David Dunnings  
13015063  
Simon Scarle

Simulated Worlds 2



Contents

[Introduction 2](#_Toc416732316)

[Research 3](#_Toc416732317)

[Predators and Prey 3](#_Toc416732318)

[Boid Simulations 4](#_Toc416732319)

[Development 5](#_Toc416732320)

[Class Iterations 6](#_Toc416732321)

[Debug Iterations 6](#_Toc416732322)

[Boid Iterations 9](#_Toc416732323)

[Class Diagram 10](#_Toc416732324)

[Gameobject 10](#_Toc416732325)

[BoidManager 11](#_Toc416732326)

[Camera 12](#_Toc416732327)

[PredCamera 13](#_Toc416732328)

[Light 14](#_Toc416732329)

[VBGO 15](#_Toc416732330)

[Boid 16](#_Toc416732331)

[Waypoint 18](#_Toc416732332)

[Critical Evaluation 19](#_Toc416732333)

[Suitability as a Game Asset 20](#_Toc416732334)

[Data File Layout 21](#_Toc416732335)

[References 22](#_Toc416732336)

# Introduction

Across the globe, every day, millions of prey are hunted by predators. The balance of nature is such that animals that require protein-heavy diets will seek out weaker animals for sustenance. The behavior of such food networks has been heavily documented by biologists and researchers alike. The transfer of food (energy) through consumption can be mapped using food webs. Predators or *secondary consumers* will eat *primary consumers* who eat *producers*. *Producers* gain their energy from the light, through means of a chemical reaction such as photosynthesis. Below is an example food web from the stonefly (*secondary consumer*) to leaf fragments (*producer*) [Fig. 1].

|  |
| --- |
| Food web |
| **Fig. 1 – A food web for the for flies and producers [REF 1]** |