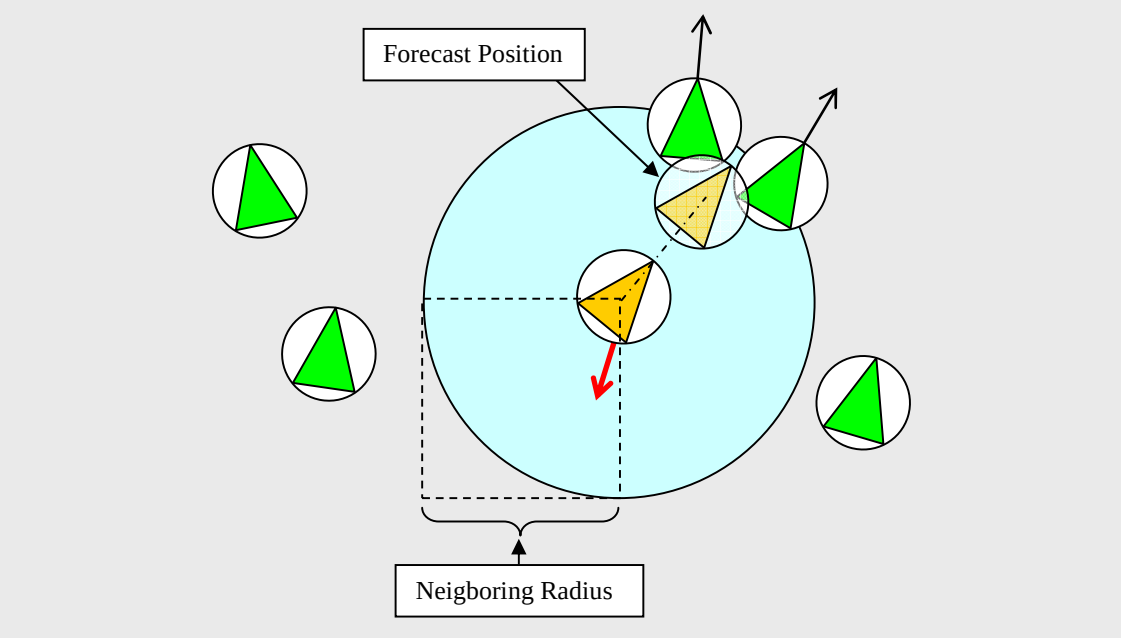
The code for alignment behavior is included, below:

```
def _alignment(self, block, positions, velocities, neighbors, outputForce):
    if positions.length() != velocities.length():
        return
    outputForce.clear()

    maxSpeed = self._attrDoubleValue(block, SteeringField.aMaxSpeed)
    maxForce = self._attrDoubleValue(block, SteeringField.aMaxForce)
    for i in neighbors.keys():
        neighborList = neighbors[i]
        averageHeading = OpenMaya.MVector(0,0,0)
        for j in neighborList:
            averageHeading = averageHeading + velocities[j].normal()

        count = len(neighborList)
        if count > 0:
            averageHeading = averageHeading * (1.0/count)

        if averageHeading.length() > 0:
            averageHeading.normalize()
            averageHeading = averageHeading * maxSpeed
            averageHeading = averageHeading - velocities[i]
        averageHeading = self._limit(averageHeading, maxForce)
        outputForce.append(averageHeading)
```

3.4.4 Neighbor Repulsion

When combining steering forces (applied to groups of particles), the particles often overlap. One might think that the separation force would always be sufficient to prevent overlapping, but sometimes it is not. The repulsion force helps minimize this overlapping by extrapolating the future position of the current particle to determine if any contact will occur in the next simulation step. If a contact is spotted, the resulting steering force will try to resolve the collision by slowing down and steering the current particle away from its neighbors. In most situations the repulsion force will prevent overlapping as long as the current particle's future position is forecasted by its velocity rather than its bounding radius. The modes of forecasting the current particle's future position can be toggled on and off by the user. Figure 3.13 depicts the repulsive force.

Figure 3.13



The formula for forecasting by velocity ensures that the forecast position is proportional to the current particle's speed (i.e., the faster the particle moves, the farther the forecast position). By contrast, the formula for forecasting by bounding radius ensures that the forecast position is constrained to the length of the bounding radius and is always parallel to the current particle's heading. The alternative forecasts are presented below:

forecast Position = *current particle* position + *current particle* velocity

forecast Position = *current particle* position +
normalize(*current particle* velocity)*bounding radius

The implementation of this force is explained in the following the steps:

Step 1: Find the neighbors of the current particle.

Step 2: Extrapolate the future position of the current particle and determine if it overlaps with any of its neighbors. The following formulas are used to find the overlap amount:

desired *Velocity* = *current particle* forecast position - *neighbor* particle position

distance = length(desired *Velocity*)

overlap = 2 * bounding radius – distance

The vector from the *neighbor* to the *current particle* is needed in order to obtain the distance between the two particles. Because the bounding radius is defined globally, all particles share the same radius. In the implementation, multiplying the bounding radius by two is the same as summing the radii of the two particles. A future release of the plug-in will allow a user to define the bounding radius per particle.

Having a per particle bounding radius would change the overlap formula to the following:

overlap = (*current particle* bounding radius +
neighbor particle bounding radius) - distance

Step3: If an overlap is spotted, normalize the desired *Velocity*, scale it by the overlap amount, and add it to the desired *Velocity* accumulator.

Step 4: Compute an arriving force using the accumulated desired *Velocity*. The resulting arrival force is the steering force for the *current particle*. Repeat the steps for the next particle.

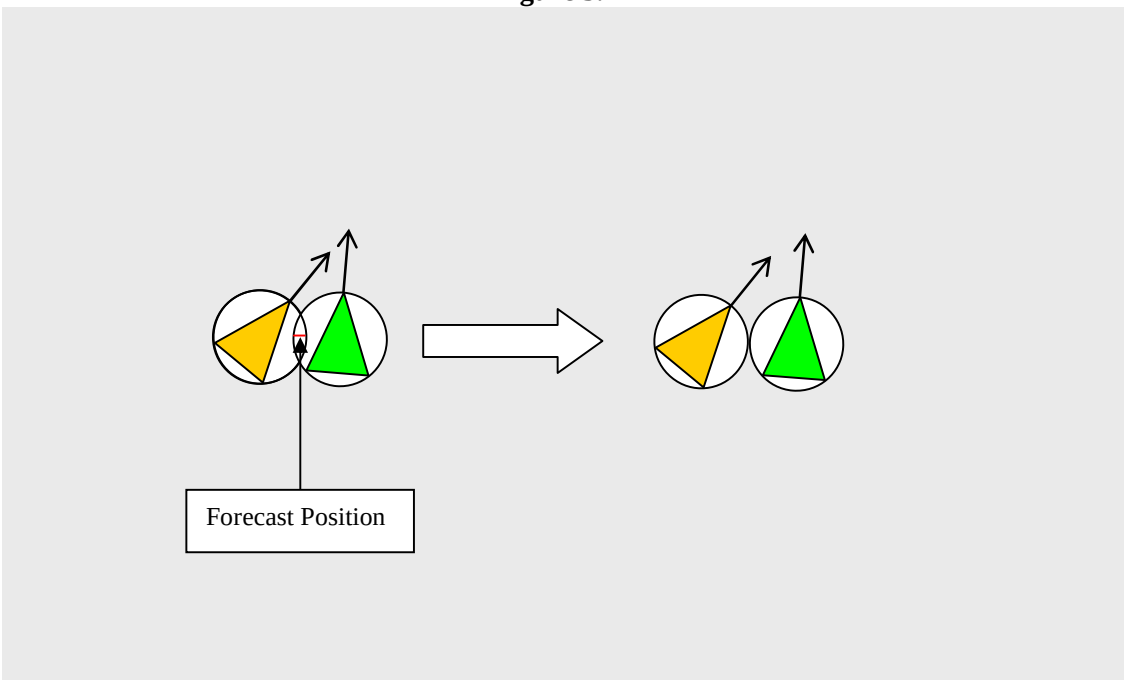
A portion of code for the repulsion force is presented below:

```
def _neighborRepulseCurrentOnly(…)
…
…
…

    for i in boidList:
        curBoid = positions[i]
        desiredVel = OpenMaya.MVector(0.0,0.0,0.0)
        neighborsList = neighbors[i]

        unitVelocity = OpenMaya.MVector(velocities[i])
        unitVelocity.normalize()

        # perdict future position of current boid
        if(forecastBySpeedOn):
            forecastPosition  = OpenMaya.MVector(curBoid + velocities[i])
        else:
            forecastPosition  = OpenMaya.MVector(curBoid + unitVelocity*bRadius)
        for j in neighborsList:
            if i != j:
                otherBoidPosition = positions[j]
                toBoid = forecastPosition  - otherBoidPosition
                distFromEachOther = toBoid.length()
                overlap = 2*bRadius - distFromEachOther

                if overlap >= 0:
                    toBoid.normalize()
                    desiredVel = desiredVel + (toBoid)* overlap

        # average desired velocity
        if leaderID>0 and i == leaderID:
            outputForce.append(OpenMaya.MVector(0.0,0.0,0.0))
        else:
            outputForce.append(self._arriveHelper(block, desiredVel, velocities[i]))
```

When combined with the zero overlap constraint (see Figure 3.14), the repulsion force completely prevents the overlapping of particles by moving the particle a distance equivalent to the overlap amount. However, because the zero overlap constraint is a non-physical constraint (does not account for mass, velocity, etc) there is the possibility of a

sliding effect, where the particle appears to be sliding on ice. A zero overlap contributing factor is exposed to the user, which allows the necessary tweaking to mitigate the potential sliding effect.

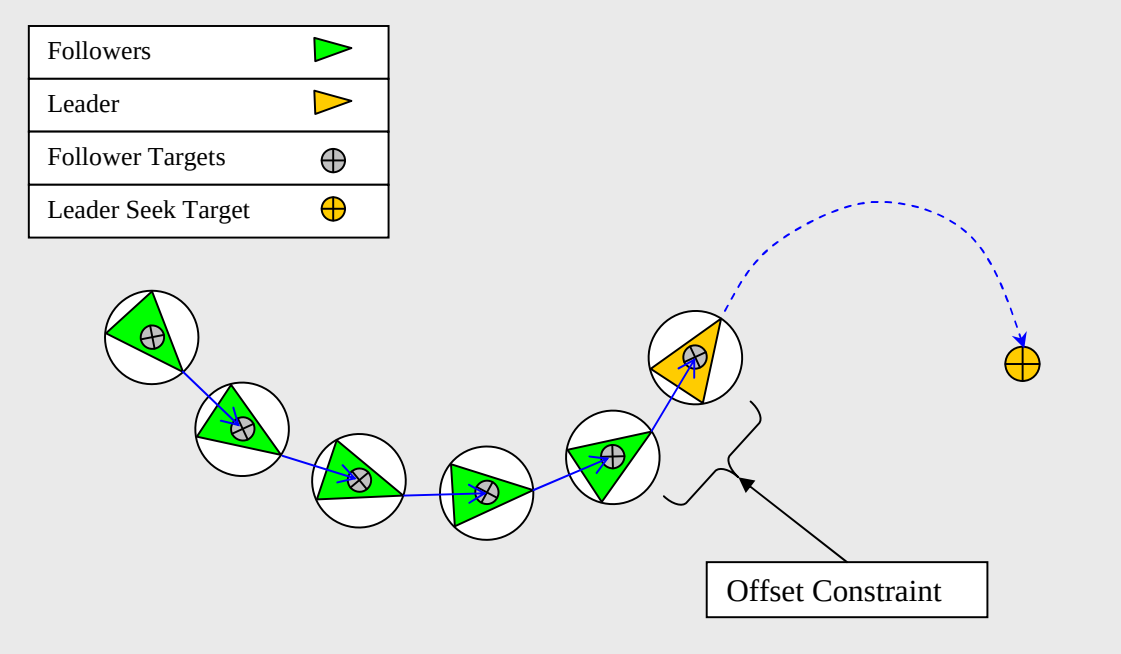
Figure 3.14



3.4.5 Leader Following Force with Offset Constraint

The leader following force steers a group of particles to move one after the other, in single file, behind a leading particle. The leading particle will usually be driven by one or more of the individual steering forces, whereas the follower particle will be driven by a force akin to the arriving behavior, moving each follower particle towards the particle in front of it (see figure 3.15). An offset constraint is used to keep the follower particles apart from each other.

Figure 3.15



The implementation of this behavior is explained in the following the steps:

Step 1: Set the current leader and current follower id's. Specifically, set (a) the current leader id to the id of the next leading particle; and (b) the current follower id to the id of the next particle in the list (but, if the id is equal to the id of the leading particle, then skip to the next particle).

Step 2: Compute a vector *V* from the position of the current leader to the position of current follower and normalize it. Vector *V* defines the heading for the follower with no offset constraint.

Step 3: Obtain the length of the user defined offset vector. Then, compensate for the current leader particle's bounding radius and the current follower's bounding radius by adding them to the offset length.

Step 4: Set the offset vector to vector *V* and scale it by the offset length computed in Step 3. This offset vector is now pointing in the direction of vector *V* and has a magnitude equal to the offset length computed in Step 3.

Step 5: Obtain the world space offset position by subtracting the offset vector from the current leading particle's position. This vector represents the desired target. Then, determine the distance from the current follower's position to the world space offset.

Step 6: Adjust the desired target to compensate for the fact that the current leader is in motion. Specifically, obtain a look-ahead time that is (i) proportional to the distance between the current leading particle and the current follower; and (ii) inversely proportional to the sum of the current leader and current follower's speeds.

Step 7: Scale the current leader's velocity by the look-ahead time. Then, add this vector to the world space offset position to obtain the adjusted target.

Step 8: Apply a modified version of the arriving behavior, using the adjusted target as the destination target. Refer to the code in the next page.

Step 9: Repeat Steps 1-8 for the subsequent particles (i.e., applying the steps to the next "current leader" and "current follower").

A portion of code for the leader following force is presented below:

```
def _followLeader(…)
…
…
…
    def doSteer(leaderID, followerID):

        currentVel = velocities[followerID]
        leaderVelocity = velocities[leaderID]
        leaderSpeed = leaderVelocity.length()
        dist = (followLeaderOffsetV.length() + 2*bRadius)

        if(followLeaderMaintainOffsetOn):
            toDir =  positions[leaderID] - positions[followerID]
            toDir.normalize()
            offsetV = toDir*dist

            worldOffsetPos = positions[leaderID] - offsetV
            ToOffset = worldOffsetPos - positions[followerID]
            lookAheadTime = ToOffset.length()/(maxSpeed  + leaderSpeed)
            target = worldOffsetPos + leaderVelocity*lookAheadTime
            desiredVel = target - positions[followerID]

            steeringForce = OpenMaya.MVector(0.0,0.0,0.0)
            arriveProximityRadius = self._attrDoubleValue(block,
                                            SteeringField.aArriveProximityRadius)
            decelerationMult = self._attrDoubleValue(block,
                                            SteeringField.aDecelerationMult)

            speed = min(lookAheadTime, maxSpeed)

            lookAheadTime = math.fabs(lookAheadTime)
            dir =  positions[leaderID] - positions[followerID]
            dist = dir.length()
            if dist > arriveProximityRadius:
                desiredVel.normalize()
                speed = leaderSpeed + (dist * lookAheadTime)

                desiredVel = desiredVel * speed

                steeringForce = (desiredVel- currentVel)
                steeringForce = self._limit(steeringForce, maxForce)
        …
        …
        …
    leaderID = self.getValidLeaderID(leaderID, numBoids)

    if leaderID < 0:
        leaderID = 0

    nextLeader = leaderID
    for i in range(numBoids):
        if i != leaderID:
            steeringForce = doSteer(nextLeader,i)
            outputForce.append(steeringForce)
            nextLeader = i
        else:
            outputForce.append(OpenMaya.MVector(0.0,0.0,0.0))
```

3.5 Computing Orientation for Particle Instancing

By default a Maya particle is just a point in space and does not possess an orientation. For simulating natural phenomena, such as smoke and fire, orientation information is not necessary. However, orientation becomes important when replacing a particle with an object. Maya's geometric instancing replaces each particle with instanced geometry. The movement of the instanced geometry is driven by the particle. With geometric instancing, all the particles share the same geometry, and thus, any change made to the original geometry will transfer to the instanced geometry.

It is important not to confuse particle instancing with a "behavioral animation system or flocking system" [Duve04]. If for example, your goal was to simulate a school of fish, particle instancing would not give you the desired behavior. In this scenario, "each fish just follows its own particle" completely oblivious of the other fish and the environment [Duve04].

Particle instancing lets you choose one of three inputs for setting the orientation of the instanced objects: *Rotation*, *AimDirection*, and *AimPosition*. The preferred method chosen (in the implementation) is the *Rotation* input because rotation values define the instanced geometry's orientation.

In the implementation, for each simulation step, the plug-in internally computes orientation vectors for each particle. Then for each particle a transformation matrix is constructed using the orientation vectors. Maya API calls are made to generate Euler rotations from the transformation matrix. The rotation angles, which are in radians, are converted to degrees. If the particle shape has a per particle rotation attribute defined, the

plug-in automatically populates it with the appropriate rotations. The instanced geometry's orientation is then synced with the new rotation values. However, in order for the rotations to work, the user first has to add a per particle vector attribute called "*rotationPP*" to the particle shape, and the particle instancer's *Rotation* input has to be set to the particle shape's *rotationPP* attribute.

Below is the pseudo code for computing the orientation vectors and Euler rotations for particles moving along a surface:

```
Create orientation vectors by calculating cross products

X = particle's normalized velocity
Y = X × SurfaceNormal
Z = Y × X

M = | X_x  X_y  X_z  0.0 |
    | Y_x  Y_y  Y_z  0.0 |
    | Z_x  Z_y  Z_z  0.0 |
    | 0.0  0.0  0.0  1.0 |

mtm = OpenMaya.MTransformationMatrix( M )
euler rotations = mtm.eulerRotation()
rotation vector = (euler rotations).asVector()

Convert angles in rotation vector from radians to degree
```

Below is the pseudo code for computing the orientation vectors and Euler rotations for particles moving in space:

Create orientation vectors by calculating cross products

$UpVector =$ Maya's scene up axis
$X =$ particle's normalized velocity
$Y = X \times UpVector$
$Z = Y \times X$

$$M = \begin{vmatrix} X_x & X_y & X_z & 0.0 \\ Y_x & Y_y & Y_z & 0.0 \\ Z_x & Z_y & Z_z & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{vmatrix}$$

```
mtm = OpenMaya.MTransformationMatrix( M )
euler rotations = mtm.eulerRotation()
rotation vector = (euler rotations).asVector()
```

Convert angles in rotation vector from radians to degree

3.6 Steering Force Accumulator

The D'Alembert principle suggests that a set of forces acting on an object may be replaced by a single force [Mill07]. This single force is calculated by adding the forces together using vector addition. Similarly, when multiple steering forces are applied to a single particle, the resultant force applied is calculated by adding each of the steering forces (but the sum is clamped to ensure it does not exceed the maximum steering force). In the SteeringField plug-in, the maximum steering force is a controllable global parameter that defines the maximum allowable force a particle can produce to propel itself. In principle, the magnitude of the resultant steering force is constrained by the

maximum steering force. In other words, if the resultant steering force is greater than the maximum force, the resultant force is clamped to the maximum force.

The SteeringField plug-in implements two force combination methods: weighted truncated sum and weighted truncated running sum with prioritization. The weighted truncated sum multiplies each steering force with a weight and adds it to a force accumulator. The resultant force is then clamped to the maximum allowable steering force. One drawback to this method of implementation is that every active force is calculated on every simulation step (even forces that have minimal effect on the resultant steering force). Another drawback to this method arises in situations where the user has to tweak the weights to get the desired "look." This balancing of weights can be difficult at times.

The weighted truncated running sum with prioritization method differs from the weighted truncated sum in that the steering forces are prioritized and processed in some order. The prioritization scheme ensures that those steering forces considered to be of importance have precedence over the others. In addition to prioritizing the steering forces, this method manages a running total of the forces applied to a particle. Prior to adding a new force to the running total, the algorithm first checks if there is any surplus force remaining, (i.e., maximum allowable force subtracted by current running total). If no surplus is remaining, the new force is disregarded. If there is surplus remaining, the new force is added to the running total [Buck05]. But if the magnitude of the new force exceeds the available surplus, the new force is truncated, such that its magnitude is equal the amount of the surplus remaining [Buck05].

3.7 Limitations

The SteeringField plug-in has certain limitations in its current state. The algorithms discussed in this paper do not factor the mass of each particle. Thus, if a user specifies a per particle mass attribute, the SteeringField plug-in will ignore it. Additionally, using a per particle radius attribute to define the bounding radius for each particle is not supported. At present, only the SteeringField's "boundRadius" attribute is supported because the underlying steering computations internally have access to the boundRadius attribute and not the particle's radius attribute. Because of this limitation, self collisions across two different particle shapes require that the user set the boundRadius to the greater radius of the two.

The function that determines the resultant force from all active forces does not use Reynolds' prioritized dithering, which is a priority based algorithm for evaluating steering forces that is dependent on preset probabilities (to determine which active behaviors contribute to the ultimate steering force). Instead the plug-in supports two methods, the weighted truncated sum method and the weighted truncated running sum with prioritization method. Reynolds' prioritized dithering may have certain advantages, including that it uses less CPU time than the other methods (at the expense of accuracy).

The special purpose simulation box is an axis-aligned box that may be scaled and translated. Although the simulation box may be rotated through Maya's direct manipulation controls, rotations are not supported by the plug-in. Modifying the box to allow rotation may increase its utility.

Chapter 4

Application and Results

4.1 Overview

The prior section described each of the forces introduced in the plug-in, as well as the ways they were implemented. Rather than presenting the results and applications for each of the forces previously discussed, this section will provide an overview of how the various forces can be commissioned to model complex behaviors in the context of Maya's particle system, (e.g. applying the steering forces in tandem with the particle system's built-in forces). After all, this is what makes the plug-in more flexible and useful than a stand-alone tool.

There are three primary effects that were created using the SteeringField plug-in: (i) the "swarming butterflies effect"; (ii) a school of starving fish targeting a piece of bait while avoiding a shark; and (iii) a fleet of vehicles that changes formations, while moving on an arbitrarily shaped terrain.

4.2 Swarming Butterflies

The first application of the steering field involves simulating the behavior of a swarm of butterflies, such that the resulting movements appear natural and spontaneous. This type of group behavior is achieved by combining the wandering, cohesion, separation, and repulsion steering forces. A volume bounded force is used in the simulation to limit the area in which the butterflies can steer.

The swarming butterflies effect captures the influence of external forces, such as wind, which result in subtle movements followed by abrupt distortions. Flight path metrics were used to judge the validity of the movements. These metrics showed stochastic curves with free-flowing arcs, a characteristic of the movements of butterflies observed in nature.

A user can create the swarming butterflies effect using the steering field and Maya's particle instancing. First, create a cloud of 15 particles with a maximum radius 10 using Maya's Particle Tool. For visualization purposes, set the particle shape's render type to sphere. Next create a steering field by running the python script below.

```
import maya.cmds as cmds
if not cmds.pluginInfo('SteeringField.py', q=True, loaded=True):
    cmds.loadPlugin( 'SteeringField.py' );
cmds.createNode("spSteeringField", name="steeringField")
```

Attach the steering field to the particles using the Maya's dynamic relationship editor and set the follow attributes using the table 4.2-1.

Table 4.2-1

Basic Parameter	Repulsion (Enabled)	Wandering (Enabled)	Separation (Enabled)	Cohesion (Enabled)
Max Force 55.0 Max Speed 150.0 Bound Radius 4.0	Magnitude 1.0 Neighbor Distance 5.0	Magnitude 0.5 Wander Radius 4.0 Wander Distance 6.0 Wander Jitter 0.25	Magnitude 0.7 Neighbor Distance 10.0	Magnitude 0.9 Neighbor Distance 30

In order to prevent the particles from veering off too far from the image plane, the simulation box must be enabled. The simulation box is a volume bounded force that keeps the particles confined within the boundary of the box. Set the magnitude of the

simulation box to 2 and scale the box by 2 percent by setting the steering field's transform scale to (1.2, 1.2, 1.2). To simulate wind, attach a turbulence field to the particle shape. Set the magnitude of the turbulence field to 80 and the attenuation to 0.

The last step is to replace the particles with animated geometry. Using Maya's geometric modeling tools, create a basic butterfly with wings. Animate the wings so they flap. Replace the particles in the scene with the animated butterfly by using Maya's geometric instancing feature. When running the simulation again, the butterflies will match the movement of the particles, but will not exhibit any rotations. This is expected because particles by default do not have a local coordinate frame. The steering field addresses this problem by internally computing a local coordinate frame for each particle. The computed local coordinate frame is centered at the origin and is derived using the particle's velocity and world up vector. The steering field computes rotation values using the local coordinate frame and stores them in a vector array. If the particle shape has a *rotationPP* attribute, the steering field will automatically copy the rotation vector array into *rotationPP* attribute. Applying the rotations is then a matter of setting the particle shape's instancer *Rotation* attribute to the *rotationPP* attribute. If for any reason the geometry is displayed upside down, the vector must be flipped by enabling the flip up vector through the steering field's basic parameters.

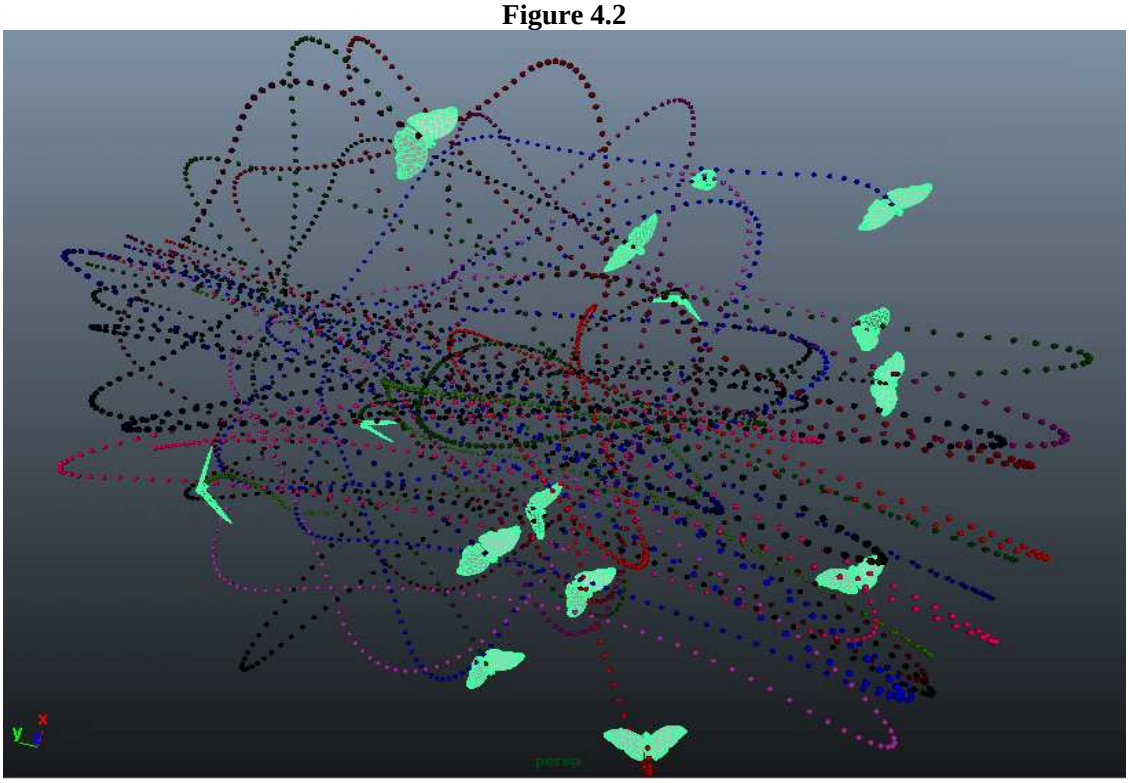
The scale of the simulation box can be animated to create the effect of imploding and discharging particles (expanding and contracting the cloud of particles). This is a fast and simple way to distort the movement of the particles. Since the simulation box by design attracts the particles, it can also be used as a track force. By animating the

translation values or attaching the simulation box to a motion path, the cloud of particles can be steered in a deterministic manner. The screen capture in figure 4.1 shows the swarming butterflies effect, and highlights the presence of the simulation box.

Figure 4.1

The screen capture in figure 4.2 records the motions of the various butterflies in the swarming butterflies effect:

Figure 4.2



4.3 Prey and Predator

The second effect involves simulating the behavior of a school of fish pursuing bait on a lure, while avoiding a wandering shark. This type of group behavior is achieved by combining the wandering, cohesion, separation, repulsion, arriving, seeking, and fleeing steering forces. In this effect, the school of fish will rush to the bait at every possible moment. When sensing danger the school of fish will immediately flee to safety. In some cases one or more fish will break from the group but eventually steer back to the group. As observable characteristic of the group behavior is the subtle and gentle

braking of each fish when they sense a neighbor or the bait. The sliding effect caused by the zero overlap contributes to the underwater look.

A user can create the seeking and fleeing school of fish effect using the SteeringField. First, create a grid of 11 particles using Maya's Particle Tool, which will represent the school of fish. Create another particle shape but with only one particle. This particle will represent the shark. For visualization purposes set all the particle shapes' render type to sphere.

Next, create three SteeringFields and attach one to the school of fish's particle shape, one to the shark's particle shape, and one to both particle shapes. The script provided below will do this.

```
import maya.cmds as cmds
if not cmds.pluginInfo('SteeringField.py', q=True, loaded=True):
    cmds.loadPlugin( 'SteeringField.py' );

cmds.createNode("spSteeringField", name="schoolOfFishSteeringField")
cmds.createNode("spSteeringField", name="predatorSteeringField")
cmds.createNode("spSteeringField", name="repulseField")


cmds.connectDynamic("fishParticles", f=" schoolOfFishSteeringField")
cmds.connectDynamic("predatorParticles", f=" predatorSteeringField ")
cmds.connectDynamic("fishParticles ", f=" repulseField ")
cmds.connectDynamic("predatorParticles", f=" repulseField ")
```

Then, set the attributes for each SteeringField using the tables in 4.3-1:

Table 4.3-1

schoolOfFishSteeringField Attributes				
Basic Parameter	**Simulation Box (Enabled)**	**Arrive (Enabled)**	**Wandering (Enabled)**	**Separation (Enabled)**
Max Force 20.0 Max Speed 60.0 Bound Radius 3.0 Zero Overlap (Enabled) Zero Overlap Damping 0.1	Magnitude 5.0 Width 50.0 Height 50.0 Depth 50.0	Magnitude 2.0 Seek Targets (Lure)	Magnitude 0.3 Wander Radius 3.0 Wander Distance 5.0 Wander Jitter 2.0	Magnitude 0.5 Neighbor Distance 5.0
Cohesion (Enabled)	**Flee (Enabled)**			
Magnitude 1.0 Neighbor Distance 30.0 Flee Targets (Shark Locator)	Magnitude 4.0 Panic Distance 15.0			

predatorSteeringField Attributes			
Basic Parameter	**Simulation Box (Enabled)**	**Seek (Enabled)**	**Wandering (Enabled)**
Max Force 20 Max Speed 50.0 Bound Radius 3.0	Magnitude 2.0 Width 50.0 Height 50.0 Depth 50.0	Magnitude 0.3 Seek Targets (Lure Locator)	Magnitude 0.7 Wander Radius 3.0 Wander Distance 5.0 Wander Jitter 2.0

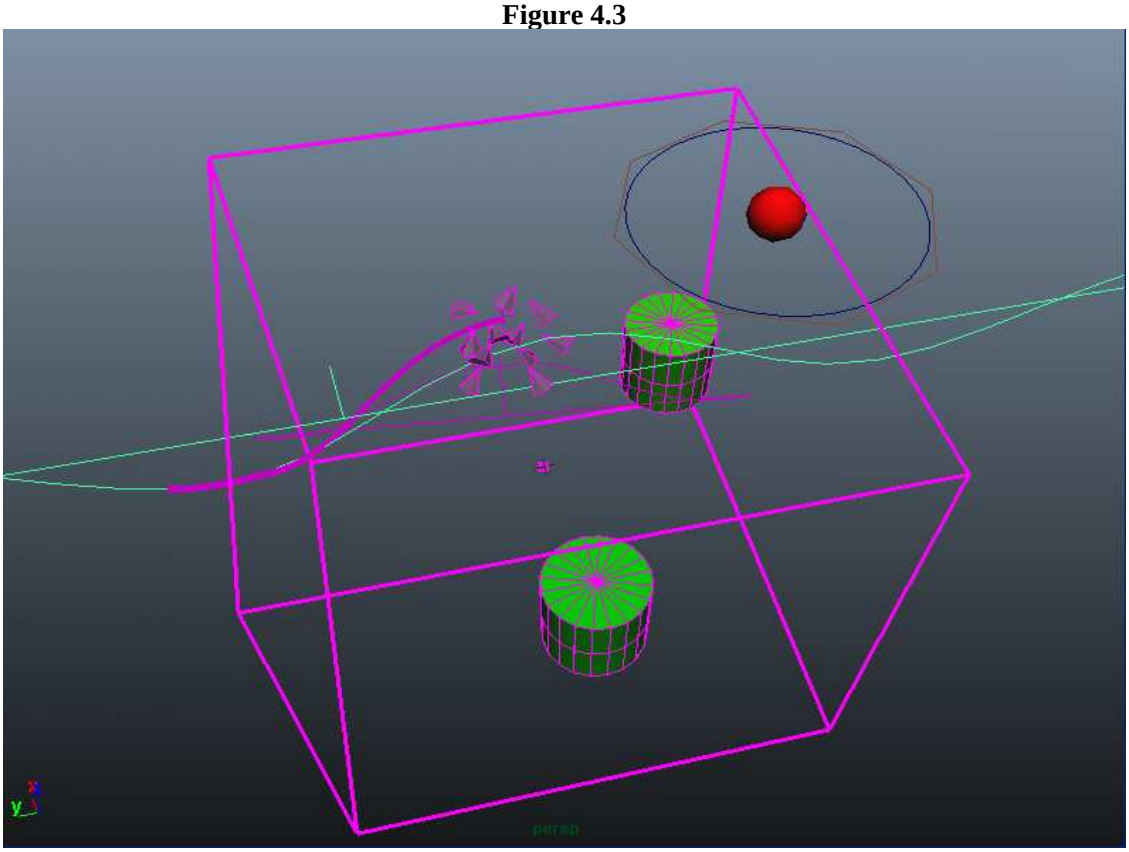
repulseField Attributes	
Basic Parameter	**Self Repulsion (Enabled)**
Max Force 25.0 Max Speed 50.0 Bound Radius 2.0	Magnitude 4.0 Forecast by Speed (Enabled) Neighbor Distance 15.0 Affect All Attached Particles (Enabled)

Each fish uses the arrive steering force to reach the lure whereas the shark uses a seek steering force. As discussed in Chapter 3, when a particle *seeks* a target, it will always overshoot the target and turn around, but when a particle *arrives* at a target it will come to a gentle stop. The arrive steering force was chosen in this setup because each

fish needs to gently halt as it reaches the bait to reflect that it has less momentum than the larger shark because of its smaller size. By contrast, the seek steering force allows the shark to overshoot the bait, reflecting that it is larger (and therefore has more momentum), than the fish. As in the previous example, the simulation box is used to define the boundary limits of the world with respect to the particles. Since both the fish and shark used simulation boxes it is important that their respective dimensions and transformations are in synchronized.

The seek and arrive forces require defining a target. First, create two locators: one representing the lure and the other representing the shark. Using Maya's connection editor, connect the sharkParticleShape *centroid* attribute to the sharkLocator *translate* attribute. This relationship will cause the movement of the particle to drive the movement of the locator. Next connect the sharkLocator *translate* attribute to the fish steering field's flee target attribute. This relationship will make the particles move away from the target when they are in the proximity (defined by the user) of the target. Finally, connect the lure locator *translate* attribute to the fish steering field's arrive target attribute. Animate the lure locator so it moves in a sinusoidal waveform using a nonlinear sine former. This will give the impression of the lure being underwater (refer to Maya's documentation for more information regarding nonlinear deformers). Once the simulation is complete, the particles can be replaced with geometry as described previously. The screenshot in figure 4.3 shows how the scene is setup.

Figure 4.3



4.4 Patrolling Fleet Grounded to Surface

This application of the SteeringField simulates a fleet of vehicles patrolling an arbitrary shaped NURBS surface. The vehicles form a line formation at the start of the simulation. For the duration of the simulation, the vehicles patrol a closed path while avoiding obstacles in the way. This type of group behavior is achieved by combining the following steering forces: path follow, surface follow, obstacle avoidance, and leader follow.

To setup this simulation, begin by creating a grid of 6 particles using Maya's Particle Tool. This group of particles will represent the fleet of vehicles. As in the previous examples, set the particle shape's render type to *sphere* for visualization

purposes. Next, create a steering field and attach it to the particle shape. The script provided below will do this.

```
import maya.cmds as cmds
if not cmds.pluginInfo('SteeringField.py', q=True, loaded=True):
    cmds.loadPlugin( 'SteeringField.py' );

cmds.createNode("spSteeringField", name="vehicleSteeringField")
cmds.connectDynamic("vehicleParticles", f=" vehicleSteeringField ")
```

The steering field requires the user to specify the NURBS surface the particle should gravitate towards. This requires setting a relationship between the NURBS surface shape node and steering field ground surface input. Since the particle will be making contact with a surface, it is important to match the particle shape's visual sphere radius to the steering field's *boundRadius* attribute. This will give a better visual cue to judge the behavior of the surface contact as the simulation is run. Specifically, it shows how the magnitude and ground offset parameters affect the gravitation of the particle to the surface. The commands below will establish these relationships.

```
cmds.connectAttr("vehicleSteeringField.boundRadius", "vehicleParticleShape.radius");
cmds.connectAttr("groundSurfaceNURBS.message ", "boidField. groundInputSurface ");
```

Set the attributes of the steering field using table 4.4-1.

Table 4.4-1

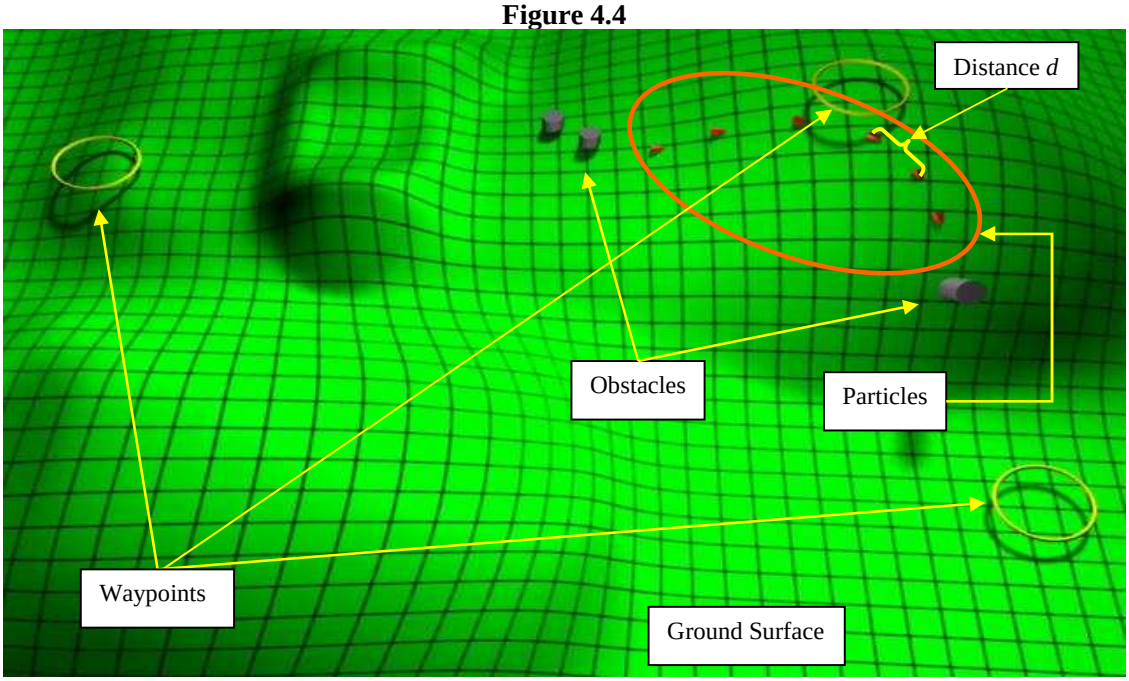
Basic Parameter	**Surface Follow (Enabled)**	**Follow Leader (Enabled)**	**Obstacle Avoidance (Enabled)**
Max Force 30.0 Max Speed 15.0 Bound Radius 3.0 Leader ID 1	Ground Offset 0.1 Forecast by Speed Off Forecast by Feeler Length 0.5 Magnitude 2.0 Uphill Multiplier 0.5 Downhill Multipler 1.0 Surface Shape Input (groundSurfaceNURBS)	Magnitude 1.0 Maintain Offset Enabled Offset (0,0,0) Arrive Proximity Radius 1.0	Magnitude 6.0 Proximity Brake Weight 0.75
Path Follow (Enabled)			
Magnitude 1.0 Waypoint Proximity Radius 1.0 Waypoints (user defined) Loop Enabled			

In order to get the path follow force to function, waypoints need to be defined. The best way to do this is to create Maya locators and position them so they form a path. Using Maya's connection editor, connect the translate attribute of each waypoint to the steering field's waypoints attribute. It is important to keep in mind that the ordering of the waypoints matter (the first waypoint corresponds to the starting target, intermediate waypoints correspond to subsequent targets, and the last waypoint corresponds to the final destination target). If looping is enabled, the particle will gravitate towards the starting waypoint once it has reached the last waypoint and start the path again.

Because the behavior requires that the particle line up in a single file, a leader particle must be specified by the user. This feature is made available through the keyable attribute *leader ID*. It is important to keep in mind that the leader can only control the particles that reside in the same particle shape as the leader. In other words, a leader belonging to one particle shape cannot act as the leader of another particle shape.

One observable disadvantage of using a force to ground a particle is the jittery motion produced along the surface normals. Because the jittering cannot easily be eliminated, the goal becomes balancing the contributing forces such that the jitter is less evident. However, for situations where the surface is supposed to represent a rough terrain, the jittering will provide realistic movement. An alternative to using a force to ground a particle is to have a shadow particle unaffected by forces that is constantly repositioned to match the position of the dynamic particle, and then projected to the closed point on the surface. The screen shot in figure 4.4 below identifies the obstacles, particles, waypoints, and ground surface in this setup.

Figure 4.4



An attribute is available to control the amount of distance *d* maintained between the particles in the line formation. This offset attribute is a vector whose length defines distance *d*.

Chapter 5

Conclusion and Future Work

This paper introduces a plug-in for Maya's particle system that is grounded in Reynolds' work on steering behaviors (as well as its various practical applications). The goal of the plug-in was to provide ready access to modeling a set of steering behaviors that integrates with and extends Maya's built-in functionality. This paper presents the steering forces implemented and describes three practical use cases, highlighting, for example, how built-in forces can work with the plug-in.

As with any scripting language, there is a trade off between speed of development and performance. The plug-in could have been written in C++ and yielded better performance in certain respects. However, the benefits of Python's scripting language overshadowed the performance gains from C++. One benefit of this introduction is that Python bridges Maya commands (formerly known as MEL commands) and Maya's API calls with the same script. Additionally, the time savings from not having to compile and link code allows more time for experimenting with different algorithms and testing functionality of software. Finally, there does not appear to be a Maya particle system plug-in (for modeling steering behaviors) written in Python that is publicly available.

In its current state, the plug-in's performance may be improved by writing performance critical functions such as the neighbor tagging function in C++ and writing Python C++ bindings to make the functions callable from Python. The process of creating Python C++ bindings entails writing wrapper functions for each function that needs to be accessible from Python and "compiling it into a C++ library that the Python

interpreter can dynamically load and use" [Pyth05]. Specifically, this implementation approach has Python performing all the high-level tasks and the C++ object(s) doing all the "heavy lifting". The performance of the group behaviors can also be improved by introducing a field of view constraint to the particle. This would restrict the number of particles included in the neighboring region. Specifically, only those particles that are inside the field of view would be considered for further processing.

5.1 Future Development

There are certain extensions that can be incorporated into the plug-in to make it more versatile. These enhancements include texture sampling sensor, grouping by color, curve targeting, scriptable steering behavior node, and spatial partitioning to increase efficiency of neighbor searches.

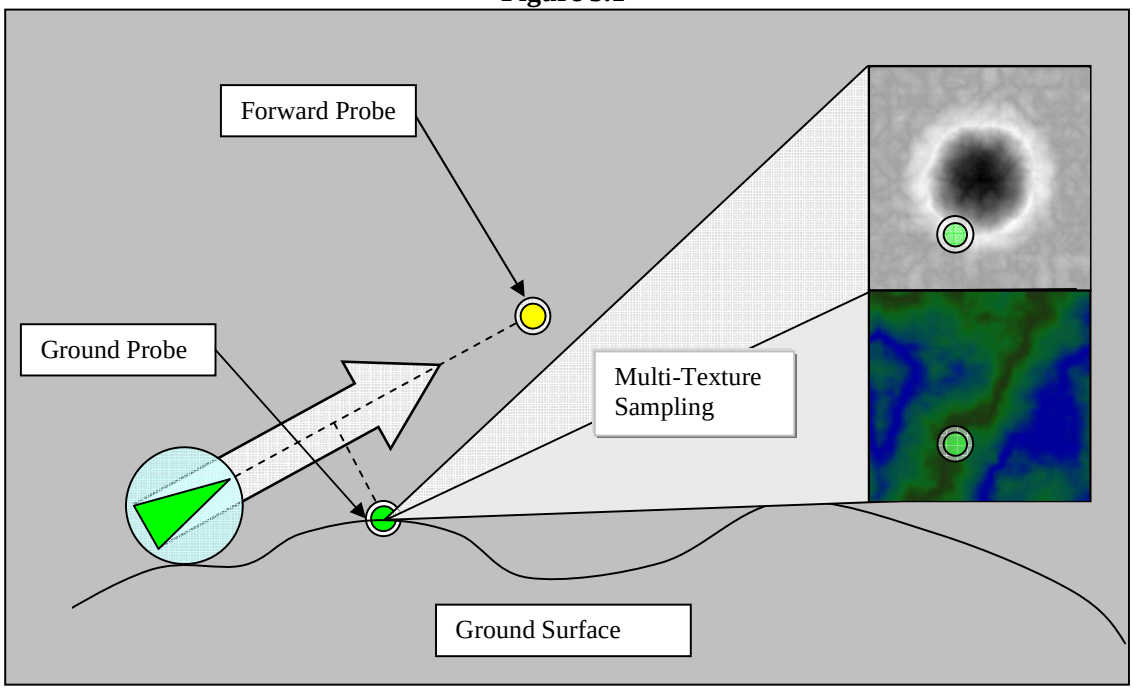
5.2 Texture Sampling Sensor

This extension would introduce a texture sampling sensor that would sample points from a surface and extract texture map data associated with the point samples. There are certain benefits to using texture maps. They can be individually crafted or procedurally generated. They can be viewed and modified interactively. And they can be animated, which would allow for interesting effects. The data retrieved from the texture would be processed and used to dictate the particle's next course of action. The types of data that can be represented on the texture maps are many. Examples may include, but are not limited to, friction maps, weight maps, flow field maps, blend shape maps, height

maps, and obstacle maps. Utilizing multilayer texture maps would add more variability to the usage of the system.

A practical application using this method is having a creature's animation cycle respond and adapt to the type of surface the creature is walking on. Imagine a terrain with mud piles, peaks, valleys, and water patches. These surface characteristics are represented as texture maps. As the creature is walking on the terrain, the sensor is actively probing the surface. The sensor consists of a forward probe and a ground probe. The forward probe would point in the direction of the creature's heading with a length adjustable by the user. The ground probe would aim towards the ground and would always be perpendicular to the forward probe. When the ground probe encounters a water patch, the creature's current animation cycle is smoothly blended with an animation cycle that exhibits characteristics of being in water. A normalize weight map on the surface determines the amount the new animation cycle blends with the current animation cycle. For example, a value of 1 would represent deep trenches; hence, the "in water" animation cycle would have complete effect. A value of 0 would represent shallow water, hence, the "in water" animation cycle would have minimal effect. Another example would be using a friction map to control the speed of the particle. One could imagine a terrain with ice patches and sticky asphalt. The illustration in figure 5.1 depicts the texture sampling sensors.

Figure 5.1



With obstacles, flow field maps could be used to guide a particle's direction. The obstacle avoidance algorithm could be modified such that the data retrieved from a texture would guide the path of the particle. For example, suppose there is a cylinder representing an obstacle textured with a flow field map representing vectors pointing east. In this scenario, upon detecting a collision, the forward probe would retrieve the flow vector from the cylinder's surface. When the steering force is calculated, the flow vector would be treated as the dominant contributing force. As a result, the particle would always steer east of the cylinder. The method would also guarantee a path that is deterministic.

5.3 Grouping by Color

The grouping by color extension would allow a user to divide particles within one particle shape node into groups. In the setup each particle would be assigned a color from a predefined set of colors. In the steering field, a new attribute would be added that would allow a user to add and associate a target with a color. Each group of particles would seek the associated target. Using this extension in combination with the seeking and other steering forces would enable the particles to move in distinct organized groups. By adding the ability to change the particles colors on-the-fly, the particle groups would break off groups and form new groups.

5.4 Curve Targeting

The curve targeting extension would enable the particles in a particle shape to target the shape of a curve. Each particle in the particle shape would be assigned a control vertex (CV) on the curve to target. The seeking or arriving forces would be used to influence the particles to move towards their respective CV target. If the shape of the curve were to change by animating the CVs of the curve, the particles would appear to morph into different shapes. Furthermore, if the curve were made dynamic, the motion of the curve would contribute to the resulting motion of the particles.

5.5 Scriptable Steering Behavior (SSB) Node

This extension would introduce a mechanism to allow a user to define custom steering behaviors in an interactive manner. Python's runtime dynamic capabilities make this possible because it allows a program to load on-the-fly components that have been

modified. The implementation would decouple the steering behaviors from the SteeringField, since now each steering behavior would reside inside an SSB node. The SSB node would expose to the user a virtual function, for which the user must implement directly through Maya's Attribute editor. The function signature's return value would be a steering force vector. Additionally, the function would have complete access to the SteeringField's attributes such as particle positions and velocities. By decoupling steering behaviors from the SteeringField, the SteeringField would be responsible for executing the implemented virtual function of each of the SSB nodes connected to it. For each executed SSB node, the SteeringField would accumulate the returned steering forces and employ one of the force combination methods described in section 3.6. Refer to appendix I for partial implementation of SSB node.

Pseudo Modification of SteeringField Node to Handle SSB Nodes

```
import new

class SteeringField(OpenMayaMPx.MPxFieldNode):
…
…
…

def _importCode(self, code, name):
    module = new.module(name)
    exec code in module.__dict__
    return module

def executeCode(code, block, position, velocity):
    import maya.OpenMaya as OpenMaya
    codeAccum = []
    codeAccum.append("import maya.OpenMaya as OpenMaya\n")
    codeAccum.append("def computeSteeringForce(block, position, velocity):\n")
    codeAccum.append("    ")
    codeAccum.append(code)
    flattenCode = "".join(codeAccum)
    self._validate(flattenCode)
    …
    …
    …
    m = self._importCode(flattenCode, "computeForceFunction")
    force = m.computeSteeringForce(block, position, velocity)
    return force

def _calculateForce(self, block, positions, velocities, forceArray):
…
…
…
    #Get all ssbNodes connected to self
    ssbNodeList = self._getAllConnectedSSBNodes()
    AccumForce = OpenMaya.MVector(0,0,0)
    for ssbNode in ssbNodeList:
        dependencyNode = OpenMaya.MFnDependencyNode(ssbNode)
        mplug = dependencyNode.findPlug("computeSteeringForce")
        force =  executeCode(mplug.asString(), block, positions[i], velocities[i])
        AccumForce = AccumForce + force
        …
        …
        …
```

5.6 Spatial Partitioning

The neighbor tagging algorithm used by group steering behaviors has an order of n^2 complexity, which means that as the number of particles increase, the time it takes to compare them increases in proportion to the square of their number [Buck05]. This limits the number of particles that can be simulated before experiencing a performance hit. As such, speed improvements can be made by incorporating spatial partitioning techniques such as kd trees, octrees, and quad-tree. For larger data sets, these algorithms will improve the neighbor tagging performance (even for small data sets, there is no real advantage) [Buck05].

APPENDICES

Appendix A

Obstacle Avoidance Ray-tracing Solution

```
# function that creates primary and secondary rays
def createFeelers(self, pos, vel, maxSpeed):
    feelers = []

    thisNode = self.thisMObject()
    plug = OpenMaya.MPlug(thisNode,SteeringNode.aFeelerLength)
    feelerLength = plug.asDouble();
    feelerLength =  feelerLength + (vel.length()/maxSpeed)*feelerLength

    localTarget = vel.normal()
    heading = localTarget
    loc = pos
    heading = heading * feelerLength
    feeler = loc + heading
    feelers.append(feeler)

    heading = localTarget
    heading =  heading.rotateBy(OpenMaya.MVector.kZaxis, math.pi/4.0)
    feeler = loc + heading*feelerLength*0.6
    feelers.append(feeler)

    heading = localTarget
    heading =  heading.rotateBy(OpenMaya.MVector.kZaxis, -math.pi/4.0)
    feeler = loc + heading*feelerLength*0.6
    feelers.append(feeler)

    return feelers

def __obstacleAvoidanceHelper(self, pos, vel, maxSpeed, maxForce, i):
    distToThisIP = 0.0
    distToClosestIP = 1000000.0 # max double

    steerForce = OpenMaya.MVector(0,0,0)
    point = OpenMaya.MVector(0,0,0)
    closestPoint = OpenMaya.MVector(0,0,0)
    closestNorm = OpenMaya.MVector(0,0,0)


    feelers = self.createFeelers(pos, vel, i)
    activeFeeler = None
    for feeler in feelers:
        # just look for plugin nodes instead of maya standard nodes
        it = OpenMaya.MItDependencyNodes(OpenMaya.MFn.kPluginDependNode)
        while(not it.isDone()):
            fn = OpenMaya.MFnDependencyNode(it.item())
            if fn.typeName() == "spBoidColliderShape":
                dagNode = OpenMaya.MFnDagNode(it.item())
                path = OpenMaya.MDagPath()
                dagNode .getPath(path)
                path.pop()
                transform = OpenMaya.MFnTransform(path)
                co = transform.translation(OpenMaya.MSpace.kWorld)

                attr = fn.attribute("radius");
                plug = OpenMaya.MPlug(it.item(), attr)
                r = plug.asDouble()

                result = self.hit(pos, co, r, feeler)

                if result[0]:
                    distToThisIP = result[1] - feeler
```

```
                    if distToThisIP.length() < distToClosestIP:
                        distToClosestIP = distToThisIP.length()
                        closestPoint = result[1]
                        closestNorm = result[2]
                        self.norm = closestNorm
                        activeFeeler = feeler
                        self.ip = closestPoint
            it.next()

        if activeFeeler is not None:
            overShoot = activeFeeler - closestPoint
            dir = closestPoint - activeFeeler
            dir.normalize
            isOutside = dir*closestNorm

            if isOutside > 0:
                steerForce = (closestNorm * overShoot.length())

    if steerForce.length() > 0:
        steerForce.normalize()
        steerForce = steerForce * maxSpeed
        steerForce = steerForce - vel
        steerForce = self.__limit(steerForce, maxForce)

    return steerForce

def __obstacleAvoidance(self, block, positions, velocities, outputForce):
    if positions.length() == 0:
        return
    outputForce.clear()
    maxSpeed = self.__attrDoubleValue(block, SteeringNode.aMaxSpeed)
    maxForce = self.__attrDoubleValue(block, SteeringNode.aMaxForce)
    for i in range(1):
        steeringForce = self.__obstacleAvoidanceHelper(positions[i], velocities[i],
                                                       maxSpeed, maxForce, i)
        outputForce.append(steeringForce)

# ray sphere intersection test
def hit(self, o, co, r, feeler):
    d = feeler - o
    d.normalize()

    A = d*d
    B = ((o-co)*d)*2.0
    C = (o-co) * (o-co) - (r*r)

    discrim = B*B - 4*A*C

    if discrim < 0:
        return [False, None, None]

    distSqrt = math.sqrt(discrim)
    if B<0:
        q = (-B - distSqrt)/2.0
    else:
        q = (-B + distSqrt)/2.0

    t0 = q/A
    t1 = C/q

    if t0>t1:
        temp = t0
        t0 = t1
        t1 = temp
    if t1 < 0:
        return [False, None, None]
    if t0 < 0:
```

```
        t = t1
        ip = o + d*t
        norm = ip - co
        norm.normalize()
        return [True, ip, norm]
    else:
        t = t0
        ip = o + d*t
        norm = ip - co
        norm.normalize()
        return [True, ip, norm]
```

Appendix B

Obstacle Avoidance Maya NURBS Surface Solution

```
def _obstacleAvoidance(self, block, positions, velocities, outputForce):
    if positions.length() != velocities.length():
        return

    outputForce.clear()
    maxSpeed = self.__attrDoubleValue(block, SteeringNode.aMaxSpeed)
    maxForce = self.__attrDoubleValue(block, SteeringNode.aMaxForce)

    size = positions.length()

    bk = OpenMaya.MScriptUtil().asBoolPtr()
    distance =  OpenMaya.MScriptUtil().asDoublePtr()
    uIP = OpenMaya.MScriptUtil().asDoublePtr()
    vIP = OpenMaya.MScriptUtil().asDoublePtr()
    intersectionPoint = OpenMaya.MPoint()
    shortestDistance = OpenMaya.MDoubleArray(size, -1)
    activeFeelerLength = OpenMaya.MDoubleArray(size, -1.0)
    shortestUIP = OpenMaya.MDoubleArray(size, 0.0)
    shortestVIP = OpenMaya.MDoubleArray(size, 0.0)
    shortestSurface = OpenMaya.MIntArray(size,0);
    tolerance = OpenMaya.MScriptUtil().asDoublePtr()
    OpenMaya.MScriptUtil.setDouble(tolerance, 1.0e-3)


    hArrayValue = block.inputArrayValue(SteeringNode.aInputSurface)
    numSurfaces = hArrayValue.elementCount();
    if numSurfaces > 0:
        hArrayValue.jumpToElement(0)
        for i in range(numSurfaces):
            elementHandle = hArrayValue.inputValue()
            surface = elementHandle.asNurbsSurfaceTransformed()
            surfaceFn = OpenMaya.MFnNurbsSurface(surface)
            for j in range(size):
                feelers = self.createFeelers(positions[j], velocities[j], maxSpeed)
                sPoint = OpenMaya.MPoint(positions[j])
                for f in range(1):
                    feeler = feelers[f]
                    rayDir = feeler - positions[j]
                    feelerLength = rayDir.length()
                    rayDir.normalize()
                    hit =
surfaceFn.intersect(sPoint,rayDir,uIP,vIP,intersectionPoint, /
OpenMaya.MScriptUtil().getDouble(tolerance), /
OpenMaya.MSpace.kObject,True,distance)
                    d = OpenMaya.MScriptUtil().getDouble(distance)

                    if hit:
                        distToThisIP = OpenMaya.MVector(intersectionPoint) – /
OpenMaya.MVector(sPoint)
                        if feelerLength > distToThisIP.length():
                            shortestDistance[j] = distToThisIP.length()
                            shortestUIP[j] = OpenMaya.MScriptUtil().getDouble(uIP)
                            shortestVIP[j] = OpenMaya.MScriptUtil().getDouble(vIP)
                            shortestSurface[j] = i
                            activeFeelerLength[j] = feelerLength
            if i < (numSurfaces - 1):
                hArrayValue.next()

        uTangent = OpenMaya.MVector()
        vTangent = OpenMaya.MVector()
```

```
for i in range(size):
    if shortestDistance[i] > 0:
        hArrayValue.jumpToElement(shortestSurface[i])
        elementHandle = hArrayValue.inputValue()
        surface = elementHandle.asNurbsSurfaceTransformed()
        surfaceFn = OpenMaya.MFnNurbsSurface(surface)
        surfaceFn.getTangents(shortestUIP[i], shortestVIP[i], uTangent, vTangent)
        uvPerpend = uTangent ^ vTangent
        uvPerpend.normalize()
        overshoot = (activeFeelerLength[i] - shortestDistance[i])
        uvPerpend = uvPerpend * overshoot
        outputForce.append(self.__seekHelper(block, uvPerpend, velocities[i]))
    else:
        outputForce.append(OpenMaya.MVector.zero)
```

Appendix C

Boid Guides Plugin (boidGuides.py)

```
#*************************************************************
#  Plugin: boidGuides.py
#
#  Description:
#
#  This plugin creates visual guides for the SteeringField node.
#  The message attriubute of the SteeringField DAG node needs to
#  be connected to this node's steeringFieldInput attribute via
#  Maya's connection editor or command editor.
#
#  Author: Edgar Rodriguez
#*************************************************************


import math, sys
import maya.cmds as cmds
import maya.OpenMaya as OpenMaya
import maya.OpenMayaMPx as OpenMayaMPx
import maya.OpenMayaRender as OpenMayaRender
import maya.OpenMayaUI as OpenMayaUI

kPluginNodeTypeName = "spBoidGuides"
spBoidGuidesNodeId = OpenMaya.MTypeId(0x87355)

glRenderer = OpenMayaRender.MHardwareRenderer.theRenderer()
glFT = glRenderer.glFunctionTable()

kDefaultParticleID = 0
'''
Usage:
import maya.cmds as cmds
node = cmds.createNode("spBoidGuides", name="boidGuides")
parentName = cmds.listRelatives(node,p=True)[0]
newName = node + '_' + parentName
cmds.rename(parentName,newName)
'''

class BoidGuides(OpenMayaMPx.MPxLocatorNode):
    def __init__(self):
        OpenMayaMPx.MPxLocatorNode.__init__(self)
        # class variables
        aParticleID = OpenMaya.MObject()
        self._showAvoidSensor  = False
        self._showWanderSensor = False
        self._showFeelerSensor = False
        self._showBoundary = False
        self._allParticlesOn = False
        self._displayAxis = False
        self._particleID = kDefaultParticleID

    def compute(self, plug, dataBlock):
        return OpenMaya.MStatus.kSuccess

    # override
    def getInternalValue(self, plug, datahandle):
        if (plug == BoidGuides.aParticleID):
            datahandle.setInt(self._particleID)
        elif (plug == BoidGuides.aShowAvoidSensor):
            datahandle.setBool(self._showAvoidSensor)
        elif (plug == BoidGuides.aShowWanderSensor):
```

```
            datahandle.setBool(self._showWanderSensor)
        elif (plug == BoidGuides.aShowFeelerSensor):
            datahandle.setBool(self._showFeelerSensor)
        elif (plug == BoidGuides.aShowBoundary):
            datahandle.setBool(self._showBoundary)
        elif (plug == BoidGuides.aAllParticlesOn):
            datahandle.setBool(self._allParticlesOn)
        elif (plug == BoidGuides.aDisplayAxis):
            datahandle.setBool(self._displayAxis)
        else:
            return OpenMayaMPx.MPxLocatorNode.getInternalValue(self, plug, datahandle)
        return True

    # override
    def setInternalValue(self, plug, datahandle):
        if (plug == BoidGuides.aParticleID):
            particleID = datahandle.asInt()
            self._particleID = particleID
        elif (plug == BoidGuides.aShowAvoidSensor):
            self._showAvoidSensor = datahandle.asBool()
        elif (plug == BoidGuides.aShowWanderSensor):
            self._showWanderSensor = datahandle.asBool()
        elif (plug == BoidGuides.aShowFeelerSensor):
            self._showFeelerSensor = datahandle.asBool()
        elif (plug == BoidGuides.aShowBoundary):
            self._showBoundary = datahandle.asBool()
        elif (plug == BoidGuides.aAllParticlesOn):
            self._allParticlesOn = datahandle.asBool()
        elif (plug == BoidGuides.aDisplayAxis):
            self._displayAxis = datahandle.asBool()
        else:
            return OpenMayaMPx.MPxLocatorNode.setInternalValue(plug, datahandle)
        return True

    def getCirclePoints(self, r, size, pts):
        pt = OpenMaya.MPoint()
        pt.y = 0.0

        angleIncr = (2.0 * math.pi)/(size - 1)
        angle = 0.0

        for i in range(size):
            pt.x = r * math.cos(angle)
            pt.y = r * math.sin(angle)
            angle += angleIncr
            pts.append(pt)

    def drawCircle(self, r, size):
        pts = OpenMaya.MPointArray()
        self.getCirclePoints(r, size, pts)

        glFT.glBegin(OpenMayaRender.MGL_LINE_STRIP)
        for i in range(size):
            glFT.glVertex3f(pts[i].x, pts[i].y, pts[i].z)
        glFT.glEnd()

    def draw(self, view, path, style, status):

        # check if a boid is connected to this node (Visual Guides)
        thisNode = self.thisMObject()
        nodeFn = OpenMaya.MFnDependencyNode(thisNode)

        plug = nodeFn.findPlug("steeringFieldInput")

        if not plug.isConnected():
            return
```

```
connections = OpenMaya.MPlugArray()
plug.connectedTo(connections, True, False)
boidPlug = connections[0]
nNode = boidPlug.node()

bradius = 1
velocities = []
positions  = []
dagNode = OpenMaya.MFnDependencyNode(nNode)

plug = dagNode.findPlug("boundRadius")
bradius = plug.asDouble()

plug = dagNode.findPlug("wanderRadius")
wradius = plug.asDouble()

plug = dagNode.findPlug("feelerLength")
flength = plug.asDouble()

plug = dagNode.findPlug("wanderTargetPosition")
mfnVectorArray = OpenMaya.MFnVectorArrayData(plug.asMObject())
wanderTargetPositions = mfnVectorArray.array()

plug = dagNode.findPlug("wanderSpherePosition")
mfnVectorArray = OpenMaya.MFnVectorArrayData(plug.asMObject())
wanderSpherePositions = mfnVectorArray.array()

plug = dagNode.findPlug("upDirection")
mfnVectorArray = OpenMaya.MFnVectorArrayData(plug.asMObject())
upDirectionArray = mfnVectorArray.array()

plug = dagNode.findPlug("inputData")
numElements = plug.numConnectedElements()
for i in range(numElements):
    # returns inputData[i].inputData
    inputData0 = plug.elementByPhysicalIndex(i)
    # returns inputData[i].inputData[logicalIndex]
    inputData1 = plug.elementByLogicalIndex(inputData0.logicalIndex())

    for j in range(inputData1.numChildren()):
        iplug = inputData1.child(j)
        if "inputVelocities" in iplug.name():
            object = iplug.asMObject()
            mfnVectorArray = OpenMaya.MFnVectorArrayData(object)
            velocities.append(mfnVectorArray.array())
        if "inputPositions" in iplug.name():
            object = iplug.asMObject()
            mfnVectorArray = OpenMaya.MFnVectorArrayData(object)
            positions.append(mfnVectorArray.array())

view.beginGL()
glFT.glPushAttrib(OpenMayaRender.MGL_POLYGON_BIT)
glFT.glPolygonMode(OpenMayaRender.MGL_FRONT_AND_BACK, OpenMayaRender.MGL_LINE)

if(self._showAvoidSensor):
    self._drawAvoidSensor(bradius, positions, velocities,
                          self._particleID,self._allParticlesOn)
if(self._showFeelerSensor):
    self._drawFeelerSensor(flength, positions, velocities,
                          self._particleID,self._allParticlesOn)
if(self._displayAxis):
    self._drawAxis(10, positions, velocities, upDirectionArray,
                   self._particleID,self._allParticlesOn)
if(self._showBoundary):
    self._drawBoundary(bradius, positions, velocities,
                       self._particleID,self._allParticlesOn)
if(self._showWanderSensor):
```

```
        self._drawWanderSensor(wradius, positions, velocities, wanderTargetPositions,
                               wanderSpherePositions,
                               self._particleID,self._allParticlesOn)

    glFT.glPopAttrib()
    view.endGL()

def isBounded(self):
    return False

def _drawAxis(self, size, positions, velocities, upDirectionArray, id,
              allParticlesOn):
    upVector = OpenMaya.MGlobal.upAxis()

    for i in range(len(positions)):
        pArray = positions[i]
        vArray = velocities[i]

        if pArray.length()==0 or vArray.length()==0:
            return

        if allParticlesOn:
            idList = range(pArray.length())
        else:
            idList = [id]

        for id in idList:
            if(upDirectionArray.length() > 0):
                upVector = upDirectionArray[id]
            p0 = OpenMaya.MVector(pArray[id].x,pArray[id].y,pArray[id].z)
            unitVel = OpenMaya.MVector(vArray[id].x,vArray[id].y,vArray[id].z)
            unitVel = unitVel.normal()

            x = unitVel
            y = upVector^x
            z = x^y

            # X Axis
            p1 = p0 + x*size
            glFT.glBegin(OpenMayaRender.MGL_LINES)
            glFT.glColor3f(1.0, 0.0, 0.0)
            glFT.glVertex3f(p0.x, p0.y, p0.z)
            glFT.glVertex3f(p1.x, p1.y, p1.z)
            glFT.glEnd()

            # Y Axis
            p1 = p0 + y*size
            glFT.glBegin(OpenMayaRender.MGL_LINES)
            glFT.glColor3f(0.0, 1.0, 0.0)
            glFT.glVertex3f(p0.x, p0.y, p0.z)
            glFT.glVertex3f(p1.x, p1.y, p1.z)
            glFT.glEnd()

            # Z Axis
            p1 = p0 + z*size
            glFT.glBegin(OpenMayaRender.MGL_LINES)
            glFT.glColor3f(0.0, 0.0, 1.0)
            glFT.glVertex3f(p0.x, p0.y, p0.z)
            glFT.glVertex3f(p1.x, p1.y, p1.z)
            glFT.glEnd()

def _drawWanderSensor(self, wradius, positions, velocities, wanderTargetPositions,
                      wanderSpherePositions, id, allParticlesOn):
    for i in range(len(positions)):
        pArray = positions[i]
        vArray = velocities[i]
```

```
        if pArray.length()==0 or vArray.length()==0:
            return

        if allParticlesOn:
            idList = range(pArray.length())
        else:
            idList = [id]

        for id in idList:

            p0 = OpenMaya.MVector(pArray[id].x,pArray[id].y,pArray[id].z)
            p1 = p0 + OpenMaya.MVector(vArray[id].x,vArray[id].y,vArray[id].z)

            wsp = wanderSpherePositions[id]
            wtp = wanderTargetPositions[id]

            if wsp is None or wtp is None:
                return

            glFT.glColor3f(1.0, 1.0, 0.0)
            glFT.glPushMatrix()
            glFT.glTranslatef(wsp.x, wsp.y, wsp.z)
            self.drawCircle(wradius, 20)
            glFT.glPopMatrix()

            glFT.glColor3f(1.0, 0.0, 1.0)
            glFT.glPushMatrix()
            glFT.glTranslatef(wtp.x, wtp.y, wtp.z)
            self.drawCircle(0.2, 15)
            glFT.glPopMatrix()

            glFT.glColor3f(1.0, 1.0, 0.0)
            glFT.glBegin(OpenMayaRender.MGL_LINES)
            glFT.glVertex3f(p0.x, p0.y, p0.z)
            glFT.glVertex3f(wsp.x, wsp.y, wsp.z)
            glFT.glEnd()

            glFT.glBegin(OpenMayaRender.MGL_LINES)
            glFT.glVertex3f(wsp.x, wsp.y, wsp.z)
            glFT.glVertex3f(wtp.x, wtp.y, wtp.z)
            glFT.glEnd()

    def _drawFeelerSensor(self, flength, positions, velocities, id, allParticlesOn):
        for i in range(len(positions)):
            pArray = positions[i]
            vArray = velocities[i]

            if pArray.length()==0 or vArray.length()==0:
                return

            if allParticlesOn:
                idList = range(pArray.length())
            else:
                idList = [id]

            for id in idList:
                p0 = OpenMaya.MVector(pArray[id].x,pArray[id].y,pArray[id].z)
                unitVel = OpenMaya.MVector(vArray[id].x,vArray[id].y,vArray[id].z)
                unitVel = unitVel.normal()
                p1 = p0 + unitVel*flength
                glFT.glBegin(OpenMayaRender.MGL_LINES)
                glFT.glColor3f(1.0, 0.0, 0.0)
                glFT.glVertex3f(p0.x, p0.y, p0.z)
                glFT.glVertex3f(p1.x, p1.y, p1.z)
                glFT.glEnd()

    def _drawBoundary(self, bradius, positions, velocities, id, allParticlesOn):
```

```
    for i in range(len(positions)):
        pArray = positions[i]
        vArray = velocities[i]

        if pArray.length()==0 or vArray.length()==0:
            return

        if allParticlesOn:
            idList = range(pArray.length())
        else:
            idList = [id]

        for id in idList:
            p0 = OpenMaya.MVector(pArray[id].x,pArray[id].y,pArray[id].z)
            glFT.glColor3f(0.0, 1.0, 0.0)
            glFT.glPushMatrix()
            glFT.glTranslatef(p0.x, p0.y, p0.z)
            self.drawCircle(bradius, 15)
            glFT.glRotatef(90.0,1.0,0.0,0.0)
            glFT.glRotatef(90.0,0.0,0.0,1.0)
            self.drawCircle(bradius, 15)
            # appoximate sphere shape
            #for i in range(6):
            #    angle = 60*i
            #    glFT.glPushMatrix()
            #    glFT.glRotatef(angle,1.0,0.0,0.0)
            #    self.drawCircle(bradius, 15)
            #    glFT.glPopMatrix()
            glFT.glPopMatrix()

def _drawAvoidSensor(self, bradius, positions, velocities, id, allParticlesOn):
    for i in range(len(positions)):
        pArray = positions[i]
        vArray = velocities[i]

        if pArray.length()==0 or vArray.length()==0:
            return

        if allParticlesOn:
            idList = range(pArray.length())
        else:
            idList = [id]

        for id in idList:
            p0 = OpenMaya.MVector(pArray[id].x,pArray[id].y,pArray[id].z)
            vel = OpenMaya.MVector(vArray[id].x,vArray[id].y,vArray[id].z)
            p1 = p0 + vel

            direction = OpenMaya.MVector(vel)
            direction.normalize()

            glFT.glColor3f(0.0, 0.0, 1.0)
            numGuides = 3
            #0,1,2
            for i in range(numGuides+1):
                inc = (vel.length()/numGuides)*i
                pos = p0 + direction*inc
                glFT.glPushMatrix()
                glFT.glTranslatef(pos.x, pos.y,pos.z)
                self.drawCircle(bradius, 10)
                glFT.glPopMatrix()

            glFT.glBegin(OpenMayaRender.MGL_LINES)
            glFT.glVertex3f(p0.x, p0.y, p0.z)
            glFT.glVertex3f(p1.x, p1.y, p1.z)
            glFT.glEnd()
```

```
def addAttribute(attr, name):
    try:
        BoidGuides.addAttribute(attr)
    except:
        msg = "ERROR adding " + name + " attribute."
        statusError(msg)

# creator
def nodeCreator():
    return OpenMayaMPx.asMPxPtr( BoidGuides() )

# initializer
def nodeInitializer():

    def setOptions(attr):
        attr.setHidden(False)
        attr.setKeyable(True)
        attr.setInternal(True)

    msgAttr = OpenMaya.MFnMessageAttribute()
    numericAttrFn = OpenMaya.MFnNumericAttribute()

    BoidGuides.aAllParticlesOn = numericAttrFn.create("allParticlesOn", "allp",
                                 OpenMaya.MFnNumericData.kBoolean)
    setOptions(numericAttrFn)
    addAttribute(BoidGuides.aAllParticlesOn, "allParticlesOn")

    BoidGuides.aParticleID = numericAttrFn.create("particleID", "pid",
                                 OpenMaya.MFnNumericData.kLong, kDefaultParticleID)
    setOptions(numericAttrFn)
    addAttribute(BoidGuides.aParticleID, "particleID")

    BoidGuides.aShowAvoidSensor = numericAttrFn.create("showAvoidSensor", "sas",
                                 OpenMaya.MFnNumericData.kBoolean)
    setOptions(numericAttrFn)
    addAttribute(BoidGuides.aShowAvoidSensor, "showAvoidSensor")

    BoidGuides.aShowWanderSensor = numericAttrFn.create("showWanderSensor", "sws",
                                 OpenMaya.MFnNumericData.kBoolean)
    setOptions(numericAttrFn)
    addAttribute(BoidGuides.aShowWanderSensor, "showWanderSensor")

    BoidGuides.aShowFeelerSensor = numericAttrFn.create("showFeelerSensor", "sfs",
                                 OpenMaya.MFnNumericData.kBoolean)
    setOptions(numericAttrFn)
    addAttribute(BoidGuides.aShowFeelerSensor, "showFeelerSensor")

    BoidGuides.aShowBoundary = numericAttrFn.create("showBoundary", "sb",
                                 OpenMaya.MFnNumericData.kBoolean)
    setOptions(numericAttrFn)
    addAttribute(BoidGuides.aShowBoundary, "showBoundary")

    BoidGuides.aDisplayAxis = numericAttrFn.create("displayAxis", "displayAxis",
                                 OpenMaya.MFnNumericData.kBoolean)
    setOptions(numericAttrFn)
    addAttribute(BoidGuides.aDisplayAxis, "displayAxis")

    BoidGuides.aSteeringFieldInput = msgAttr.create("steeringFieldInput", "bin")
    addAttribute(BoidGuides.aSteeringFieldInput, "steeringFieldInput")

# initialize the script plug-in
def initializePlugin(mobject):
    mplugin = OpenMayaMPx.MFnPlugin(mobject, "DPA@Clemson", "1.0", "Any")
    try:
        mplugin.registerNode(kPluginNodeTypeName, spBoidGuidesNodeId, nodeCreator,
nodeInitializer, OpenMayaMPx.MPxNode.kLocatorNode)
    except:
```

```
        statusError("Failed to register node: %s" % kPluginNodeTypeName)


# uninitialize the script plug-in
def uninitializePlugin(mobject):
    mplugin = OpenMayaMPx.MFnPlugin(mobject)
    try:
        mplugin.deregisterNode(spBoidGuidesNodeId)
    except:
        statusError("Failed to deregister node: %s" % kPluginNodeTypeName)
```

Appendix D

Collider Sphere Plug-in (boidCollider.py)

```
#************************************************************
#  Plugin: boidCollider.py
#
#  Description:
#
#  Author: Edgar Rodriguez
#
#  Basic collision primitive recognized by SteeringField.
#************************************************************

import math, sys
import maya.OpenMaya as OpenMaya
import maya.OpenMayaMPx as OpenMayaMPx
import maya.OpenMayaRender as OpenMayaRender
import maya.OpenMayaUI as OpenMayaUI
import maya.cmds as cmds

kPluginNodeTypeName = "spBoidColliderSphere"
spBoidColliderSphereNodeId = OpenMaya.MTypeId(0x87357)

glRenderer = OpenMayaRender.MHardwareRenderer.theRenderer()
glFT = glRenderer.glFunctionTable()

'''
import maya.cmds as cmds
node = cmds.createNode("spBoidColliderSphere", name="boidCollider")
parentName = cmds.listRelatives(node,p=True)[0]
newName = node + '_' + parentName
cmds.rename(parentName,newName)
'''

kDefaultRadius = 1.0

class boidColliderSphere(OpenMayaMPx.MPxLocatorNode):
    def __init__(self):
        OpenMayaMPx.MPxLocatorNode.__init__(self)
        # class variables
        aRadius = OpenMaya.MObject()
        self._radius = kDefaultRadius
        self._initConnections = 1

    def compute(self, plug, dataBlock):
        return OpenMaya.MStatus.kSuccess

    def getTranslation(self):
        dagFn = OpenMaya.MFnDagNode(self.thisMObject())
        path = OpenMaya.MDagPath()
        dagFn.getPath(path)
        path.pop()
        transformFn = OpenMaya.MFnTransform(path)
        return transformFn.getTranslation(OpenMaya.MSpace.kWorld)

    def getScale(self):
        dagFn = OpenMaya.MFnDagNode(self.thisMObject())
        path = OpenMaya.MDagPath()
        dagFn.getPath(path)
        path.pop()
        transformFn = OpenMaya.MFnTransform(path)
        doubleArray = OpenMaya.MScriptUtil()
        doubleArray.createFromList( [1.0, 1.0, 1.0], 3 )
```

```
        doubleArrayPtr = doubleArray.asDoublePtr()
        transformFn.getScale(doubleArrayPtr)
        vec = OpenMaya.MVector( doubleArrayPtr )
        return vec

    # override
    def getInternalValue(self, plug, datahandle):
        if (plug == boidColliderSphere.aRadius):
            datahandle.setDouble(self._radius)
        else:
            return OpenMayaMPx.MPxLocatorNode.getInternalValue(self, plug, datahandle)
        return True

    # override
    def setInternalValue(self, plug, datahandle):
        # the minimum radius is 0.25
        if (plug == boidColliderSphere.aRadius):
            radius = datahandle.asDouble()
            if (radius < 0.25):
                radius = 0.25
            self._radius = radius
        else:
            return OpenMayaMPx.MPxLocatorNode.setInternalValue(plug, datahandle)
        return True

    # override
    def excludeAsLocator(self):
        return True

    # override
    def draw(self, view, path, style, status):
        view.beginGL()

        glFT.glPushAttrib( OpenMayaRender.MGL_ALL_ATTRIB_BITS )
        glFT.glPolygonMode(OpenMayaRender.MGL_FRONT_AND_BACK, OpenMayaRender.MGL_LINE)

        # draw the sphere
        for j in range(0,10):
            angle = 36.0 * j # on degrees
            glFT.glPushMatrix()
            glFT.glRotatef(angle,0.0,0.0,1.0)
            glFT.glBegin(OpenMayaRender.MGL_POLYGON)
            for i in range(0,36):
                angle = 10.0*i
                angle = angle * math.pi/180.0 # in radians
                if (i == 360):
                    glFT.glVertex3f(self._radius*math.cos(0), 0.0,
self._radius*math.sin(0))
                else:
                    glFT.glVertex3f(self._radius*math.cos(angle), 0.0,
self._radius*math.sin(angle))
            glFT.glEnd()
            glFT.glPopMatrix()
        glFT.glPushMatrix()
        glFT.glBegin(OpenMayaRender.MGL_POLYGON)
        for i in range(0,36):
            angle = 10.0*i
            angle = angle * math.pi/180.0 # in radians
            if (i == 360):
                glFT.glVertex3f(self._radius*math.cos(0), self._radius*math.sin(0), 0.0)
            else:
                glFT.glVertex3f(self._radius*math.cos(angle),
self._radius*math.sin(angle), 0.0)
        glFT.glEnd()
        glFT.glPopMatrix()
        glFT.glPopAttrib()
        view.endGL()
```

```
    def isBounded(self):
        return True

    # override
    def boundingBox(self):
        result = OpenMaya.MBoundingBox()
        r = self._radius
        result.expand(OpenMaya.MPoint(r,r,r))
        result.expand(OpenMaya.MPoint(-r,-r,-r))
        return result

# creator
def nodeCreator():
    return OpenMayaMPx.asMPxPtr( boidColliderSphere() )

# initializer
def nodeInitializer():
    # utility func for numeric attrs
    def setOptions(attr):
        attr.setHidden(False)
        attr.setKeyable(True)
        attr.setInternal(True)

    numericAttr = OpenMaya.MFnNumericAttribute()

    boidColliderSphere.aRadius = numericAttr.create("radius", "r",
OpenMaya.MFnNumericData.kDouble, kDefaultRadius)
    setOptions(numericAttr)
    boidColliderSphere.addAttribute(boidColliderSphere.aRadius)

# initialize the script plug-in
def initializePlugin(mobject):
    mplugin = OpenMayaMPx.MFnPlugin(mobject, "DPA@Clemson", "1.0", "Any")
    try:
        mplugin.registerNode(kPluginNodeTypeName, spBoidColliderSphereNodeId,
nodeCreator, nodeInitializer, OpenMayaMPx.MPxNode.kLocatorNode)
    except:
        statusError("Failed to register node: %s" % kPluginNodeTypeName)

# uninitialize the script plug-in
def uninitializePlugin(mobject):
    mplugin = OpenMayaMPx.MFnPlugin(mobject)
    try:
        mplugin.deregisterNode(spBoidColliderSphereNodeId)
    except:
        statusError("Failed to deregister node: %s" % kPluginNodeTypeName)
```

Appendix E

Steering Field Plug-in (SteeringField.py)

```
#************************************************************
#  Start Date:   03.13.2009
#  Finish Date:  05.31.2010
#  Revised Date: 06.27.2010
#
#  Plugin: SteeringField.py
#
#  Description:
#
#  The steeringField node implements a custom force that enable
#  particles to have some degree of autonomous movement. Using a set
#  of basic rules the particles  are able to respond to their
#  immediate neighbors and environment.
#
#  Author: Edgar Rodriguez
#************************************************************

import math, sys, random
import maya.OpenMayaAnim as OpenMayaAnim
import maya.OpenMaya as OpenMaya
import maya.OpenMayaUI as OpenMayaUI
import maya.OpenMayaMPx as OpenMayaMPx
import maya.OpenMayaFX as OpenMayaFX
import maya.OpenMayaRender as OpenMayaRender

kPluginName = "spSteeringField"
kPluginNodeId = OpenMaya.MTypeId(0x87356)

glRenderer = OpenMayaRender.MHardwareRenderer.theRenderer()
glFT = glRenderer.glFunctionTable()

'''
Usage:
import maya.cmds as cmds
cmds.createNode("spSteeringField", name="steeringField")
'''

def randomClamped():  # returns -1 < n < 1
    return random.random() - random.random()

def statusError(msg):
    sys.stderr.write("%s\n" % msg)
    raise

class Capsule(object):
    # the values a and b define the length of the capsule
    def __init__(self,a,b,radius):
        self.a = a
        self.b = b
        self.radius = radius

class SteeringField(OpenMayaMPx.MPxFieldNode):
    '''
    Attributes
    '''
    def __init__(self):

        OpenMayaMPx.MPxFieldNode.__init__(self)
        self._animControl = OpenMayaAnim.MAnimControl()
```

```
        self._debug_projQ = OpenMaya.MVector(1,1,1)
        self._debug_perpQ = OpenMaya.MVector(1,1,1)
        self._debug_A = OpenMaya.MVector(1,1,1)
        self._debug_P = OpenMaya.MVector(1,1,1)
        self.debug_curVel = OpenMaya.MVector(1,1,1)

        self._curParticleShape = None
        self._restPositions = None
        self._currentWayPoint = []
        self._particleShapes = []
        self._particlePositions = {}
        self._particleVelocities = {}

    def _limit(self, v, max):

        thisNode = self.thisMObject()
        nodeFn = OpenMaya.MFnDependencyNode(thisNode)
        plug = nodeFn.findPlug("weightTruncatedSumPrioritizationOn")

        weightTruncatedSumPrioritizationOn = plug.asBool()

        if weightTruncatedSumPrioritizationOn:
            return v

        if(v.length() > max):
            v.normalize()
            v = v * max
        return v

    def _fleeTarget(self, block, positions, velocities, targets, outputForce):
        if positions.length() != velocities.length():
            return
        outputForce.clear()

        targetSize = targets.length();
        if targetSize == 0:
            return

        maxSpeed = self._attrDoubleValue(block, SteeringField.aMaxSpeed)
        maxForce = self._attrDoubleValue(block, SteeringField.aMaxForce)
        seed = self._attrDoubleValue(block, SteeringField.aSeed)
        panicDistance = self._attrDoubleValue(block, SteeringField.aPanicDistance)
        panicDistanceSq = panicDistance*panicDistance

        random.seed(seed)
        for i in range(positions.length()):
            targetIndex = random.randint(1,targetSize) - 1
            desiredVel = positions[i] - targets[targetIndex]
            DistanceSq = desiredVel.x*desiredVel.x + desiredVel.y*desiredVel.y +
desiredVel.z*desiredVel.z
            # proximity test, distance is in squared-distance space
            if(DistanceSq > panicDistanceSq):
                steeringForce = OpenMaya.MVector(0.0,0.0,0.0)
            else:
                desiredVel.normalize()
                desiredVel = desiredVel * maxSpeed
                steeringForce = desiredVel - velocities[i]
                steeringForce = self._limit(steeringForce, maxForce)

            outputForce.append(steeringForce)

    def _seekHelper(self, block, desiredVel, currentVel):
        maxSpeed = self._attrDoubleValue(block, SteeringField.aMaxSpeed)
        maxForce = self._attrDoubleValue(block, SteeringField.aMaxForce)
        dist = desiredVel.length()
        steeringForce = OpenMaya.MVector(0.0,0.0,0.0)
        if (dist > 0):
```

```
            desiredVel.normalize()
            desiredVel = desiredVel * maxSpeed
            steeringForce = desiredVel - currentVel
            steeringForce = self._limit(steeringForce, maxForce)

        return steeringForce

    def _surfaceFollowHelper(self, block, desiredVel, currentVel):
        maxSpeed = self._attrDoubleValue(block, SteeringField.aMaxSpeed)
        maxForce = self._attrDoubleValue(block, SteeringField.aMaxForce)

        upwardMult = self._attrDoubleValue(block, SteeringField.aGroundUpwardMult)
        downwardMult = self._attrDoubleValue(block, SteeringField.aGroundDownwardMult)

        dist = desiredVel.length()
        '''
        compute dot product to determine if boid is going upward or downward, vector
(0,0,-1)
        refers to a planes normal, in this case if the dot product is > 0 then the
particle's
        normalized velocity lies on the same side of the plane, hence the particle is
going down
        if the dot product is < 0 then the particle's normalized velocity lies on the
opposide
        side of the plane, hence the particle is going up
        '''
        direction = currentVel.normal() * OpenMaya.MVector(0,0,-1)

        if (dist > 0):
            desiredVel.normalize()
             # going down, speed up
            if direction > 0:
                desiredVel = desiredVel * maxSpeed * downwardMult
            else:
                # slow down
                desiredVel = desiredVel * maxSpeed * upwardMult

            steeringForce = (desiredVel - currentVel)
            steeringForce = self._limit(steeringForce, maxForce)
        else:
            steeringForce = OpenMaya.MVector(0.0,0.0,0.0)
        return steeringForce

    def _heading(self, block, positions, velocities, outputForce):

        if positions.length() != velocities.length():
            return
        outputForce.clear()

        leaderID = self._attrLongValue(block, SteeringField.aFollowLeaderID)
        leaderID = self.getValidLeaderID(leaderID, positions.length())

        if leaderID > 0:
            pRange = range(leaderID, leaderID+1)

        for i in range(positions.length()):
            headingHandle = block.inputValue(SteeringField.aHeading)
            heading = headingHandle.asFloatVector()
            currentVel = velocities[i]
            if heading.length() == 0.0:
                desiredVel = OpenMaya.MVector(currentVel)
            else:
                desiredVel = OpenMaya.MVector(heading)

            steeringForce = self._seekHelper(block, desiredVel, currentVel)
            if leaderID > 0:
                if i != leaderID:
```

```
                steeringForce = OpenMaya.MVector(0.0,0.0,0.0)
            outputForce.append(steeringForce)

    def _pathFollow(self, block, positions, velocities, wayPoints, outputForce):

        if positions.length() != velocities.length():
            return
        outputForce.clear()

        leaderID = self._attrLongValue(block, SteeringField.aFollowLeaderID)
        leaderID = self.getValidLeaderID(leaderID, positions.length())
        wayPointProximityRadius = self._attrDoubleValue(block,
SteeringField.aWayPointProximityRadius)
        pathFollowLoopOn = self._attrBooleanValue(block, SteeringField.aPathFollowLoopOn)

        numWayPoints = wayPoints.length();

        if numWayPoints == 0:
            return

        for i in range(positions.length()):


            if self._currentWayPoint[i] >= numWayPoints:
                if pathFollowLoopOn:
                    self._currentWayPoint[i] = 0
                else:
                    self._currentWayPoint[i] = int(numWayPoints - 1)

            wayPoint = wayPoints[self._currentWayPoint[i]]

            desiredVel = wayPoint - positions[i]
            distToWayPoint = desiredVel.length()
            steerFunc = self._seekHelper
            if(distToWayPoint<=wayPointProximityRadius):
                if self._currentWayPoint[i] < numWayPoints:
                    self._currentWayPoint[i] = int(self._currentWayPoint[i] + 1)
                else:
                    steerFunc = self._arriveHelper

            currentVel = velocities[i]
            steeringForce = steerFunc(block, desiredVel, currentVel)

            if leaderID>0:
                if i != leaderID:
                    steeringForce = OpenMaya.MVector(0.0,0.0,0.0)

            outputForce.append(steeringForce)

    def _seekTarget(self, block, positions, velocities, targets, outputForce):

        if positions.length() != velocities.length():
            return
        outputForce.clear()

        leaderID = self._attrLongValue(block, SteeringField.aFollowLeaderID)
        leaderID = self.getValidLeaderID(leaderID, positions.length())
        seed = self._attrDoubleValue(block, SteeringField.aSeed)

        targetSize = targets.length();

        if targetSize == 0:
            return

        random.seed(seed)
        for i in range(positions.length()):
            targetIndex = random.randint(1,targetSize) - 1
```

```
            desiredVel = targets[targetIndex] - positions[i]
            currentVel = velocities[i]
            steeringForce = self._seekHelper(block, desiredVel, currentVel)
            if leaderID>0:
                if i != leaderID:
                    steeringForce = OpenMaya.MVector(0.0,0.0,0.0)
            outputForce.append(steeringForce)

    def _arriveHelper(self, block, desiredVel, currentVel):
        maxSpeed = self._attrDoubleValue(block, SteeringField.aMaxSpeed)
        maxForce = self._attrDoubleValue(block, SteeringField.aMaxForce)
        decelerationMult = self._attrDoubleValue(block, SteeringField.aDecelerationMult)
        '''
        deceleration damping factor
        '''
        decelerationDamping = self._attrDoubleValue(block,
SteeringField.aDecelerationDamping)

        dist = desiredVel.length()

        steeringForce = OpenMaya.MVector(0.0,0.0,0.0)

        if ( dist > 0.0 ):
            '''
            calculate the speed required to reach target given the desiredVel
            '''
            speed = dist/(decelerationMult*decelerationDamping)

            '''
            make sure the current speed does not exceed the maxSpeed
            '''
            speed = min(speed, maxSpeed)

            desiredVel = desiredVel * speed

            steeringForce = desiredVel - currentVel
            steeringForce = self._limit(steeringForce, maxForce)

        return steeringForce

    def _arriveTarget(self, block, positions, velocities, targets, outputForce):
        if positions.length() != velocities.length():
            return
        outputForce.clear()

        leaderID = self._attrLongValue(block, SteeringField.aFollowLeaderID)
        leaderID = self.getValidLeaderID(leaderID, positions.length())
        seed = self._attrDoubleValue(block, SteeringField.aSeed)

        targetSize = targets.length();
        if targetSize == 0:
            return

        random.seed(seed)
        for i in range(positions.length()):
            targetIndex = random.randint(1,targetSize) - 1
            desiredVel = targets[targetIndex] - positions[i]
            currentVel = velocities[i]
            steeringForce = self._arriveHelper(block, desiredVel, currentVel)
            if leaderID>0:
                if i != leaderID:
                    steeringForce = OpenMaya.MVector(0.0,0.0,0.0)
            outputForce.append(steeringForce)

    def _wander(self, block, positions, velocities, outputForce):
        if positions.length() != velocities.length():
            return
```

```
        outputForce.clear()

        leaderID = self._attrLongValue(block, SteeringField.aFollowLeaderID)
        leaderID = self.getValidLeaderID(leaderID, positions.length())

        wanderDistance = self._attrDoubleValue(block, SteeringField.aWanderDistance)
        wanderRadius = self._attrDoubleValue(block, SteeringField.aWanderRadius)
        wanderJitter = self._attrDoubleValue(block, SteeringField.aWanderJitter)

        dataVectorArrayFn = OpenMaya.MFnVectorArrayData()
        targetData = block.outputValue(SteeringField.aWanderTargetPosition).data();
        dataVectorArrayFn.setObject(targetData)
        outTargetPositionArray = dataVectorArrayFn.array()
        outTargetPositionArray.setLength(positions.length())

        dataVectorArrayFn = OpenMaya.MFnVectorArrayData()
        targetData = block.outputValue(SteeringField.aWanderSpherePosition).data();
        dataVectorArrayFn.setObject(targetData)
        outWanderSpherePositionArray = dataVectorArrayFn.array()
        outWanderSpherePositionArray.setLength(positions.length())

        dataVectorArrayFn = OpenMaya.MFnVectorArrayData()
        targetData = block.outputValue(SteeringField.aWanderTarget).data();
        dataVectorArrayFn.setObject(targetData)
        outWanderTargetArray = dataVectorArrayFn.array()
        outWanderTargetArray.setLength(positions.length())

        wanderScaleHandle = block.inputValue(SteeringField.aWanderScale)
        wanderScale = wanderScaleHandle.asFloatVector()

        for i in range(positions.length()):
            wanderCircleOffset = OpenMaya.MVector(velocities[i])
            wanderCircleOffset.normalize()
            wanderCircleOffset = wanderCircleOffset*wanderDistance
            wanderCircleOffset = wanderCircleOffset + positions[i]

            outWanderTargetArray[i].x = (outWanderTargetArray[i].x +
randomClamped()*wanderJitter)*wanderScale.x
            outWanderTargetArray[i].y = (outWanderTargetArray[i].y +
randomClamped()*wanderJitter)*wanderScale.y
            outWanderTargetArray[i].z = (outWanderTargetArray[i].z +
randomClamped()*wanderJitter)*wanderScale.z
            outWanderTargetArray[i].normalize()
            outWanderTargetArray[i].x = outWanderTargetArray[i].x * wanderRadius
            outWanderTargetArray[i].y = outWanderTargetArray[i].y * wanderRadius
            outWanderTargetArray[i].z = outWanderTargetArray[i].z * wanderRadius

            targetWorldPosition = outWanderTargetArray[i] + wanderCircleOffset
            desiredVel = targetWorldPosition - positions[i]

            outWanderSpherePositionArray[i].x = wanderCircleOffset.x
            outWanderSpherePositionArray[i].y = wanderCircleOffset.y
            outWanderSpherePositionArray[i].z = wanderCircleOffset.z

            outTargetPositionArray[i].x = targetWorldPosition.x
            outTargetPositionArray[i].y = targetWorldPosition.y
            outTargetPositionArray[i].z = targetWorldPosition.z

            steeringForce = self._seekHelper(block, desiredVel,velocities[i])
            if leaderID > 0:
                if i != leaderID:
                    steeringForce = OpenMaya.MVector(0.0,0.0,0.0)
            outputForce.append(steeringForce)

    def _listNeighbors(self, block, positions, velocities, nRadius):
        if positions.length() != velocities.length():
            return
```

```
        bRadius = self._attrDoubleValue(block, SteeringField.aBoundRadius) # boid
bounding sphere radius
        neighbors = {}

        for i in range(positions.length()):
            curBoidPos = positions[i]
            # stores ids of boids neighboring curBoid
            neighbors[i] = []
            for j in range(positions.length()):
                offsetV = positions[j] - curBoidPos
                if j != i:
                    if offsetV.length() <= (nRadius + bRadius):
                        l = neighbors[i]
                        l.append(j)
        return neighbors

    def _separate(self, block, positions, velocities, neighbors, outputForce):
        if positions.length() != velocities.length():
            return
        outputForce.clear()

        maxSpeed = self._attrDoubleValue(block, SteeringField.aMaxSpeed)
        maxForce = self._attrDoubleValue(block, SteeringField.aMaxForce)
        for i in neighbors.keys():
            neighborList = neighbors[i]
            pos = positions[i]
            desiredVel = OpenMaya.MVector(0,0,0)
            for j in neighborList:
                toVector = pos - positions[j]
                dist = toVector.length()
                if dist > 0.001:
                    toVector.normalize()
                    toVector = toVector * (1.0/dist)
                    desiredVel = desiredVel + toVector

            steeringForce = OpenMaya.MVector(0,0,0)
            if desiredVel.length() > 0:
                desiredVel.normalize()
                desiredVel = desiredVel * maxSpeed
                steeringForce = desiredVel - velocities[i]
            steeringForce = self._limit(steeringForce, maxForce)
            outputForce.append(steeringForce)

    def _alignment(self, block, positions, velocities, neighbors, outputForce):
        if positions.length() != velocities.length():
            return
        outputForce.clear()

        maxSpeed = self._attrDoubleValue(block, SteeringField.aMaxSpeed)
        maxForce = self._attrDoubleValue(block, SteeringField.aMaxForce)
        for i in neighbors.keys():
            neighborList = neighbors[i]
            averageHeading = OpenMaya.MVector(0,0,0)
            for j in neighborList:
                averageHeading = averageHeading + velocities[j].normal()

            count = len(neighborList)
            if count > 0:
                averageHeading = averageHeading * (1.0/count)

            if averageHeading.length() > 0:
                averageHeading.normalize()
                averageHeading = averageHeading * maxSpeed
                averageHeading = averageHeading - velocities[i]
            averageHeading = self._limit(averageHeading, maxForce)
            outputForce.append(averageHeading)
```

```
    def _cohesion(self,block, positions, velocities, neighbors, outputForce):
        if positions.length() != velocities.length():
            return
        outputForce.clear()

        for i in neighbors.keys():
            neighborList = neighbors[i]
            steeringForce = OpenMaya.MVector(0,0,0)
            centerOfMass = OpenMaya.MVector(0,0,0)
            for j in neighborList:
                centerOfMass = centerOfMass + positions[j]

            count = len(neighborList)
            if count > 0:
                centerOfMass = centerOfMass * (1.0/count)
                desiredVel = centerOfMass - positions[i]
                steeringForce = self._seekHelper(block, desiredVel, velocities[i])

            outputForce.append(steeringForce)

    def _calculateForce(self, block, points, velocities, forceArray):
        forceArray.clear()
        size = points.length()

        if len(self._currentWayPoint) <= 0:
            self._currentWayPoint = [int(i*0) for i in range(size)]

        upVector = OpenMaya.MGlobal.upAxis()

        dataVectorArrayFn = OpenMaya.MFnVectorArrayData()
        data = block.outputValue(SteeringField.aXYZRotation).data();
        dataVectorArrayFn.setObject(data)
        outXYZRotationArray = dataVectorArrayFn.array()

        dataVectorArrayFn = OpenMaya.MFnVectorArrayData()
        data = block.outputValue(SteeringField.aXYZProjection).data();
        dataVectorArrayFn.setObject(data)
        outXYZProjectionArray = dataVectorArrayFn.array()

        flipUpVectorOn = self._attrBooleanValue(block, SteeringField.aFlipUpVectorOn)
        maxForce = self._attrDoubleValue(block, SteeringField.aMaxForce)
        scaleForceHandle = block.inputValue(SteeringField.aScaleForce)
        scaleForce = scaleForceHandle.asFloatVector()

        surfaceProjectionOn = self._attrBooleanValue(block,
SteeringField.aSurfaceProjectionOn)
        separationOn = self._attrBooleanValue(block, SteeringField.aSeparationOn)
        wanderOn = self._attrBooleanValue(block, SteeringField.aWanderOn)
        arriveOn = self._attrBooleanValue(block, SteeringField.aArriveOn)
        cohesionOn = self._attrBooleanValue(block, SteeringField.aCohesionOn)
        alignmentOn = self._attrBooleanValue(block, SteeringField.aAlignOn)
        obstacleAvoidanceOn = self._attrBooleanValue(block,
SteeringField.aObstacleAvoidanceOn)
        headingOn = self._attrBooleanValue(block, SteeringField.aHeadingOn)
        seekOn = self._attrBooleanValue(block, SteeringField.aSeekOn)
        fleeOn = self._attrBooleanValue(block, SteeringField.aFleeOn)
        groundOn = self._attrBooleanValue(block, SteeringField.aGroundOn)
        boxOn = self._attrBooleanValue(block, SteeringField.aBoxOn)
        followLeaderOn = self._attrBooleanValue(block, SteeringField.aFollowLeaderOn)
        pathFollowOn = self._attrBooleanValue(block, SteeringField.aPathFollowOn)
        neighborRepulseOn = self._attrBooleanValue(block,
SteeringField.aNeighborRepulseOn)

        forceArray_separation = OpenMaya.MVectorArray(size)
        forceArray_alignment = OpenMaya.MVectorArray(size)
        forceArray_cohesion = OpenMaya.MVectorArray(size)
        forceArray_followLeader = OpenMaya.MVectorArray(size)
```

```
        forceArray_arrive = OpenMaya.MVectorArray(size)
        forceArray_wander = OpenMaya.MVectorArray(size)
        forceArray_heading = OpenMaya.MVectorArray(size)
        forceArray_seek = OpenMaya.MVectorArray(size)
        forceArray_flee = OpenMaya.MVectorArray(size)
        forceArray_avoidance = OpenMaya.MVectorArray(size)
        forceArray_ground = OpenMaya.MVectorArray(size)
        forceArray_box = OpenMaya.MVectorArray(size)
        forceArray_pathFollow = OpenMaya.MVectorArray(size)
        forceArray_neighborRepulse = OpenMaya.MVectorArray(size)

        seekTargets = self.getTargets(block,SteeringField.aSeekTargets)
        arriveTargets = self.getTargets(block,SteeringField.aArriveTargets)
        fleeTargets = self.getTargets(block,SteeringField.aFleeTargets)
        wayPoints = self.getWayPoints(block)

        weightTruncatedSumPrioritizationOn = self._attrBooleanValue(block,
SteeringField.aWeightTruncatedSumPrioritizationOn)

        if surfaceProjectionOn:
            self._surfaceProjection(block, points, velocities)

        if separationOn:
            nRadius = self._attrDoubleValue(block,
SteeringField.aSeparationNeighborDistance)
            neighbors = self._listNeighbors(block, points, velocities, nRadius)
            self._separate(block, points, velocities, neighbors, forceArray_separation)
            neighbors.clear()

        if alignmentOn:
            nRadius = self._attrDoubleValue(block, SteeringField.aAlignNeighborDistance)
            neighbors = self._listNeighbors(block, points, velocities, nRadius)
            self._alignment(block, points, velocities, neighbors, forceArray_alignment)
            neighbors.clear()

        if cohesionOn:
            nRadius = self._attrDoubleValue(block,
SteeringField.aCohesionNeighborDistance)
            neighbors = self._listNeighbors(block, points, velocities, nRadius)
            self._cohesion(block, points, velocities, neighbors, forceArray_cohesion)
            neighbors.clear()

        if followLeaderOn:
            self._followLeader(block, points, velocities, forceArray_followLeader)

        if pathFollowOn:
            self._pathFollow(block, points, velocities, wayPoints, forceArray_pathFollow)

        if arriveOn:
            self._arriveTarget(block, points, velocities, arriveTargets,
forceArray_arrive)

        if wanderOn:
            self._wander(block, points, velocities, forceArray_wander)

        if headingOn:
            self._heading(block, points, velocities, forceArray_heading)

        if fleeOn:
            self._fleeTarget(block, points, velocities, fleeTargets, forceArray_flee)

        if seekOn:
            self._seekTarget(block, points, velocities, seekTargets, forceArray_seek)

        if obstacleAvoidanceOn:
            self._obstacleAvoidance(block, points, velocities, forceArray_avoidance)
```

```
        if boxOn:
            self._boxAvoidance(block, points, velocities, forceArray_box)

        if groundOn:
            self._surfaceFollow(block, points, velocities, forceArray_ground)

        if neighborRepulseOn:
            nRadius = self._attrDoubleValue(block,
SteeringField.aReplusionNeighborDistance)
            neighbors = self._listNeighbors(block, points, velocities, nRadius)
            self._neighborRepulse(block, points, velocities, neighbors,
forceArray_neighborRepulse)

        multSeparation = self._attrDoubleValue(block, SteeringField.aMultSeparation)
        multAlignment  = self._attrDoubleValue(block, SteeringField.aMultAlign)
        multCohesion   = self._attrDoubleValue(block, SteeringField.aMultCohesion)
        multFollowLeader   = self._attrDoubleValue(block,
SteeringField.aMultFollowLeader)
        multArrive     = self._attrDoubleValue(block, SteeringField.aMultArrive)
        multWander     = self._attrDoubleValue(block, SteeringField.aMultWander)
        multHeading    = self._attrDoubleValue(block, SteeringField.aMultHeading)
        multSeek       = self._attrDoubleValue(block, SteeringField.aMultSeek)
        multFlee       = self._attrDoubleValue(block, SteeringField.aMultFlee)
        multAvoid      = self._attrDoubleValue(block, SteeringField.aMultAvoid)
        multGround     = self._attrDoubleValue(block, SteeringField.aMultGround)
        multBox     = self._attrDoubleValue(block, SteeringField.aMultBox)
        multPathFollow = self._attrDoubleValue(block, SteeringField.aMultPathFollow)
        multNeighborRepulse = self._attrDoubleValue(block,
SteeringField.aMultNeighborRepulse)

        if not groundOn:
            dataVectorArrayFn = OpenMaya.MFnVectorArrayData()
            data = block.outputValue(SteeringField.aUpDirection).data();
            dataVectorArrayFn.setObject(data)
            outUpDirectionArray = dataVectorArrayFn.array()
            outUpDirectionArray.clear()
            outUpDirectionArray.setLength(points.length())

            outXYZRotationArray.clear()
            outXYZRotationArray.setLength(points.length())

            for i in range(points.length()):
                unitVel = velocities[i].normal()
                if flipUpVectorOn:
                    x = unitVel
                    y = x^upVector
                    z = y^x
                else:
                    x = unitVel
                    y = upVector^x
                    z = x^y

                upv = z.normal()
                outUpDirectionArray[i].x = upv.x
                outUpDirectionArray[i].y = upv.y
                outUpDirectionArray[i].z = upv.z

                MatComponents = (x.x, x.y, x.z, 0.0, y.x, y.y, y.z, 0.0, z.x, z.y, z.z,
0.0, 0.0, 0.0, 0.0, 1.0)
                matrix = OpenMaya.MMatrix()
                OpenMaya.MScriptUtil.createMatrixFromList( MatComponents, matrix)
                mtm = OpenMaya.MTransformationMatrix(matrix)
                erot = mtm.eulerRotation()
                rot = erot.asVector()
                mangle = OpenMaya.MAngle()

                mangle.setValue(rot.x)
```

```
x = mangle.asDegrees()

mangle.setValue(rot.y)
y = mangle.asDegrees()

mangle.setValue(rot.z)
z = mangle.asDegrees()

outXYZRotationArray[i].x = x
outXYZRotationArray[i].y = y
outXYZRotationArray[i].z = z

if weightTruncatedSumPrioritizationOn:
    runningTotal = OpenMaya.MVector(0.0,0.0,0.0)
    if boxOn:
        force = forceArray_box[i]*multBox
        self._accumulateForce(block,runningTotal,force,p=True)

    if obstacleAvoidanceOn:
        force = forceArray_avoidance[i]*multAvoid
        self._accumulateForce(block,runningTotal,force)

    if neighborRepulseOn:
        force = forceArray_neighborRepulse[i]*multNeighborRepulse
        self._accumulateForce(block,runningTotal,force)

    if fleeOn:
        force = forceArray_flee[i]*multFlee
        self._accumulateForce(block,runningTotal,force,p=True)

    if followLeaderOn:
        force = forceArray_followLeader[i]*multFollowLeader
        self._accumulateForce(block,runningTotal,force)

    if separationOn:
        force = forceArray_separation[i]*multSeparation
        self._accumulateForce(block,runningTotal,force, p=True)

    if alignmentOn:
        force = forceArray_alignment[i]*multAlignment
        self._accumulateForce(block,runningTotal,force)

    if cohesionOn:
        force = forceArray_cohesion[i]*multCohesion
        self._accumulateForce(block,runningTotal,force)

    if pathFollowOn:
        force = forceArray_pathFollow[i]*multPathFollow
        self._accumulateForce(block,runningTotal,force)

    if seekOn:
        force = forceArray_seek[i]*multSeek
        self._accumulateForce(block,runningTotal,force)

    if arriveOn:
        force = forceArray_arrive[i]*multArrive
        self._accumulateForce(block,runningTotal,force)

    if wanderOn:
        force = forceArray_wander[i]*multWander
        self._accumulateForce(block,runningTotal,force, p=True)

    if headingOn:
        force = forceArray_heading[i]*multHeading
        self._accumulateForce(block,runningTotal,force)

    runningTotal.x = (runningTotal.x * scaleForce.x)
```

```
                    runningTotal.y = (runningTotal.y * scaleForce.y)
                    runningTotal.z = (runningTotal.z * scaleForce.z)
                    forceArray.append(runningTotal)
                else:
                    force = forceArray_pathFollow[i]*multPathFollow +
forceArray_box[i]*multBox + \
                    forceArray_followLeader[i]*multFollowLeader +
forceArray_neighborRepulse[i]*multNeighborRepulse + \
                    forceArray_avoidance[i]*multAvoid + forceArray_flee[i]*multFlee +
forceArray_seek[i]*multSeek + forceArray_heading[i]*multHeading + \
                    forceArray_separation[i]*multSeparation +
forceArray_alignment[i]*multAlignment + forceArray_cohesion[i]*multCohesion + \
                    forceArray_arrive[i]*multArrive + forceArray_wander[i]*multWander

                    force.x = (force.x * scaleForce.x)
                    force.y = (force.y * scaleForce.y)
                    force.z = (force.z * scaleForce.z)
                    forceArray.append(force)
        else:
            for i in range(points.length()):

                force = forceArray_pathFollow[i]*multPathFollow +
forceArray_box[i]*multBox + \
                forceArray_followLeader[i]*multFollowLeader +
forceArray_neighborRepulse[i]*multNeighborRepulse + forceArray_avoidance[i]*multAvoid + \
                forceArray_flee[i]*multFlee + forceArray_seek[i]*multSeek +
forceArray_heading[i]*multHeading + forceArray_separation[i]*multSeparation + \
                forceArray_alignment[i]*multAlignment +
forceArray_cohesion[i]*multCohesion + forceArray_arrive[i]*multArrive +
forceArray_wander[i]*multWander

                if weightTruncatedSumPrioritizationOn:
                    runningTotal = OpenMaya.MVector(0.0,0.0,0.0)

                    if surfaceProjectionOn:
                        force.x = (force.x * scaleForce.x) +
forceArray_ground[i].x*multGround
                        force.y = (force.y * scaleForce.y) +
forceArray_ground[i].y*multGround
                        force.z = (force.z * scaleForce.z)
                        self._accumulateForce(block,runningTotal[i],force)
                    else:
                        force.x = (force.x * scaleForce.x) +
forceArray_ground[i].x*multGround
                        force.y = (force.y * scaleForce.y) +
forceArray_ground[i].y*multGround
                        force.z = (force.z * scaleForce.z) +
forceArray_ground[i].z*multGround
                        self._accumulateForce(block,runningTotal,force)

                    if boxOn:
                        force = forceArray_box[i]*multBox
                        self._accumulateForce(block,runningTotal,force)

                    if obstacleAvoidanceOn:
                        force = forceArray_avoidance[i]*multAvoid
                        self._accumulateForce(block,runningTotal,force)

                    if neighborRepulseOn:
                        force = forceArray_neighborRepulse[i]*multNeighborRepulse
                        self._accumulateForce(block,runningTotal,force)

                    if fleeOn:
                        force = forceArray_flee[i]*multFlee
                        self._accumulateForce(block,runningTotal,force, p=True)

                    if followLeaderOn:
```

```
                    force = forceArray_followLeader[i]*multFollowLeader
                    self._accumulateForce(block,runningTotal,force)

                if separationOn:
                    force = forceArray_separation[i]*multSeparation
                    self._accumulateForce(block,runningTotal,force)

                if alignmentOn:
                    force = forceArray_alignment[i]*multAlignment
                    self._accumulateForce(block,runningTotal,force)

                if cohesionOn:
                    force = forceArray_cohesion[i]*multCohesion
                    self._accumulateForce(block,runningTotal,force)

                if pathFollowOn:
                    force = forceArray_pathFollow[i]*multPathFollow
                    self._accumulateForce(block,runningTotal,force)

                if seekOn:
                    force = forceArray_seek[i]*multSeek
                    self._accumulateForce(block,runningTotal,force)

                if arriveOn:
                    force = forceArray_arrive[i]*multArrive
                    self._accumulateForce(block,runningTotal,force)

                if wanderOn:
                    force = forceArray_wander[i]*multWander
                    self._accumulateForce(block,runningTotal,force)

                if headingOn:
                    force = forceArray_heading[i]*multHeading
                    self._accumulateForce(block,runningTotal,force)

                forceArray.append(runningTotal)
            else:
                force = forceArray_pathFollow[i]*multPathFollow +
forceArray_box[i]*multBox + \
                forceArray_followLeader[i]*multFollowLeader +
forceArray_neighborRepulse[i]*multNeighborRepulse + forceArray_avoidance[i]*multAvoid + \
                forceArray_flee[i]*multFlee + forceArray_seek[i]*multSeek +
forceArray_heading[i]*multHeading + forceArray_separation[i]*multSeparation + \
                forceArray_alignment[i]*multAlignment +
forceArray_cohesion[i]*multCohesion + forceArray_arrive[i]*multArrive +
forceArray_wander[i]*multWander
                if surfaceProjectionOn:
                    force.x = (force.x * scaleForce.x) +
forceArray_ground[i].x*multGround
                    force.y = (force.y * scaleForce.y) +
forceArray_ground[i].y*multGround
                    force.z = (force.z * scaleForce.z)
                else:
                    force.x = (force.x * scaleForce.x) +
forceArray_ground[i].x*multGround
                    force.y = (force.y * scaleForce.y) +
forceArray_ground[i].y*multGround
                    force.z = (force.z * scaleForce.z) +
forceArray_ground[i].z*multGround

                forceArray.append(force)


        if self._curParticleShape is not None:
            particleMFn = OpenMayaFX.MFnParticleSystem(self._curParticleShape)
            particleMFn.setPerParticleAttribute("rotationPP",outXYZRotationArray)
```

```
def _accumulateForce(self, block, RunningTotal, ForceToAdd, p=False):

    maxForce = self._attrDoubleValue(block, SteeringField.aMaxForce)

    forceUsed = RunningTotal.length()

    forceRemaining = maxForce - forceUsed

    forceToApply = ForceToAdd.length()

    if forceToApply < forceRemaining:
        RunningTotal.x = RunningTotal.x + ForceToAdd.x
        RunningTotal.y = RunningTotal.y + ForceToAdd.y
        RunningTotal.z = RunningTotal.z + ForceToAdd.z
    else:
        RunningTotal.z = RunningTotal.x + (ForceToAdd.normal().x * forceRemaining)
        RunningTotal.y = RunningTotal.y + (ForceToAdd.normal().y * forceRemaining)
        RunningTotal.z = RunningTotal.z + (ForceToAdd.normal().z * forceRemaining)

    return RunningTotal

def compute(self, plug, block):
    """
    Compute output force.
    """
    outputForce = OpenMayaMPx.cvar.MPxFieldNode_mOutputForce
    mDeltaTime = OpenMayaMPx.cvar.MPxFieldNode_mDeltaTime

    if not (plug == outputForce):
        return OpenMaya.MStatus.kUnknownParameter

    '''
    get the logical index of the element this plug refers to.
    '''
    try:
        multiIndex = plug.logicalIndex()
    except:
        statusError("ERROR in plug.logicalIndex.")

    self._particleShapes = []
    thisNode = self.thisMObject()
    mplug = OpenMaya.MPlug(thisNode, outputForce)
    for i in range(mplug.numElements()):
        eplug = mplug[i]
        plugArray = OpenMaya.MPlugArray()
        if (eplug.connectedTo(plugArray,True,True)):
            if eplug.logicalIndex() == multiIndex:
                self._curParticleShape = plugArray[0].node()
            self._particleShapes.append(plugArray[0].node())

    '''
    Get input data handle
    '''
    inputData = OpenMayaMPx.cvar.MPxFieldNode_mInputData
    try:
        hInputArray = block.outputArrayValue(inputData)
    except:
        statusError("ERROR in hInputArray = block.outputArrayValue().")

    try:
        hInputArray.jumpToElement(multiIndex)
    except:
        statusError("ERROR: hInputArray.jumpToElement failed.")

    '''
    get children of aInputData.
    '''
```

```
try:
    hCompond = hInputArray.inputValue()
except:
    statusError("ERROR in hCompond=hInputArray.inputValue")

inputPositions = OpenMayaMPx.cvar.MPxFieldNode_mInputPositions

hPosition = hCompond.child(inputPositions)
dPosition = hPosition.data()
fnPosition = OpenMaya.MFnVectorArrayData(dPosition)
try:
    points = fnPosition.array()
    key = OpenMaya.MFnDagNode(self._curParticleShape).fullPathName()
    self._particlePositions[key] = points
except:
    statusError("ERROR in fnPosition.array(), not find points.")

inputVelocities = OpenMayaMPx.cvar.MPxFieldNode_mInputVelocities
hVelocity = hCompond.child(inputVelocities)
dVelocity = hVelocity.data()
fnVelocity = OpenMaya.MFnVectorArrayData(dVelocity)
try:
    velocities = fnVelocity.array()
    key = OpenMaya.MFnDagNode(self._curParticleShape).fullPathName()
    self._particleVelocities[key] = velocities
except:
    statusError("ERROR in fnVelocity.array(), not find velocities.")

inputMass = OpenMayaMPx.cvar.MPxFieldNode_mInputMass

# this version of the plugin does not support mass
hMass = hCompond.child(inputMass)
dMass = hMass.data()
fnMass = OpenMaya.MFnDoubleArrayData(dMass)
try:
    masses = fnMass.array()
except:
    statusError("ERROR in fnMass.array(), not find masses.")

#hDeltaTime = hCompond.child(mDeltaTime)
#deltaTime = hDeltaTime.asTime().value()

'''
handle to mhInputPPData
Early version was storing certain data in the particleShape.
I might revisit this again
'''
#mInputPPData  = OpenMayaMPx.cvar.MPxFieldNode_mInputPPData
#try:
#    mhInputPPData = block.inputArrayValue(mInputPPData)
#except:
#    statusError("ERROR in mhInputPPData = block.inputArrayValue().")
#try:
#    status = mhInputPPData.jumpToElement(multiIndex)
#except:
#    statusError("ERROR: mhInputPPData.jumpToElement failed.")
#try:
#    hInputPPData = mhInputPPData.inputValue()
#except:
#    statusError("ERROR in hInputPPData = mhInputPPData.inputValue.")

#dInputPPData = hInputPPData.data()
#inputPPArray = OpenMaya.MFnArrayAttrsData(dInputPPData)

#arrayExist = True #inputPPArray.checkArrayExist("wanderTheta", doubleType)
#if arrayExist:
#     ThetaArray = inputPPArray.getDoubleData("wanderTheta")
```

```
        #
        #arrayExist = True #inputPPArray.checkArrayExist("wanderTarget",
OpenMaya.MFnArrayAttrsData.kVectorArray)
        #if arrayExist:
        #    wanderTargetArray = inputPPArray.getVectorData("wanderTarget")

        #arrayExist = True #inputPPArray.checkArrayExist("wanderRay",
OpenMaya.MFnArrayAttrsData.kVectorArray)
        #if arrayExist:
        #    wanderRayArray = inputPPArray.getVectorData("wanderRay")


        if self._restPositions is None:
            self._restPositions = OpenMaya.MVectorArray(points.length())
            self._restPositions.copy(points)

        self.position = points
        self.velocity = velocities

        '''
        Compute the output force.
        '''
        forceArray = OpenMaya.MVectorArray()
        targetArray = OpenMaya.MVectorArray()

        self._calculateForce(block, points, velocities, forceArray)

        '''
        get output data handle
        '''
        try:
            hOutArray = block.outputArrayValue(outputForce)
        except:
            statusError("ERROR in hOutArray = block.outputArrayValue.")
        try:
            bOutArray = hOutArray.builder()
        except:
            statusError("ERROR in bOutArray = hOutArray.builder.")

        '''
        get output force array from block.
        '''
        try:
            hOut = bOutArray.addElement(multiIndex)
        except:
            statusError("ERROR in hOut = bOutArray.addElement.")

        fnOutputForce = OpenMaya.MFnVectorArrayData()
        try:
            dOutputForce = fnOutputForce.create(forceArray)
        except:
            statusError("ERROR in dOutputForce = fnOutputForce.create")

        '''
        update data block with new output force data.
        '''
        hOut.setMObject(dOutputForce)
        block.setClean(plug)

        zeroOverlapOn = self._attrBooleanValue(block, SteeringField.aZeroOverlapOn)
        if zeroOverlapOn:
            self._zeroOverlap(block)

    def _boxAvoidance(self, block, positions, velocities, outputForce):
        if positions.length() != velocities.length():
            return
        outputForce.clear()
```

```
        maxForce = self._attrDoubleValue(block, SteeringField.aMaxForce)
        maxSpeed = self._attrDoubleValue(block, SteeringField.aMaxSpeed)
        bRadius = self._attrDoubleValue(block, SteeringField.aBoundRadius) # boid
bounding sphere radius
        feelerLength = self._attrDoubleValue(block, SteeringField.aFeelerLength)
        width = self._attrDoubleValue(block, SteeringField.aBoxWidth)
        height = self._attrDoubleValue(block, SteeringField.aBoxHeight)
        depth = self._attrDoubleValue(block, SteeringField.aBoxDepth)

        thisNode = self.thisMObject()
        nodeFn = OpenMaya.MFnDependencyNode(thisNode)

        tx = nodeFn.findPlug("translateX").asDouble()
        ty = nodeFn.findPlug("translateY").asDouble()
        tz = nodeFn.findPlug("translateZ").asDouble()

        sx = nodeFn.findPlug("scaleX").asDouble()
        sy = nodeFn.findPlug("scaleY").asDouble()
        sz = nodeFn.findPlug("scaleZ").asDouble()

        width =  width * sx
        depth =  depth * sy
        height=  height * sz

        for i in range(positions.length()):

            steeringForce = OpenMaya.MVector(0.0,0.0,0.0)

            if (positions[i].x+bRadius) > tx + width/2.0:
               steeringForce = OpenMaya.MVector.xAxis*-1.0
            elif (positions[i].x+bRadius) <  tx + width/-2.0:
               steeringForce = OpenMaya.MVector.xAxis

            if (positions[i].y+bRadius) > ty + depth/2.0:
                steeringForce = OpenMaya.MVector.yAxis*-1.0
            elif (positions[i].y+bRadius) < ty + depth/-2.0:
               steeringForce = OpenMaya.MVector.yAxis

            if (positions[i].z+bRadius) > tz + height/2.0:
                steeringForce = OpenMaya.MVector.zAxis*-1.0
            elif (positions[i].z+bRadius) < tz + height/-2.0:
               steeringForce = OpenMaya.MVector.zAxis

            outputForce.append(self._seekHelper(block, steeringForce, velocities[i]))

    def _surfaceFollow(self, block, positions, velocities, outputForce):

        thisNode = self.thisMObject()
        nodeFn = OpenMaya.MFnDependencyNode(thisNode)
        g_plug = nodeFn.findPlug("groundInputSurface")

        if not g_plug.isConnected():
            return

        if positions.length() != velocities.length():
            return

        dataVectorArrayFn = OpenMaya.MFnVectorArrayData()
        data = block.outputValue(SteeringField.aUpDirection).data();
        dataVectorArrayFn.setObject(data)
        outUpDirectionArray = dataVectorArrayFn.array()
        outUpDirectionArray.clear()
        outUpDirectionArray.setLength(positions.length())

        dataVectorArrayFn = OpenMaya.MFnVectorArrayData()
        data = block.outputValue(SteeringField.aXYZRotation).data();
```

```
        dataVectorArrayFn.setObject(data)
        outXYZRotationArray = dataVectorArrayFn.array()
        outXYZRotationArray.clear()
        outXYZRotationArray.setLength(positions.length())

        groundOffset = self._attrDoubleValue(block, SteeringField.aGroundOffset)
        feelerLength = self._attrDoubleValue(block, SteeringField.aFeelerLength)
        flipUpVectorOn = self._attrBooleanValue(block, SteeringField.aFlipUpVectorOn)
        groundforecastBySpeedOn = self._attrBooleanValue(block,
SteeringField.aGroundforecastBySpeedOn)

        outputForce.clear()

        for i in range(positions.length()):
            unitVelocity = OpenMaya.MVector(velocities[i])
            mag = unitVelocity.length()
            unitVelocity.normalize()
            forecastPosition = None

            # perdict position of boid
            if groundforecastBySpeedOn:
                # by speed
                forecastPosition = OpenMaya.MPoint(positions[i] + (velocities[i]))
            else:
                # by feeler length
                forecastPosition  = OpenMaya.MPoint(positions[i] +
(unitVelocity*(feelerLength)))

            connections = OpenMaya.MPlugArray()
            g_plug.connectedTo(connections, True, False)
            groundPlug = connections[0]
            nNode = groundPlug.node()

            mfnNurb = OpenMaya.MFnNurbsSurface(nNode)
            uIP = OpenMaya.MScriptUtil().asDoublePtr()
            vIP = OpenMaya.MScriptUtil().asDoublePtr()
            closestSurfacePoint = mfnNurb.closestPoint(forecastPosition,uIP,vIP)

            overshoot = OpenMaya.MVector(forecastPosition-closestSurfacePoint).length()

            u = OpenMaya.MScriptUtil().getDouble(uIP)
            v = OpenMaya.MScriptUtil().getDouble(vIP)

            uvPerpend = mfnNurb.normal(u,v)

            outUpDirectionArray[i].x = uvPerpend.x
            outUpDirectionArray[i].y = uvPerpend.y
            outUpDirectionArray[i].z = uvPerpend.z

            if flipUpVectorOn:
                x = unitVelocity
                y = x^uvPerpend # y
                z = y^x #z
            else:
                x = unitVelocity
                y = uvPerpend^x # y
                z = x^y #z

            MatComponents = (x.x, x.y, x.z, 0.0, y.x, y.y, y.z, 0.0, z.x, z.y, z.z, 0.0,
0.0, 0.0, 0.0, 1.0)
            matrix = OpenMaya.MMatrix()
            OpenMaya.MScriptUtil.createMatrixFromList( MatComponents, matrix)
            mtm = OpenMaya.MTransformationMatrix(matrix)
            erot = mtm.eulerRotation()
            rot = erot.asVector()
            mangle = OpenMaya.MAngle()
```

```
            mangle.setValue(rot.x)
            x = mangle.asDegrees()

            mangle.setValue(rot.y)
            y = mangle.asDegrees()

            mangle.setValue(rot.z)
            z = mangle.asDegrees()

            outXYZRotationArray[i].x = x
            outXYZRotationArray[i].y = y
            outXYZRotationArray[i].z = z

            if(overshoot <= groundOffset):
                outputForce.append(self._surfaceFollowHelper(block, uvPerpend,
velocities[i]))
            else:
                desiredVel = OpenMaya.MVector(closestSurfacePoint - positions[i])
                outputForce.append(self._surfaceFollowHelper(block, desiredVel,
velocities[i]))


    def _surfaceProjection(self, block, positions, velocities):

        thisNode = self.thisMObject()
        nodeFn = OpenMaya.MFnDependencyNode(thisNode)
        g_plug = nodeFn.findPlug("groundInputSurface")

        if not g_plug.isConnected():
            return

        if positions.length() != velocities.length():
            return

        if self._curParticleShape is None:
            return

        dataVectorArrayFn = OpenMaya.MFnVectorArrayData()
        data = block.outputValue(SteeringField.aUpDirection).data();
        dataVectorArrayFn.setObject(data)
        outUpDirectionArray = dataVectorArrayFn.array()
        outUpDirectionArray.clear()
        outUpDirectionArray.setLength(positions.length())

        dataVectorArrayFn = OpenMaya.MFnVectorArrayData()
        data = block.outputValue(SteeringField.aXYZRotation).data();
        dataVectorArrayFn.setObject(data)
        outXYZRotationArray = dataVectorArrayFn.array()
        outXYZRotationArray.clear()
        outXYZRotationArray.setLength(positions.length())

        dataVectorArrayFn = OpenMaya.MFnVectorArrayData()
        data = block.outputValue(SteeringField.aXYZProjection).data();
        dataVectorArrayFn.setObject(data)
        outXYZProjectionArray = dataVectorArrayFn.array()
        outXYZProjectionArray.clear()
        outXYZProjectionArray.setLength(positions.length())

        surfaceProjectionOffset = self._attrDoubleValue(block,
SteeringField.aSurfaceProjectionOffset)
        feelerLength = self._attrDoubleValue(block, SteeringField.aFeelerLength)
        projectionWithVelocityOn = self._attrBooleanValue(block,
SteeringField.aProjectionWithVelocityOn)
        flipUpVectorOn = self._attrBooleanValue(block, SteeringField.aFlipUpVectorOn)

        for i in range(positions.length()):
            unitVelocity = OpenMaya.MVector(velocities[i])
```

```
            mag = unitVelocity.length()
            unitVelocity.normalize()

            # perdict position of boid
            if projectionWithVelocityOn:
                forecastPosition  = OpenMaya.MPoint(positions[i] +
(unitVelocity*(feelerLength+mag)))
            else:
                forecastPosition = OpenMaya.MPoint(positions[i])


            connections = OpenMaya.MPlugArray()
            g_plug.connectedTo(connections, True, False)
            groundPlug = connections[0]
            nNode = groundPlug.node()

            mfnNurb = OpenMaya.MFnNurbsSurface(nNode)
            uIP = OpenMaya.MScriptUtil().asDoublePtr()
            vIP = OpenMaya.MScriptUtil().asDoublePtr()
            closestSurfacePoint = mfnNurb.closestPoint(forecastPosition,uIP,vIP)

            u = OpenMaya.MScriptUtil().getDouble(uIP)
            v = OpenMaya.MScriptUtil().getDouble(vIP)

            uvPerpend = mfnNurb.normal(u,v)

            outUpDirectionArray[i].x = uvPerpend.x
            outUpDirectionArray[i].y = uvPerpend.y
            outUpDirectionArray[i].z = uvPerpend.z

            offset = OpenMaya.MVector(uvPerpend)*surfaceProjectionOffset

            outXYZProjectionArray[i].x = closestSurfacePoint.x + offset.x
            outXYZProjectionArray[i].y = closestSurfacePoint.y + offset.y
            outXYZProjectionArray[i].z = closestSurfacePoint.z + offset.z

            if flipUpVectorOn:
                x = unitVelocity
                y = x^uvPerpend # y
                z = y^x #z
            else:
                x = unitVelocity
                y = uvPerpend^x # y
                z = x^y #z

            MatComponents = (x.x, x.y, x.z, 0.0, y.x, y.y, y.z, 0.0, z.x, z.y, z.z, 0.0,
0.0, 0.0, 0.0, 1.0)
            matrix = OpenMaya.MMatrix()
            OpenMaya.MScriptUtil.createMatrixFromList( MatComponents, matrix)
            mtm = OpenMaya.MTransformationMatrix(matrix)
            erot = mtm.eulerRotation()
            rot = erot.asVector()
            mangle = OpenMaya.MAngle()

            mangle.setValue(rot.x)
            x = mangle.asDegrees()

            mangle.setValue(rot.y)
            y = mangle.asDegrees()

            mangle.setValue(rot.z)
            z = mangle.asDegrees()

            outXYZRotationArray[i].x = x
            outXYZRotationArray[i].y = y
            outXYZRotationArray[i].z = z
```

```
        particleMFn = OpenMayaFX.MFnParticleSystem(self._curParticleShape)
        particleMFn.setPerParticleAttribute("position",outXYZProjectionArray)

    def createFeelers(self, pos, vel, maxSpeed):
        feelers = []

        thisNode = self.thisMObject()
        plug = OpenMaya.MPlug(thisNode,SteeringField.aFeelerLength)
        feelerLength = plug.asDouble();
        feelerLength =  feelerLength + (vel.length()/maxSpeed)*feelerLength

        localTarget = vel.normal()
        heading = localTarget
        loc = pos
        heading = heading * feelerLength
        feeler = loc + heading
        feelers.append(feeler)

        heading = localTarget
        heading =  heading.rotateBy(OpenMaya.MVector.kZaxis, math.pi/4.0)
        feeler = loc + heading*feelerLength*0.6
        feelers.append(feeler)

        heading = localTarget
        heading =  heading.rotateBy(OpenMaya.MVector.kZaxis, -math.pi/4.0)
        feeler = loc + heading*feelerLength*0.6
        feelers.append(feeler)

        return feelers

    def _followLeader(self, block, positions, velocities, outputForce):
        if positions.length() != velocities.length():
            return
        outputForce.clear()

        numBoids = positions.length()

        followLeaderMaintainOffsetOn = self._attrBooleanValue(block,
SteeringField.aFollowLeaderMaintainOffsetOn)
        maxForce = self._attrDoubleValue(block, SteeringField.aMaxForce)
        maxSpeed = self._attrDoubleValue(block, SteeringField.aMaxSpeed)
        leaderID = self._attrLongValue(block, SteeringField.aFollowLeaderID)
        bRadius = self._attrDoubleValue(block, SteeringField.aBoundRadius) # boid
bounding sphere radius
        followLeaderOffsetVHandle = block.inputValue(SteeringField.aFollowLeaderOffset)
        followLeaderOffsetV = OpenMaya.MVector(followLeaderOffsetVHandle.asFloatVector())

        def doSteer(leaderID, followerID):

                currentVel = velocities[followerID]
                leaderVelocity = velocities[leaderID]
                leaderSpeed = leaderVelocity.length()

                dist = (followLeaderOffsetV.length() + 2*bRadius)

                if(followLeaderMaintainOffsetOn):
                    toDir =  positions[leaderID] - positions[followerID]
                    toDir.normalize()
                    offsetV = toDir*dist

                    worldOffsetPos = positions[leaderID] - offsetV
                    ToOffset = worldOffsetPos - positions[followerID]
                    lookAheadTime = ToOffset.length()/(maxSpeed  + leaderSpeed)
                    target = worldOffsetPos + leaderVelocity*lookAheadTime
                    desiredVel = target - positions[followerID]

                    steeringForce = OpenMaya.MVector(0.0,0.0,0.0)
```

```
                    arriveProximityRadius = self._attrDoubleValue(block,
SteeringField.aArriveProximityRadius)
                    decelerationMult = self._attrDoubleValue(block,
SteeringField.aDecelerationMult)

                    speed = min(lookAheadTime, maxSpeed)

                    lookAheadTime = math.fabs(lookAheadTime)
                    dir =  positions[leaderID] - positions[followerID]
                    dist = dir.length()
                    if dist > arriveProximityRadius:
                        desiredVel.normalize()
                        speed = leaderSpeed + (dist * lookAheadTime)

                    desiredVel = desiredVel * speed

                    steeringForce = (desiredVel- currentVel)
                    steeringForce = self._limit(steeringForce, maxForce)
                else:
                     decelerationMult = self._attrDoubleValue(block,
SteeringField.aDecelerationMult)
                     decelerationDamping = self._attrDoubleValue(block,
SteeringField.aDecelerationDamping)
                     offsetDir =  followLeaderOffsetV.normal()
                     offsetV = offsetDir*dist
                     worldOffsetPos = positions[leaderID] + offsetV
                     ToOffset = worldOffsetPos - positions[followerID]
                     lookAheadTime = ToOffset.length()/(maxSpeed  + leaderSpeed)
                     target = worldOffsetPos + leaderVelocity*lookAheadTime
                     desiredVel = target - positions[followerID]

                     speed = desiredVel.length()/(decelerationMult*decelerationDamping)
                     speed = min(speed, maxSpeed)/ desiredVel.length()

                     desiredVel = desiredVel * speed

                     steeringForce = desiredVel - currentVel
                     steeringForce = self._limit(steeringForce, maxForce)

                return steeringForce

        leaderID = self.getValidLeaderID(leaderID, numBoids)

        if leaderID < 0:
            leaderID = 0

        nextLeader = leaderID
        for i in range(numBoids):
            if i != leaderID:
                steeringForce = doSteer(nextLeader,i)
                outputForce.append(steeringForce)
                nextLeader = i
            else:
                outputForce.append(OpenMaya.MVector(0.0,0.0,0.0))

    def getValidLeaderID(self, leaderID, numBoids):

        if leaderID>(numBoids-1):
            leaderID = numBoids-1

        return leaderID


    def _zeroOverlap(self, block):

        bRadius = self._attrDoubleValue(block, SteeringField.aBoundRadius) # boid
```

```
bounding sphere radius
        multZeroOverlap = self._attrDoubleValue(block, SteeringField.aZeroOverlap)

        inputVArrayTable = {}
        outputVArrayTable = {}

        id = 0

        for pshape in self._particleShapes:
            particleMFn = OpenMayaFX.MFnParticleSystem(pshape)
            posArray = OpenMaya.MVectorArray()
            particleMFn.getPerParticleAttribute("position",posArray)
            outputVArrayTable[pshape] = posArray
            for i in range(posArray.length()):
                inputVArrayTable[id] = posArray[i]
                id = id + 1

        idList = inputVArrayTable.keys()

        for i in idList:
            curBoid = inputVArrayTable[i]
            for j in idList:
                otherBoidPosition = inputVArrayTable[j]
                if i != j:

                    toBoid = curBoid - otherBoidPosition
                    distFromEachOther = toBoid.length()
                    if distFromEachOther > 0.001:
                        overlap = 2*bRadius - distFromEachOther
                        if overlap > 0:
                            newPos = OpenMaya.MVector(curBoid)
                            newPos = newPos +
(toBoid*((1.0/distFromEachOther)*overlap)*multZeroOverlap)
                            curBoid.x = newPos.x
                            curBoid.y = newPos.y
                            curBoid.z = newPos.z

        for pshape in outputVArrayTable.keys():
            particleMFn = OpenMayaFX.MFnParticleSystem(pshape)
            posArray = outputVArrayTable[pshape]
            particleMFn.setPerParticleAttribute("position",posArray)

    def _neighborRepulse(self, block, positions, velocities, neighbors, outputForce):

            neighborRepulseAgainstAllOn = self._attrBooleanValue(block,
SteeringField.aNeighborRepulseAgainstAllOn)

            if neighborRepulseAgainstAllOn:
                self._neighborRepulseAgainstAll(block, positions, velocities, neighbors,
outputForce)
            else:
                self._neighborRepulseCurrentOnly(block, positions, velocities, neighbors,
outputForce)

    def _neighborRepulseCurrentOnly(self, block, positions, velocities, neighbors,
outputForce):
        if positions.length() != velocities.length():
            return

        maxSpeed = self._attrDoubleValue(block, SteeringField.aMaxSpeed)
        maxForce = self._attrDoubleValue(block, SteeringField.aMaxForce)
        leaderID = self._attrLongValue(block, SteeringField.aFollowLeaderID)
        forecastBySpeedOn = self._attrBooleanValue(block,
SteeringField.aforecastBySpeedOn)
        bRadius = self._attrDoubleValue(block, SteeringField.aBoundRadius) # boid
bounding sphere radius
```

```
        outputForce.clear()

        boidList = range(positions.length())

        for i in boidList:
            curBoid = positions[i]
            desiredVel = OpenMaya.MVector(0.0,0.0,0.0)
            neighborsList = neighbors[i]

            unitVelocity = OpenMaya.MVector(velocities[i])
            unitVelocity.normalize()

            # perdict future position of current boid
            if(forecastBySpeedOn):
                forecastPosition  = OpenMaya.MVector(curBoid + velocities[i])
            else:
                forecastPosition  = OpenMaya.MVector(curBoid + unitVelocity*bRadius)
            for j in neighborsList:
                if i != j:
                    otherBoidPosition = positions[j]
                    toBoid = forecastPosition - otherBoidPosition
                    distFromEachOther = toBoid.length()
                    overlap = 2*bRadius - distFromEachOther

                    if overlap >= 0:
                        toBoid.normalize()
                        desiredVel = desiredVel + (toBoid)* overlap

            # average desired velocity
            if leaderID>0 and i == leaderID:
                outputForce.append(OpenMaya.MVector(0.0,0.0,0.0))
            else:
                outputForce.append(self._arriveHelper(block, desiredVel, velocities[i]))

    def _neighborRepulseAgainstAll(self, block, positions, velocities, neighbors,
outputForce):

        if self._curParticleShape is None:
            return

        maxSpeed = self._attrDoubleValue(block, SteeringField.aMaxSpeed)
        maxForce = self._attrDoubleValue(block, SteeringField.aMaxForce)
        leaderID = self._attrLongValue(block, SteeringField.aFollowLeaderID)
        forecastBySpeedOn = self._attrBooleanValue(block,
SteeringField.aforecastBySpeedOn)
        bRadius = self._attrDoubleValue(block, SteeringField.aBoundRadius) # boid
bounding sphere radius
        nRadius = self._attrDoubleValue(block, SteeringField.aReplusionNeighborDistance)

        outputForce.clear()

        inputVPosArrayTable0 = {}
        inputVPosArrayTable1 = {}

        inputVVelArrayTable0 = {}
        inputVVelArrayTable1 = {}

        id = 0

        particleMFn = OpenMayaFX.MFnParticleSystem(self._curParticleShape)
        posArray = OpenMaya.MVectorArray()
        velArray = OpenMaya.MVectorArray()
        particleMFn.getPerParticleAttribute("position",posArray)
        particleMFn.getPerParticleAttribute("velocity",velArray)
        count = particleMFn.count()
        for i in range(posArray.length()):
            inputVPosArrayTable0[id] = OpenMaya.MVector(posArray[i])
```

```
            inputVPosArrayTable1[id] = OpenMaya.MVector(posArray[i])
            inputVVelArrayTable0[id] = OpenMaya.MVector(velArray[i])
            inputVVelArrayTable1[id] = OpenMaya.MVector(velArray[i])
            id = id + 1

        curParticleDagPath = OpenMaya.MFnDagNode(self._curParticleShape).fullPathName()

        for pshape in self._particleShapes:
            key = OpenMaya.MFnDagNode(pshape).fullPathName()
            if curParticleDagPath != key:
                if key in self._particlePositions.keys():
                    other_positions = self._particlePositions[key]
                    other_velocities = self._particleVelocities[key]
                    for i in range(other_positions.length()):
                        inputVPosArrayTable1[id] = OpenMaya.MVector(other_positions[i])
                        inputVVelArrayTable1[id] = OpenMaya.MVector(other_velocities[i])
                        id = id + 1

        idList0 = inputVPosArrayTable0.keys()
        idList1 = inputVPosArrayTable1.keys()

        for i in idList0:
            curBoid = inputVPosArrayTable0[i]
            desiredVel = OpenMaya.MVector(0.0,0.0,0.0)

            unitVelocity = OpenMaya.MVector(inputVVelArrayTable0[i])
            unitVelocity.normalize()

            # perdict future position of current boid
            if(forecastBySpeedOn):
                forecastPosition  = OpenMaya.MVector(curBoid + inputVVelArrayTable0[i])
            else:
                forecastPosition  = OpenMaya.MVector(curBoid + unitVelocity*bRadius)
            for j in idList1:
                if i != j:
                    otherBoidPosition = inputVPosArrayTable1[j]

                    distance = (otherBoidPosition  - curBoid).length()

                    if distance <= nRadius:

                        toBoid = forecastPosition - otherBoidPosition
                        distFromEachOther = toBoid.length()
                        overlap = 2*bRadius - distFromEachOther

                        if overlap >= 0:
                            toBoid.normalize()
                            desiredVel = desiredVel + (toBoid)* overlap

            if leaderID>0 and i == leaderID:
                outputForce.append(OpenMaya.MVector(0.0,0.0,0.0))
            else:
                outputForce.append(self._arriveHelper(block, desiredVel,
inputVVelArrayTable0[i]))


    def _debugCollisions(self, view, path, style, status):

        view.beginGL()
        glFT.glPushAttrib(OpenMayaRender.MGL_POLYGON_BIT)
        glFT.glPolygonMode(OpenMayaRender.MGL_FRONT_AND_BACK, OpenMayaRender.MGL_LINE)

        thisNode = self.thisMObject()
        nodeFn = OpenMaya.MFnDependencyNode(thisNode)
        tx = nodeFn.findPlug("translateX").asDouble()
        ty = nodeFn.findPlug("translateY").asDouble()
        tz = nodeFn.findPlug("translateZ").asDouble()
```

```
    s = (self._debug_projQ)

    a = self._debug_A
    P = self._debug_P + a

    s = s + a
    s2 = self._debug_perpQ*-1.0 + s

    v = a + self.debug_curVel

    a.x = a.x - tx
    a.y = a.y - ty
    a.z = a.z - tz

    P.x = P.x - tx
    P.y = P.y - ty
    P.z = P.z - tz

    v.x = v.x - tx
    v.y = v.y - ty
    v.z = v.z - tz

    s.x = s.x - tx
    s.y = s.y - ty
    s.z = s.z - tz

    s2.x = s2.x - tx
    s2.y = s2.y - ty
    s2.z = s2.z - tz
    glFT.glLineWidth(3.0)
    glFT.glBegin(OpenMayaRender.MGL_LINES)
    glFT.glColor3f(1.0, 0.0, 0.0)
    glFT.glVertex3f(a.x, a.y, a.z)
    glFT.glVertex3f(P.x, P.y, P.z)
    glFT.glEnd()

    glFT.glBegin(OpenMayaRender.MGL_LINES)
    glFT.glColor3f(0.0, 1.0, 0.0)
    glFT.glVertex3f(a.x, a.y, a.z)
    glFT.glVertex3f(v.x, v.y, v.z)
    glFT.glEnd()

    glFT.glBegin(OpenMayaRender.MGL_LINES)
    glFT.glColor3f(1.0, 1.0, 0.0)
    glFT.glVertex3f(s.x, s.y, s.z)
    glFT.glVertex3f(s2.x, s2.y, s2.z)
    glFT.glEnd()

    glFT.glLineWidth(1.0)

    glFT.glPopAttrib()
    view.endGL()


def draw(self, view, path, style, status):

    curFrame = self._animControl.currentTime().value()
    minFrame = self._animControl.minTime().value()

    thisNode = self.thisMObject()

    if curFrame == minFrame:
        plug = OpenMaya.MPlug(thisNode,SteeringField.aSeed)
        seed = width = plug.asDouble()
        random.seed(seed)
        if self._curParticleShape is not None:
```

```
        particleMFn = OpenMayaFX.MFnParticleSystem(self._curParticleShape)
        size = particleMFn.count()
        self._currentWayPoint = [int(i*0) for i in range(size)]

view.beginGL()
glFT.glPushAttrib(OpenMayaRender.MGL_POLYGON_BIT)
glFT.glPolygonMode(OpenMayaRender.MGL_FRONT_AND_BACK, OpenMayaRender.MGL_LINE)

plug = OpenMaya.MPlug(thisNode,SteeringField.aBoxWidth)
width = plug.asDouble()

plug = OpenMaya.MPlug(thisNode,SteeringField.aBoxHeight)
height = plug.asDouble()

plug = OpenMaya.MPlug(thisNode,SteeringField.aBoxDepth)
depth = plug.asDouble()

plug = OpenMaya.MPlug(thisNode,SteeringField.aShowBox)
showBox = plug.asBool()

plug = OpenMaya.MPlug(thisNode,SteeringField.aObstacleAvoidanceDebugOn)
obstacleAvoidanceDebugOn = plug.asBool()

nodeFn = OpenMaya.MFnDependencyNode(thisNode)

sx = nodeFn.findPlug("scaleX").asDouble()
sy = nodeFn.findPlug("scaleY").asDouble()
sz = nodeFn.findPlug("scaleZ").asDouble()

width =  width * sx
depth =  depth * sy
height=  height * sz

if showBox:
    #bottom
    glFT.glLineWidth(3.0)
    glFT.glBegin(OpenMayaRender.MGL_LINES)
    glFT.glColor3f(1.0, 0.0, 1.0)
    glFT.glVertex3f(-width/2.0,-depth/2.0,-height/2.0)
    glFT.glVertex3f(width/2.0,-depth/2.0,-height/2.0)
    glFT.glEnd()

    glFT.glBegin(OpenMayaRender.MGL_LINES)
    glFT.glVertex3f(-width/2.0,depth/2.0,-height/2.0)
    glFT.glVertex3f(width/2.0,depth/2.0,-height/2.0)
    glFT.glEnd()

    glFT.glBegin(OpenMayaRender.MGL_LINES)
    glFT.glVertex3f(-width/2.0,depth/2.0,-height/2.0)
    glFT.glVertex3f(-width/2.0,-depth/2.0,-height/2.0)
    glFT.glEnd()

    glFT.glBegin(OpenMayaRender.MGL_LINES)
    glFT.glVertex3f(width/2.0,depth/2.0,-height/2.0)
    glFT.glVertex3f(width/2.0,-depth/2.0,-height/2.0)

    #top
    glFT.glBegin(OpenMayaRender.MGL_LINES)
    glFT.glVertex3f(-width/2.0,-depth/2.0,height/2.0)
    glFT.glVertex3f(width/2.0,-depth/2.0,height/2.0)
    glFT.glEnd()

    glFT.glBegin(OpenMayaRender.MGL_LINES)
    glFT.glVertex3f(-width/2.0,depth/2.0,height/2.0)
    glFT.glVertex3f(width/2.0,depth/2.0,height/2.0)
    glFT.glEnd()
```

```
            glFT.glBegin(OpenMayaRender.MGL_LINES)
            glFT.glVertex3f(-width/2.0,depth/2.0,height/2.0)
            glFT.glVertex3f(-width/2.0,-depth/2.0,height/2.0)
            glFT.glEnd()

            glFT.glBegin(OpenMayaRender.MGL_LINES)
            glFT.glVertex3f(width/2.0,depth/2.0,height/2.0)
            glFT.glVertex3f(width/2.0,-depth/2.0,height/2.0)
            glFT.glEnd()

            #Columns
            glFT.glBegin(OpenMayaRender.MGL_LINES)
            glFT.glVertex3f(-width/2.0,-depth/2.0,-height/2.0)
            glFT.glVertex3f(-width/2.0,-depth/2.0,height/2.0)
            glFT.glEnd()

            glFT.glBegin(OpenMayaRender.MGL_LINES)
            glFT.glVertex3f(width/2.0,depth/2.0,-height/2.0)
            glFT.glVertex3f(width/2.0,depth/2.0,height/2.0)
            glFT.glEnd()

            glFT.glBegin(OpenMayaRender.MGL_LINES)
            glFT.glVertex3f(-width/2.0,depth/2.0,-height/2.0)
            glFT.glVertex3f(-width/2.0,depth/2.0,height/2.0)
            glFT.glEnd()

            glFT.glBegin(OpenMayaRender.MGL_LINES)
            glFT.glVertex3f(width/2.0,-depth/2.0,-height/2.0)
            glFT.glVertex3f(width/2.0,-depth/2.0,height/2.0)
            glFT.glEnd()
            glFT.glLineWidth(1.0)

        glFT.glPopAttrib()
        view.endGL()

        if obstacleAvoidanceDebugOn:
            self._debugCollisions(view, path, style, status)

    def _obstacleAvoidanceHelper(self, block, pos, vel, i):

        maxSpeed = self._attrDoubleValue(block, SteeringField.aMaxSpeed)
        maxForce = self._attrDoubleValue(block, SteeringField.aMaxForce)
        brakingWeight = self._attrDoubleValue(block, SteeringField.aBrakeWeight) # boid
brake weight
        bRadius = self._attrDoubleValue(block, SteeringField.aBoundRadius) # boid
bounding sphere radius

        steeringForce = OpenMaya.MVector(0.0,0.0,0.0)

        # just look for spBoidColliderSphere nodes instead of maya standard nodes
        it = OpenMaya.MItDependencyNodes(OpenMaya.MFn.kPluginDependNode)
        while(not it.isDone()):
                fn = OpenMaya.MFnDependencyNode(it.item())
                if fn.typeName() == "spBoidColliderSphere":
                    dagNode = OpenMaya.MFnDagNode(it.item())
                    path = OpenMaya.MDagPath()
                    dagNode .getPath(path)
                    path.pop()
                    transform = OpenMaya.MFnTransform(path)
                    c = transform.translation(OpenMaya.MSpace.kWorld)

                    doubleArray = OpenMaya.MScriptUtil()
                    doubleArray.createFromList( [0.0, 0.0, 0.0], 3 )
                    doubleArrayPtr = doubleArray.asDoublePtr()

                    sc = transform.getScale(doubleArrayPtr)
                    vec = OpenMaya.MVector(doubleArrayPtr )
```

```
                attr = fn.attribute("radius");
                plug = OpenMaya.MPlug(it.item(), attr)
                r = plug.asDouble()*vec.x

                a = pos
                b = OpenMaya.MVector(vel)
                b = a + b

                capsule = Capsule(a,b, bRadius)

                hit,t,d,overlap = self.SphereCapsuleIntersection(c, r, capsule)


                if hit:
                    projQ = d - a
                    P = c - a
                    perpQ = P - projQ

                    self._debug_projQ = projQ
                    self._debug_perpQ = perpQ
                    self._debug_A = a
                    self._debug_P = c - a
                    self.debug_curVel = vel

                    lateralForce = (perpQ.normal()*-1.0)*overlap
                    brakeForce   = vel*-1.0* overlap *brakingWeight
                    steeringForce = brakeForce+lateralForce
                    steeringForce = self._limit(steeringForce, maxForce)

            it.next()
    return steeringForce

def ClosestPtPointSegment(self, c, a, b):
    ab = b - a
    ca = c - a
    ab_dot_ab = ab*ab

    # only compute closest point to segment when particle is moving
    if ab_dot_ab > 0:

        t = (ca*ab)/ab_dot_ab

        unclamped_d = a + (ab*t)

        if t < 0.0:
            t = 0.0
        if t > 1.0:
           t = 1.0
        d = a + (ab*t)
        return t,d, unclamped_d
    else: # zero velocity
        return 0, a, a

def SphereCapsuleIntersection(self, sc, sr, capsule):

    ca = capsule.a
    cb = capsule.b
    cr = capsule.radius

    t,d, ud = self.ClosestPtPointSegment(sc,ca,cb)

    dist =  (sc - d).length()

    radius = sr + cr

    if (dist  <= radius):
```

```
            overlap = math.fabs(radius – dist)
            return True, t, ud, ov, overlap
        else:
            return False, -1.0, None, 0

    def _obstacleAvoidance(self, block, positions, velocities, outputForce):
        if positions.length() == 0:
            return
        outputForce.clear()
        for i in range(positions.length()):
            steeringForce = self._obstacleAvoidanceHelper(block, positions[i],
velocities[i], i)
            outputForce.append(steeringForce)

    def getTargets(self, block, targetType):

        hArrayValue = block.inputArrayValue(targetType)
        targets = OpenMaya.MVectorArray()

        numPoints = hArrayValue.elementCount();

        if numPoints > 0:
            hArrayValue.jumpToElement(0)
            for i in range(numPoints):
                elementHandle = hArrayValue.inputValue()
                targets.append(elementHandle.asVector())
                if i < (numPoints - 1):
                    hArrayValue.next()
        return targets

    def getWayPoints(self, block):

        hArrayValue = block.inputArrayValue(SteeringField.aPathFollowWayPoints)
        wayPoints = OpenMaya.MVectorArray()

        numPoints = hArrayValue.elementCount();

        if numPoints > 0:
            hArrayValue.jumpToElement(0)
            for i in range(numPoints):
                elementHandle = hArrayValue.inputValue()
                wayPoints.append(elementHandle.asVector())
                if i < (numPoints - 1):
                    hArrayValue.next()
        return wayPoints

    def getForceAtPoint(self, points, velocities, masses, forceArray, deltaTime):
        """
        This method is not required to be overridden, it is only necessary
        for compatibility with the MFnField function set.
        """
        block = forceCache()

    def iconSizeAndOrigin(self, width, height, xbo, ybo):
        OpenMaya.MScriptUtil.setUint( width, 32 )
        OpenMaya.MScriptUtil.setUint( height, 32 )
        OpenMaya.MScriptUtil.setUint( xbo, 4 )
        OpenMaya.MScriptUtil.setUint( ybo, 4 )
        return OpenMaya.MStatus.kSuccess

    def iconBitmap(self, bitmap):
        OpenMaya.MScriptUtil.setUcharArray( bitmap, 0, 0x7E)
        OpenMaya.MScriptUtil.setUcharArray( bitmap, 4, 0xFF)
        OpenMaya.MScriptUtil.setUcharArray( bitmap, 8, 0xDB)
        OpenMaya.MScriptUtil.setUcharArray( bitmap, 12, 0xDB)
        OpenMaya.MScriptUtil.setUcharArray( bitmap, 16, 0xDB)
        OpenMaya.MScriptUtil.setUcharArray( bitmap, 20, 0x7E)
```

```
        OpenMaya.MScriptUtil.setUcharArray( bitmap, 24, 0xFF)
        OpenMaya.MScriptUtil.setUcharArray( bitmap, 28, 0xFF)
        return OpenMaya.MStatus.kSuccess

    # methods to get attribute value of this node's attributes
    def _attrLongValue(self, block, attr):
        hValue = block.inputValue(attr)
        return hValue.asLong()

    def _attrDoubleValue(self, block, attr):
        hValue = block.inputValue(attr)
        return hValue.asDouble()

    def _attrBooleanValue(self, block, attr):
        hValue = block.inputValue(attr)
        return hValue.asBool()

def addAttribute(attr, name):
    try:
        SteeringField.addAttribute(attr)
    except:
        msg = "ERROR adding " + name + " attribute."
        statusError(msg)

# creator
def nodeCreator():
    return OpenMayaMPx.asMPxPtr( SteeringField() )

# initializer
def nodeInitializer():
    print("init SteeringField v1")
    numAttrFn = OpenMaya.MFnNumericAttribute()
    typAttrFn = OpenMaya.MFnTypedAttribute()
    msgAttrFn = OpenMaya.MFnMessageAttribute()
    msgAttrArrayFn = OpenMaya.MFnMessageAttribute()

    # create the steering force attributes.
    SteeringField.aGroundOffset = numAttrFn.create("groundOffset", "go",
OpenMaya.MFnNumericData.kDouble, 1.0)
    SteeringField.aSurfaceProjectionOffset = numAttrFn.create("surfaceProjectionOffset",
"spjoffset", OpenMaya.MFnNumericData.kDouble, 1.0)
    SteeringField.aGroundUpwardMult = numAttrFn.create("groundUpwardMult", "gum",
OpenMaya.MFnNumericData.kDouble, 0.5)
    SteeringField.aGroundDownwardMult = numAttrFn.create("groundDownwardMult", "gdm",
OpenMaya.MFnNumericData.kDouble, 1.0)
    SteeringField.aMaxSpeed = numAttrFn.create("maxSpeed", "ms",
OpenMaya.MFnNumericData.kDouble, 1.0)
    SteeringField.aMaxForce = numAttrFn.create("maxForce", "mf",
OpenMaya.MFnNumericData.kDouble, 0.5)
    SteeringField.aBoundRadius = numAttrFn.create("boundRadius", "br",
OpenMaya.MFnNumericData.kDouble, 1.0)
    SteeringField.aArriveProximityRadius = numAttrFn.create("arriveProximityRadius",
"apr", OpenMaya.MFnNumericData.kDouble, 1.0)
    SteeringField.aWayPointProximityRadius = numAttrFn.create("wayPointProximityRadius",
"wppr", OpenMaya.MFnNumericData.kDouble, 1.0)
    SteeringField.aDecelerationMult = numAttrFn.create("decelerationMult", "dm",
OpenMaya.MFnNumericData.kDouble, 5.0)
    SteeringField.aDecelerationDamping = numAttrFn.create("decelerationDamping", "dd",
OpenMaya.MFnNumericData.kDouble, 1.0)
    SteeringField.aPanicDistance = numAttrFn.create("panicDistance", "pds",
OpenMaya.MFnNumericData.kDouble, 100.0)
    SteeringField.aBrakeWeight = numAttrFn.create("brakeWeight", "bw",
OpenMaya.MFnNumericData.kDouble, 1.0)
    SteeringField.aSeed = numAttrFn.create("seed", "seed",
OpenMaya.MFnNumericData.kDouble, 2938)
    SteeringField.aFollowLeaderID = numAttrFn.create("followLeaderID", "flid",
OpenMaya.MFnNumericData.kLong, -1)
```

```
    SteeringField.aBoxWidth = numAttrFn.create("boxWidth", "boxWidth",
OpenMaya.MFnNumericData.kDouble, 20.0)
    SteeringField.aBoxHeight = numAttrFn.create("boxHeight", "boxHeight",
OpenMaya.MFnNumericData.kDouble, 20.0)
    SteeringField.aBoxDepth = numAttrFn.create("boxDepth", "boxDepth",
OpenMaya.MFnNumericData.kDouble, 20.0)

    # wander attributes
    SteeringField.aWanderRadius = numAttrFn.create("wanderRadius", "wanderRadius",
OpenMaya.MFnNumericData.kDouble, 4.0)
    SteeringField.aWanderDistance = numAttrFn.create("wanderDistance", "wanderDistance",
OpenMaya.MFnNumericData.kDouble, 6.0)
    SteeringField.aWanderJitter = numAttrFn.create("wanderJitter", "wanderJitter",
OpenMaya.MFnNumericData.kDouble, 0.25)

    SteeringField.aWanderScale  = numAttrFn.createPoint("wanderScale", "ws")
    numAttrFn.setDefault(1.0, 1.0, 1.0)

    SteeringField.aHeading  = numAttrFn.createPoint("heading", "h")
    numAttrFn.setDefault(1.0, 0.0, 0.0)

    SteeringField.aScaleForce  = numAttrFn.createPoint("scaleForce", "sf")
    numAttrFn.setDefault(1.0, 1.0, 1.0)

    SteeringField.aFollowLeaderOffset  = numAttrFn.createPoint("followLeaderOffset",
"flo")
    numAttrFn.setDefault(1.0, 0.0, 0.0)

    # for probing, used in ground surface constraint force
    SteeringField.aFeelerLength = numAttrFn.create("feelerLength", "fl",
OpenMaya.MFnNumericData.kDouble, 1.0)

    # Force Multipliers
    SteeringField.aMultSeparation = numAttrFn.create("multSeparation", "mlts",
OpenMaya.MFnNumericData.kDouble, 1.0)
    SteeringField.aMultCohesion = numAttrFn.create("multCohesion", "mltc",
OpenMaya.MFnNumericData.kDouble, 1.0)
    SteeringField.aMultAlign = numAttrFn.create("multAlign", "mlta",
OpenMaya.MFnNumericData.kDouble, 1.0)
    SteeringField.aMultSeek = numAttrFn.create("multSeek", "mltsk",
OpenMaya.MFnNumericData.kDouble, 1.0)
    SteeringField.aMultAvoid = numAttrFn.create("multAvoid", "mltav",
OpenMaya.MFnNumericData.kDouble, 1.0)
    SteeringField.aMultFlee = numAttrFn.create("multFlee", "mltfl",
OpenMaya.MFnNumericData.kDouble, 1.0)
    SteeringField.aMultHeading = numAttrFn.create("multHeading", "mlth",
OpenMaya.MFnNumericData.kDouble, 1.0)
    SteeringField.aMultWander = numAttrFn.create("multWander", "mltw",
OpenMaya.MFnNumericData.kDouble, 1.0)
    SteeringField.aMultArrive = numAttrFn.create("multArrive", "mltar",
OpenMaya.MFnNumericData.kDouble, 1.0)
    SteeringField.aMultFollowLeader = numAttrFn.create("multFollowLeader", "mpo",
OpenMaya.MFnNumericData.kDouble, 1.0)
    SteeringField.aMultGround = numAttrFn.create("multGround", "mltg",
OpenMaya.MFnNumericData.kDouble, 1.0)
    SteeringField.aMultBox = numAttrFn.create("multBox", "multBox",
OpenMaya.MFnNumericData.kDouble, 1.0)
    SteeringField.aMultPathFollow = numAttrFn.create("multPathFollow", "multpf",
OpenMaya.MFnNumericData.kDouble, 1.0)
    SteeringField.aZeroOverlap = numAttrFn.create("multZeroOverlap", "mltzo",
OpenMaya.MFnNumericData.kDouble, 1.0)
    SteeringField.aMultNeighborRepulse = numAttrFn.create("multNeighborRepulse", "mltnr",
OpenMaya.MFnNumericData.kDouble, 1.0)

    SteeringField.aShowBox = numAttrFn.create("showBox", "showBox",
OpenMaya.MFnNumericData.kBoolean)
    SteeringField.aObstacleAvoidanceOn = numAttrFn.create("avoidObstacleOn", "aoon",
```

```
OpenMaya.MFnNumericData.kBoolean)
    SteeringField.aObstacleAvoidanceDebugOn = numAttrFn.create("avoidObstacleDebugOn",
"aodon", OpenMaya.MFnNumericData.kBoolean)
    SteeringField.aSeekOn = numAttrFn.create("seekOn", "skon",
OpenMaya.MFnNumericData.kBoolean)
    SteeringField.aArriveOn = numAttrFn.create("arriveOn", "aron",
OpenMaya.MFnNumericData.kBoolean)
    SteeringField.aFleeOn = numAttrFn.create("fleeOn", "fon",
OpenMaya.MFnNumericData.kBoolean)
    SteeringField.aHeadingOn = numAttrFn.create("headingOn", "hon",
OpenMaya.MFnNumericData.kBoolean)
    SteeringField.aWanderOn = numAttrFn.create("wanderOn", "won",
OpenMaya.MFnNumericData.kBoolean)
    SteeringField.aSeparationOn = numAttrFn.create("separationOn", "son",
OpenMaya.MFnNumericData.kBoolean)
    SteeringField.aCohesionOn = numAttrFn.create("cohesionOn", "con",
OpenMaya.MFnNumericData.kBoolean)
    SteeringField.aAlignOn = numAttrFn.create("alignOn", "alon",
OpenMaya.MFnNumericData.kBoolean)
    SteeringField.aGroundOn = numAttrFn.create("groundOn", "gon",
OpenMaya.MFnNumericData.kBoolean)
    SteeringField.aSurfaceProjectionOn = numAttrFn.create("surfaceProjectionOn", "spjon",
OpenMaya.MFnNumericData.kBoolean)
    SteeringField.aProjectionWithVelocityOn =
numAttrFn.create("projectionWithVelocityOn", "fwvelon", OpenMaya.MFnNumericData.kBoolean)
    SteeringField.aFlipUpVectorOn = numAttrFn.create("flipUpVectorOn", "fupvon",
OpenMaya.MFnNumericData.kBoolean)
    SteeringField.aBoxOn = numAttrFn.create("boxOn", "boxOn",
OpenMaya.MFnNumericData.kBoolean)
    SteeringField.aPathFollowOn = numAttrFn.create("pathFollowOn", "pfon",
OpenMaya.MFnNumericData.kBoolean)
    SteeringField.aPathFollowLoopOn = numAttrFn.create("pathFollowLoopOn", "pflon",
OpenMaya.MFnNumericData.kBoolean)
    SteeringField.aFollowLeaderOn = numAttrFn.create("followLeaderOn", "opon",
OpenMaya.MFnNumericData.kBoolean)
    SteeringField.aFollowLeaderMaintainOffsetOn =
numAttrFn.create("followLeaderMaintainOffsetOn", "flmoon",
OpenMaya.MFnNumericData.kBoolean)
    SteeringField.aZeroOverlapOn = numAttrFn.create("zeroOverlapOn", "zoon",
OpenMaya.MFnNumericData.kBoolean)
    SteeringField.aNeighborRepulseOn = numAttrFn.create("neighborRepulseOn", "nron",
OpenMaya.MFnNumericData.kBoolean)
    SteeringField.aNeighborRepulseAgainstAllOn =
numAttrFn.create("neighborRepulseAgainstAllOn", "nraaon",
OpenMaya.MFnNumericData.kBoolean)
    SteeringField.aforecastBySpeedOn = numAttrFn.create("forecastBySpeedOn",
"forecastBySpeedOn", OpenMaya.MFnNumericData.kBoolean)
    SteeringField.aGroundforecastBySpeedOn = numAttrFn.create("groundForecastBySpeedOn",
"groundForecastBySpeedOn", OpenMaya.MFnNumericData.kBoolean)
    SteeringField.aWeightTruncatedSumPrioritizationOn =
numAttrFn.create("weightTruncatedSumPrioritizationOn", "wtspon",
OpenMaya.MFnNumericData.kBoolean)

    SteeringField.aSeparationNeighborDistance =
numAttrFn.create("separationNeighborDistance", "snd", OpenMaya.MFnNumericData.kDouble,
1.0)
    SteeringField.aCohesionNeighborDistance =
numAttrFn.create("cohesionNeighborDistance", "cnd", OpenMaya.MFnNumericData.kDouble, 1.0)
    SteeringField.aAlignNeighborDistance = numAttrFn.create("alignNeighborDistance",
"and", OpenMaya.MFnNumericData.kDouble, 1.0)
    SteeringField.aReplusionNeighborDistance =
numAttrFn.create("replusionNeighborDistance", "rndist", OpenMaya.MFnNumericData.kDouble,
1.0)

    SteeringField.aSeekTargets = numAttrFn.create("seekTargets", "seekTargets",
OpenMaya.MFnNumericData.k3Double)
    numAttrFn.setArray(True)
```

```
    SteeringField.aArriveTargets = numAttrFn.create("arriveTargets", "arriveTargets",
OpenMaya.MFnNumericData.k3Double)
    numAttrFn.setArray(True)

    SteeringField.aFleeTargets = numAttrFn.create("fleeTargets", "fleeTargets",
OpenMaya.MFnNumericData.k3Double)
    numAttrFn.setArray(True)

    SteeringField.aPathFollowWayPoints = numAttrFn.create("pathFollowWayPoints", "pfwp",
OpenMaya.MFnNumericData.k3Double)
    numAttrFn.setArray(True)

    # input for ground surface
    SteeringField.aGroundInputSurface = msgAttrFn.create("groundInputSurface", "gis")

    # for use on nurb surface collision avoidance steering force
    #SteeringField.aInputSurface = typAttrFn.create("inputSurface", "is",
OpenMaya.MFnData.kNurbsSurface)
    #typAttrFn.setArray(True);

    typAttrSpecialFnRot = OpenMaya.MFnTypedAttribute()
    defaultVectArray = OpenMaya.MVectorArray()
    vectArrayDataFn =  OpenMaya.MFnVectorArrayData()
    vecArray = vectArrayDataFn.create(defaultVectArray)
    SteeringField.aXYZRotation = typAttrSpecialFnRot.create("xyzRotation", "xyzRotation",
OpenMaya.MFnData.kVectorArray, vecArray)
    typAttrSpecialFnRot.setHidden(True)
    typAttrSpecialFnRot.setConnectable(False)
    typAttrSpecialFnRot.setInternal(True)

    typAttrSpecialFnPrj = OpenMaya.MFnTypedAttribute()
    defaultVectArray = OpenMaya.MVectorArray()
    vectArrayDataFn =  OpenMaya.MFnVectorArrayData()
    vecArray = vectArrayDataFn.create(defaultVectArray)
    SteeringField.aXYZProjection = typAttrSpecialFnPrj.create("xyzProjection",
"xyzProjection", OpenMaya.MFnData.kVectorArray, vecArray)
    typAttrSpecialFnPrj.setHidden(True)
    typAttrSpecialFnPrj.setConnectable(False)
    typAttrSpecialFnPrj.setInternal(True)

    typAttrSpecialFn = OpenMaya.MFnTypedAttribute()
    defaultVectArray = OpenMaya.MVectorArray()
    vectArrayDataFn =  OpenMaya.MFnVectorArrayData()
    vecArray = vectArrayDataFn.create(defaultVectArray)
    SteeringField.aUpDirection = typAttrSpecialFn.create("upDirection", "upDirection",
OpenMaya.MFnData.kVectorArray, vecArray)
    typAttrSpecialFn.setHidden(True)
    typAttrSpecialFn.setConnectable(False)
    typAttrSpecialFn.setInternal(True)

    # internal attributes to store wander sensor vectors
    typAttrSpecialFn0 = OpenMaya.MFnTypedAttribute()
    defaultVectArray = OpenMaya.MVectorArray()
    vectArrayDataFn =  OpenMaya.MFnVectorArrayData()
    vecArray = vectArrayDataFn.create(defaultVectArray)
    SteeringField.aWanderTargetPosition =
typAttrSpecialFn0.create("wanderTargetPosition", "wtp", OpenMaya.MFnData.kVectorArray,
vecArray)
    typAttrSpecialFn0.setHidden(True)
    typAttrSpecialFn0.setConnectable(False)
    typAttrSpecialFn0.setInternal(True)

    # internal attributes to store wander sensor vectors
    typAttrSpecialFn1 = OpenMaya.MFnTypedAttribute()
    defaultVectArray = OpenMaya.MVectorArray()
    vectArrayDataFn =  OpenMaya.MFnVectorArrayData()
```

```
    vecArray = vectArrayDataFn.create(defaultVectArray)
    SteeringField.aWanderSpherePosition =
typAttrSpecialFn1.create("wanderSpherePosition", "wsp", OpenMaya.MFnData.kVectorArray,
vecArray)
    typAttrSpecialFn1.setHidden(True)
    typAttrSpecialFn1.setConnectable(False)
    typAttrSpecialFn1.setInternal(True)

    # internal attributes to store wander sensor vectors
    typAttrSpecialFn2 = OpenMaya.MFnTypedAttribute()
    defaultVectArray = OpenMaya.MVectorArray()
    vectArrayDataFn =  OpenMaya.MFnVectorArrayData()
    vecArray = vectArrayDataFn.create(defaultVectArray)
    SteeringField.aWanderTarget = typAttrSpecialFn2.create("wanderTarget", "wt",
OpenMaya.MFnData.kVectorArray, vecArray)
    typAttrSpecialFn2.setHidden(True)
    typAttrSpecialFn2.setConnectable(False)
    typAttrSpecialFn2.setInternal(True)

    addAttribute(SteeringField.aShowBox, "showBox")
    addAttribute(SteeringField.aZeroOverlapOn, "zeroOverlapOn")
    addAttribute(SteeringField.aNeighborRepulseOn, "neighborRepulseOn")
    addAttribute(SteeringField.aNeighborRepulseAgainstAllOn,
"neighborRepulseAgainstAllOn")
    addAttribute(SteeringField.aforecastBySpeedOn, "forecastBySpeedOn")
    addAttribute(SteeringField.aObstacleAvoidanceOn, "avoidObstacleOn")
    addAttribute(SteeringField.aObstacleAvoidanceDebugOn, "avoidObstacleDebugOn")

    addAttribute(SteeringField.aSeekOn, "seekOn")
    addAttribute(SteeringField.aArriveOn, "arriveOn")
    addAttribute(SteeringField.aFleeOn, "fleeOn")
    addAttribute(SteeringField.aHeadingOn, "headingOn")
    addAttribute(SteeringField.aWanderOn, "wanderOn")
    addAttribute(SteeringField.aSeparationOn, "separationOn")
    addAttribute(SteeringField.aCohesionOn, "cohesionOn")
    addAttribute(SteeringField.aAlignOn, "alignOn")
    addAttribute(SteeringField.aFollowLeaderOn, "followLeaderOn")
    addAttribute(SteeringField.aBoxOn, "boxOn")
    addAttribute(SteeringField.aPathFollowOn, "pathFollowOn")
    addAttribute(SteeringField.aPathFollowLoopOn, "pathFollowLoopOn")
    addAttribute(SteeringField.aGroundOn, "groundOn")
    addAttribute(SteeringField.aSurfaceProjectionOn, "surfaceProjectionOn")
    addAttribute(SteeringField.aProjectionWithVelocityOn, "projectionWithVelocityOn")

    addAttribute(SteeringField.aFlipUpVectorOn, "flipUpVectorOn")
    addAttribute(SteeringField.aWeightTruncatedSumPrioritizationOn,
"weightTruncatedSumPrioritizationOn")

    addAttribute(SteeringField.aZeroOverlap, "multZeroOverlap")
    addAttribute(SteeringField.aMultSeparation, "multSeparation")
    addAttribute(SteeringField.aMultNeighborRepulse, "multNeighborRepulse")
    addAttribute(SteeringField.aMultCohesion, "multCohesion")
    addAttribute(SteeringField.aMultFollowLeader, "multFollowLeader")
    addAttribute(SteeringField.aMultAlign, "multAlign")
    addAttribute(SteeringField.aMultArrive, "multArrive")
    addAttribute(SteeringField.aMultSeek, "multSeek")
    addAttribute(SteeringField.aMultFlee, "multFlee")
    addAttribute(SteeringField.aMultAvoid, "multAvoid")
    addAttribute(SteeringField.aMultHeading, "multHeading")
    addAttribute(SteeringField.aMultWander, "multWander")
    addAttribute(SteeringField.aMultBox, "multBox")
    addAttribute(SteeringField.aMultPathFollow, "multPathFollow")
    addAttribute(SteeringField.aMultGround, "multGround")
    addAttribute(SteeringField.aSeparationNeighborDistance, "separationNeighborDistance")
    addAttribute(SteeringField.aCohesionNeighborDistance, "cohesionNeighborDistance")
    addAttribute(SteeringField.aAlignNeighborDistance, "alignNeighborDistance")
    addAttribute(SteeringField.aReplusionNeighborDistance, "replusionNeighborDistance")
```

```
    addAttribute(SteeringField.aScaleForce,"scaleForce")
    addAttribute(SteeringField.aFeelerLength, "feelerLength")
    addAttribute(SteeringField.aGroundOffset, "groundOffset")
    addAttribute(SteeringField.aGroundforecastBySpeedOn, "groundForecastBySpeedOn")
    addAttribute(SteeringField.aSurfaceProjectionOffset, "surfaceProjectionOffset")

    addAttribute(SteeringField.aGroundUpwardMult, "groundUpwardMult")
    addAttribute(SteeringField.aGroundDownwardMult, "groundDownwardMult")
    addAttribute(SteeringField.aMaxSpeed, "maxSpeed")
    addAttribute(SteeringField.aMaxForce, "maxForce")
    addAttribute(SteeringField.aBoundRadius, "boundRadius")
    addAttribute(SteeringField.aArriveProximityRadius, "arriveProximityRadius")
    addAttribute(SteeringField.aDecelerationMult, "decelerationMult")
    addAttribute(SteeringField.aDecelerationDamping, "decelerationDamping")
    addAttribute(SteeringField.aPanicDistance, "panicDistance")
    addAttribute(SteeringField.aBrakeWeight, "brakeWeight")
    addAttribute(SteeringField.aSeed, "seed")
    addAttribute(SteeringField.aFollowLeaderID, "followLeaderID")
    addAttribute(SteeringField.aWayPointProximityRadius, "wayPointProximityRadius")
    addAttribute(SteeringField.aBoxWidth, "boxWidth")
    addAttribute(SteeringField.aBoxHeight, "boxHeight")
    addAttribute(SteeringField.aBoxDepth, "boxDepth")

    # wander attributes
    addAttribute(SteeringField.aWanderScale, "wanderScale")
    addAttribute(SteeringField.aWanderRadius, "wanderRadius")
    addAttribute(SteeringField.aWanderDistance, "wanderDistance")
    addAttribute(SteeringField.aWanderJitter, "wanderJitter")

    addAttribute(SteeringField.aHeading,"heading")
    addAttribute(SteeringField.aFollowLeaderOffset, "followLeaderOffset")
    addAttribute(SteeringField.aFollowLeaderMaintainOffsetOn,
"followLeaderMaintainOffsetOn")

    addAttribute(SteeringField.aSeekTargets, "seekTargets")
    addAttribute(SteeringField.aArriveTargets, "arriveTargets")
    addAttribute(SteeringField.aFleeTargets, "fleeTargets")
    addAttribute(SteeringField.aPathFollowWayPoints, "pathFollowWayPoints")
    addAttribute(SteeringField.aGroundInputSurface, "groundInputSurface")

    # internal private attributes, hidden from user, used by wander steering force
    addAttribute(SteeringField.aWanderTargetPosition, "wanderTargetPosition")
    addAttribute(SteeringField.aWanderSpherePosition, "wanderSpherePosition")
    addAttribute(SteeringField.aWanderTarget, "wanderTarget")
    addAttribute(SteeringField.aUpDirection, "upDirection")
    addAttribute(SteeringField.aXYZRotation, "xyzRotation")
    addAttribute(SteeringField.aXYZProjection,"xyzProjection")

# initialize the script plug-in
def initializePlugin(mobject):
    mplugin = OpenMayaMPx.MFnPlugin(mobject, "DPA@Clemson", "1.0", "Any")
    try:
        mplugin.registerNode(kPluginName, kPluginNodeId, nodeCreator, nodeInitializer,
OpenMayaMPx.MPxNode.kFieldNode)
    except:
        statusError("Failed to register node: %s" % kPluginName)

# uninitialize the script plug-in
def uninitializePlugin(mobject):
    mplugin = OpenMayaMPx.MFnPlugin(mobject)
    try:
        mplugin.deregisterNode(kPluginNodeId)
    except:
        statusError("Failed to deregister node: %s" % kPluginName)
```

Appendix F

AEspBoidColliderSphereTemplate.mel

```
// File: AEspBoidColliderSphereTemplate.mel
//
// MEL Script for Attribute Editor template for spBoidColliderSphere
//
// Author: Edgar Rodriguez

global proc AEspBoidColliderSphereTemplate(string $nodeName)
{
   editorTemplate -beginScrollLayout;
      editorTemplate -beginLayout "Collider Attributes" -collapse false;
         editorTemplate -beginNoOptimize;
         editorTemplate -addSeparator;
         editorTemplate -addControl "radius";
         editorTemplate -endNoOptimize;
      editorTemplate -endLayout;

      editorTemplate -suppress "LocalPosition";
      editorTemplate -suppress "LocalScale";

   editorTemplate -addExtraControls;
   editorTemplate -endScrollLayout;
}
```

Appendix G

AEspBoidGuidesTemplate.mel

```
// File: AEspBoidGuidesTemplate.mel
//
// MEL Script for Attribute Editor template for spBoidGuides
//
// Author: Edgar Rodriguez

global proc AEspBoidGuidesTemplate(string $nodeName)
{
   editorTemplate -beginScrollLayout;
      editorTemplate -beginLayout "Visual Guide Controls" -collapse false;
         editorTemplate -beginNoOptimize;
         editorTemplate -addSeparator;
         editorTemplate -l "Show Guides on All Boids" -addControl "allParticlesOn";
         editorTemplate -l "Show Guides by Boid ID" -addControl "particleID";
         editorTemplate -addSeparator;
         editorTemplate -l "Show Avoid Sensor Guides" -addControl "showAvoidSensor";
         editorTemplate -l "Show Wander Sensor Guides" -addControl "showWanderSensor";
         editorTemplate -l "Show Feeler Sensor Guides"  -addControl "showFeelerSensor";
         editorTemplate -l "Show Collision Boundary Guides"  -addControl "showBoundary";
         editorTemplate -l "Show Rotation XYZ Axis Guides"  -addControl "displayAxis";
         editorTemplate -addSeparator;
         editorTemplate -addControl "steeringFieldInput";
         editorTemplate -endNoOptimize;
      editorTemplate -endLayout;

      editorTemplate -suppress "LocalPosition";
      editorTemplate -suppress "LocalScale";

   editorTemplate -addExtraControls;
   editorTemplate -endScrollLayout;
}
```

Appendix H

AEspSteeringFieldTemplate.mel

```
// File: AEspSteeringFieldTemplate.mel
//
// MEL Script for Attribute Editor template for spSteeringField
//
// Author: Edgar Rodriguez

global proc AEspSteeringFieldTemplate(string $nodeName)
{
   editorTemplate -beginScrollLayout;
      editorTemplate -beginLayout "Basic Parameters" -collapse true;
         editorTemplate -beginNoOptimize;
         editorTemplate -addSeparator;
         editorTemplate -l "Prioritization On" -addControl
"weightTruncatedSumPrioritizationOn";
         editorTemplate -addControl "maxForce";
         editorTemplate -addControl "maxSpeed";
         editorTemplate -addControl "boundRadius";
         editorTemplate -addControl "seed";
         editorTemplate -addSeparator;
         editorTemplate -l "Leader ID" -addControl "followLeaderID";
         editorTemplate -addSeparator;
         editorTemplate -addControl "scaleForce";
         editorTemplate -addSeparator;
         editorTemplate -l "Flip Up Vector" -addControl "flipUpVectorOn";
         editorTemplate -addSeparator;
         editorTemplate -l "Enable Zero Overlap" -addControl "zeroOverlapOn";
         editorTemplate -l "Zero Overlap Damping" -addControl "multZeroOverlap";
         editorTemplate -addSeparator;
         editorTemplate -endNoOptimize;
      editorTemplate -endLayout;

      editorTemplate -beginLayout "Simulation Box Parameters" -collapse true;
         editorTemplate -beginNoOptimize;
         editorTemplate -addSeparator;
         editorTemplate -l "Enable" -addControl "boxOn";
         editorTemplate -l "Show Guides" -addControl "showBox";
         editorTemplate -l "magnitude" -addControl "multBox";
         editorTemplate -l "Width" -addControl "boxWidth";
         editorTemplate -l "Height" -addControl "boxHeight";
         editorTemplate -l "Depth" -addControl "boxDepth";
         editorTemplate -endNoOptimize;
      editorTemplate -endLayout;

      editorTemplate -beginLayout "Self Replusion Parameters" -collapse true;
         editorTemplate -beginNoOptimize;
         editorTemplate -addSeparator;
         editorTemplate -l "Enable" -addControl "neighborRepulseOn";
         editorTemplate -l "magnitude" -addControl "multNeighborRepulse";
         editorTemplate -addControl "forecastBySpeedOn";
         editorTemplate -l "Neighbor Distance" -addControl "replusionNeighborDistance";
         editorTemplate -addSeparator;
         editorTemplate -l "Affect All Attached Particles" -addControl
"neighborRepulseAgainstAllOn";
         editorTemplate -endNoOptimize;
      editorTemplate -endLayout;

      editorTemplate -beginLayout "Seek Parameters" -collapse true;
         editorTemplate -beginNoOptimize;
         editorTemplate -addSeparator;
         editorTemplate -l "Enable" -addControl "seekOn";
```

```
            editorTemplate -l "magnitude" -addControl "multSeek";
            editorTemplate -addControl "seekTargets";
            editorTemplate -endNoOptimize;
        editorTemplate -endLayout;

        editorTemplate -beginLayout "Arrive Parameters" -collapse true;
            editorTemplate -beginNoOptimize;
            editorTemplate -addSeparator;
            editorTemplate -l "Enable" -addControl "arriveOn";
            editorTemplate -l "magnitude" -addControl "multArrive";
            editorTemplate -addControl "decelerationDamping";
            editorTemplate -l "Deceleration Multiplier" -addControl "decelerationMult";
            editorTemplate -addControl "arriveTargets";
            editorTemplate -endNoOptimize;
        editorTemplate -endLayout;

        editorTemplate -beginLayout "Flee Parameters" -collapse true;
            editorTemplate -beginNoOptimize;
            editorTemplate -addSeparator;
            editorTemplate -l "Enable" -addControl "fleeOn";
            editorTemplate -l "magnitude" -addControl "multFlee";
            editorTemplate -addControl "panicDistance";
            editorTemplate -addControl "fleeTargets";
            editorTemplate -endNoOptimize;
        editorTemplate -endLayout;

        editorTemplate -beginLayout "Wander Parameters" -collapse true;
            editorTemplate -beginNoOptimize;
            editorTemplate -addSeparator;
            editorTemplate -l "Enable" -addControl "wanderOn";
            editorTemplate -l "magnitude" -addControl "multWander";
            editorTemplate -addControl "wanderRadius";
            editorTemplate -addControl "wanderDistance";
            editorTemplate -addControl "wanderJitter";
            editorTemplate -addControl "wanderScale";
            editorTemplate -endNoOptimize;
        editorTemplate -endLayout;

        editorTemplate -beginLayout "Heading Parameters" -collapse true;
            editorTemplate -beginNoOptimize;
            editorTemplate -addSeparator;
            editorTemplate -l "Enable" -addControl "headingOn";
            editorTemplate -l "magnitude" -addControl "multHeading";
            editorTemplate -l "Direction" -addControl "heading";
            editorTemplate -endNoOptimize;
        editorTemplate -endLayout;

        editorTemplate -beginLayout "Surface Follow Parameters" -collapse true;
            editorTemplate -beginNoOptimize;
            editorTemplate -addSeparator;
            editorTemplate -l "Enable" -addControl "groundOn";
            editorTemplate -addControl "groundOffset";
            editorTemplate -l "forcast by speed" -addControl "groundForecastBySpeedOn";
            editorTemplate -l "forcast by feeler length" -addControl "feelerLength";
            editorTemplate -l "magnitude" -addControl "multGround";
            editorTemplate -l "Uphill Multiplier" -addControl "groundUpwardMult";
            editorTemplate -l "Downhill Multiplier" -addControl "groundDownwardMult";
            editorTemplate -l "Surface Shape Input" -addControl "groundInputSurface";
            editorTemplate -addSeparator;
            editorTemplate -addControl "surfaceProjectionOn";
            editorTemplate -l "Scale Projection with Velocity" -addControl
"projectionWithVelocityOn";
            editorTemplate -addControl "surfaceProjectionOffset";
            editorTemplate -endNoOptimize;
        editorTemplate -endLayout;

        editorTemplate -beginLayout "Separation Parameter" -collapse true;
```

```
            editorTemplate -beginNoOptimize;
            editorTemplate -addSeparator;
            editorTemplate -l "Enable" -addControl "separationOn";
            editorTemplate -l "magnitude" -addControl "multSeparation";
            editorTemplate -l "Neighbor Distance" -addControl "separationNeighborDistance";
            editorTemplate -endNoOptimize;
        editorTemplate -endLayout;

        editorTemplate -beginLayout "Cohesion Parameter" -collapse true;
            editorTemplate -beginNoOptimize;
            editorTemplate -addSeparator;
            editorTemplate -l "Enable" -addControl "cohesionOn";
            editorTemplate -l "magnitude" -addControl "multCohesion";
            editorTemplate -l "Neighbor Distance" -addControl "cohesionNeighborDistance";
            editorTemplate -endNoOptimize;
        editorTemplate -endLayout;

        editorTemplate -beginLayout "Align Parameter" -collapse true;
            editorTemplate -beginNoOptimize;
            editorTemplate -addSeparator;
            editorTemplate -l "Enable" -addControl "alignOn";
            editorTemplate -l "magnitude" -addControl "multAlign";
            editorTemplate -l "Neighbor Distance" -addControl "alignNeighborDistance";
            editorTemplate -endNoOptimize;
        editorTemplate -endLayout;

        editorTemplate -beginLayout "Follow Leader Parameters" -collapse true;
            editorTemplate -beginNoOptimize;
            editorTemplate -addSeparator;
            editorTemplate -l "Enable" -addControl "followLeaderOn";
            editorTemplate -l "magnitude" -addControl "multFollowLeader";
            editorTemplate -addSeparator;
            editorTemplate -l "Maintain Offset" -addControl "followLeaderMaintainOffsetOn";
            editorTemplate -l "Offset" -addControl "followLeaderOffset";
            editorTemplate -l "Arrive Proximity Radius" -addControl "arriveProximityRadius";
            editorTemplate -endNoOptimize;
        editorTemplate -endLayout;

        editorTemplate -beginLayout "Obstacle Avoidance Parameters" -collapse true;
            editorTemplate -beginNoOptimize;
            editorTemplate -addSeparator;
            editorTemplate -l "Enable" -addControl "avoidObstacleOn";
            editorTemplate -l "magnitude" -addControl "multAvoid";
            editorTemplate -l "Object Proximity Brake Weight" -addControl "brakeWeight";
            editorTemplate -l "DebugOn" -addControl "avoidObstacleDebugOn";
            editorTemplate -endNoOptimize;
        editorTemplate -endLayout;

        editorTemplate -beginLayout "Path Follow Parameters" -collapse true;
            editorTemplate -beginNoOptimize;
            editorTemplate -addSeparator;
            editorTemplate -l "Enable" -addControl "pathFollowOn";
            editorTemplate -l "magnitude" -addControl "multPathFollow";
            editorTemplate -l "Waypoint Proximity Radius" -addControl
    "wayPointProximityRadius";
            editorTemplate -l "Loop" -addControl "pathFollowLoopOn";
            editorTemplate -l "Waypoints" -addControl "pathFollowWayPoints";
            editorTemplate -endNoOptimize;
        editorTemplate -endLayout;

        editorTemplate -suppress "magnitude";
        editorTemplate -suppress "attenuation";
        editorTemplate -suppress "maxDistance";
        editorTemplate -suppress "applyPerVertex";
        editorTemplate -suppress "useMaxDistance";
        editorTemplate -suppress "volumeShape";
        editorTemplate -suppress "volumeExclusion";
```

```
        editorTemplate -suppress "volumeOffset";
        editorTemplate -suppress "volumeSweep";
        editorTemplate -suppress "falloffCurve";
        editorTemplate -suppress "renderLayerInfo";
        editorTemplate -suppress "rotateQuaternion";
        editorTemplate -suppress "mentalRayControls";
        editorTemplate -suppress "axialMagnitude";
        editorTemplate -suppress "curveRadius";
        editorTemplate -suppress "sectionRadius";
        editorTemplate -suppress "inputCurve";
        editorTemplate -suppress "trapRadius";
        editorTemplate -suppress "trapEnds";
        editorTemplate -suppress "trapInside";
        editorTemplate -suppress "inheritsTransform";
        editorTemplate -suppress "translate";
        editorTemplate -suppress "rotate";
        editorTemplate -suppress "rotateOrder";
        editorTemplate -suppress "scale";
        editorTemplate -suppress "shear";
        editorTemplate -suppress "rotateAxis";
        AEfieldInclude $nodeName;
    editorTemplate -addExtraControls;
    editorTemplate -endScrollLayout;
}
```

Appendix I

Partial Implementation of SSB Node (ssbNode.py)

```
import math, sys
import maya.OpenMaya as OpenMaya
import maya.OpenMayaMPx as OpenMayaMPx

kPluginNodeTypeName = "ssbNode"
kPluginNodeId = OpenMaya.MTypeId(0x87205)

# Node definition
class ssbNode(OpenMayaMPx.MPxNode):
        # class variables
        aCodeAttr   = OpenMaya.MObject()
        aOutputAttr = OpenMaya.MObject()

        def __init__(self):
                OpenMayaMPx.MPxNode.__init__(self)


        def compute(self, plug, data):
                if (plug == ssbNode.aOutputAttr):
                        """
                        Build UI from tag keywords in code block
                        """
                        print "updating UI"
                        data.setClean(plug)
                else:
                        return OpenMaya.MStatus.kUnknownParameter
                return OpenMaya.MStatus.kSuccess

# creator
def nodeCreator():
        return OpenMayaMPx.asMPxPtr( ssbNode() )

# initializer
def nodeInitializer():
        tAttr = OpenMaya.MFnTypedAttribute()
        fnStringData = OpenMaya.MFnStringData()
        defaultString = fnStringData.create( "Insert Code" )

        ssbNode.aCodeAttr = tAttr.create( "computeSteeringForce", "csf",
OpenMaya.MFnData.kString, defaultString )
        tAttr.setStorable(True)
        tAttr.setKeyable(False)
        ssbNode.addAttribute(ssbNode.aCodeAttr)

        ssbNode.aOutputAttr = tAttr.create( "codeOutput", "coutput",
OpenMaya.MFnData.kString, defaultString )
        tAttr.setStorable(True)
        tAttr.setKeyable(False)
        tAttr.setHidden(True)
        ssbNode.addAttribute(ssbNode.aOutputAttr)
        ssbNode.attributeAffects(ssbNode.aCodeAttr, ssbNode.aOutputAttr)

# initialize the script plug-in
def initializePlugin(mobject):
        mplugin = OpenMayaMPx.MFnPlugin(mobject, "Autodesk", "1.0", "Any")
        try:
                mplugin.registerNode( kPluginNodeTypeName, kPluginNodeId, nodeCreator,
nodeInitializer )
        except:
                sys.stderr.write( "Failed to register node: %s" % kPluginNodeTypeName )
```

```
                raise

# uninitialize the script plug-in
def uninitializePlugin(mobject):
        mplugin = OpenMayaMPx.MFnPlugin(mobject)
        try:
                mplugin.deregisterNode( kPluginNodeId )
        except:
                sys.stderr.write( "Failed to deregister node: %s" % kPluginNodeTypeName )
                raise
```

Appendix J

Plugin Installation

This plugin was developed and tested in Maya 2009

1. open the steeringfield.zip archive
2. extract the files into your maya user path
3. Edit your Maya.env file and add the following:

The Maya.env is located in the following directory:

'x.x ' corresponds to maya version

(Windows) \Documents and Settings\username\My Documents\maya\x.x
(Linux) ~username/maya/x.x

If your environment is Windows XP add the following lines to your Maya.env
MAYA_PLUG_IN_PATH = %MAYA_APP_DIR%\steeringfield_plugin\plugin
MAYA_SCRIPT_PATH = %MAYA_APP_DIR%\steeringfield_plugin\mel

If your environment is Linux add the following lines to your Maya.env
MAYA_PLUG_IN_PATH = $MAYA_APP_DIR/steeringfield_plugin/plugin
MAYA_SCRIPT_PATH = $MAYA_APP_DIR/steeringfield_plugin/mel

Note that MAYA_APP_DIR defaults to:

(Windows) \Documents and Settings\username\My Documents\maya
or
(Linux) ~username/maya

When you load maya you will need to load the following plugins:
boidCollider.py
boidGuides.py
SteeringField.py

Using Maya's Python Script editor run the following commands to load the plugins.

```
import maya.cmds as cmds

#load collision sphere plugin
if not cmds.pluginInfo('boidCollider.py', q=True, loaded=True):
    cmds.loadPlugin( 'boidCollider.py' )
```

```
#load guides plugin
if not cmds.pluginInfo('boidGuides.py', q=True, loaded=True):
   cmds.loadPlugin( 'boidGuides.py' )

#load steering field plugin
if not cmds.pluginInfo('SteeringField.py', q=True, loaded=True):
   cmds.loadPlugin( 'SteeringField.py' )
```

Geometry Instancing and Rotations

This is only needed if you are instancing geometry on the particles.
By design particles do not have rotations, since particles only
represent points in space. The steeringfield internally
computes rotation using the the particles velocity and scene
up vector or surface normals. In order for rotations to be recognized by the geomertyr instancer you will need to add a perParticle Vector Array named “rotationPP” to the particle shape node.

Example files:

butterfly_flock.mb demonstrates flocking behavior
fish_prey.mb demonstrates prey and predator behavior
surfaceFollow.mb demonstrates leader following and surface following

Bibliography

[Buck05] Buckland, Mat. "How to Create Autonomously Moving Game Agents." Programming game AI by example. Plano, Texas: Wordware Pub., 2005.

[Duve04] Duvekot, Michiel. Learning Maya 6: Dynamics: A hands-on introduction to the key tools and techniques of Maya Dynamics. Toronto: Alias, 2004.

[Eric05] Ericson, Christer. Real-time collision detection: Christer Ericson. Amsterdam: Elsevier, 2005.

[Goud03] Gould, David A. D. Complete Maya programming: an extensive guide to MEL and C++ API. Vol. 1. San Francisco: Morgan Kaufmann Pub., 2003.

[Goud05] Gould, David A. D. Complete Maya programming. an in-depth guide to 3D fundamentals, geometry, and modeling. Vol. 2. San Francisco, CA: Morgan Kaufmann, 2005.

[Kolv09] Kolve, Carsten. "Brainbugz v1.0 Behavioural Animation Engine for Maya Particles." 17 March 2009 <http://www.kolve.com/mp_brainbugz/docs/brainbugz.htm>.

[Leng02] Lengyel, Eric. Mathematics for 3D game programming and computer graphics. Hingham, Massachusetts: Charles River Media, 2002.

[MasA10] "Massive - Film, TV & Games." Massive Software - Simulating Life. 1 June 2010 <http://www.massivesoftware.com/film-television-games/>.

[MasB10] "Massive - What Is Massive?" Massive Software - Simulating Life. 1 June 2010 <http://www.massivesoftware.com/whatismassive>.

[Mill07] Millington, Ian. Game physics engine development. Amsterdam: Morgan Kaufmann, 2007.

[Pare02] Parent, Rick. "Controlling Groups of Objects." Computer animation algorithms and techniques. San Francisco: Morgan Kaufmann, 2002.

[Reyn01] Reynolds, Craig W. "Boids (Flocks, Herds, and Schools: a Distributed Behavioral Model)." Reynolds Engineering & Design. 2001. 5 April 2010 <http://www.red3d.com/cwr/boids/>.

[Reyn87] Reynolds, Craig W. "Flocks, Herds, and Schools: A Distributed Behavioral Model." Computer Graphics. Proc. of SIGGRAPH '87, Anaheim, California. Vol. 21. Ser. 4. 25-34.

[Reyn88] Reynolds, Craig W. "Not Bumping Into Things." SIGGRAPH 88 (1988): G1-G13. Notes on "obstacle avoidance" for the course on Physically Based Modeling at SIGGRAPH 88. ACM SIGGRAPH. 3 July 2010 <http://www.red3d.com/cwr/nobump/nobump.html>.

[Reyn07] Reynolds, Craig W. "OpenSteer Preliminary Documentation." OpenSteer. Research and Development Sony Computer Entertainment America. 2007. 15 June 2010 <http://opensteer.sourceforge.net/doc.html>.

[Reyn99] Reynolds, Craig W. "Steering Behaviors For Autonomous Characters." In the proceedings of Game Developers Conference 1999. San Jose, California. San Francisco: Miller Freeman Game Group, 1999. 763-82.

[Reyn04] Reynolds, Craig W. "Steering Behaviors for Autonomous Characters." OpenSteer. Research and Development Sony Computer Entertainment America. 2004. 15 June 2010 <http://opensteer.sourceforge.net/index.html>.

[Rina06] Rinaldi, Adriano. "Crowd Maker / Pedestrian behavioural animation engine for Maya particles – v0.7." Tiscali Webspace. 2006. 10 July 2010 <http://web.tiscali.it/maya_tutorial/>.

[Schn10] Schnellhammer, Christian. The Steering Behaviors Webpage. 10 June 2010 <http://www.steeringbehaviors.de/>.

[Toml10] Tomlinson, Simon L. "THE LONG AND SHORT OF STEERING IN COMPUTER GAMES." Int. Journal of SIMULATION 5: 33-44. 3 June 2010 <http://ducati.doc.ntu.ac.uk/uksim/journal/Vol-5/No-1&2/TOMLINSON.pdf>.

[Pyth05] Writing C extensions for Python. 2005. 13 June 2010 <http://wiki.cacr.caltech.edu/danse/index.php/Writing_C_extensions_for_Python>.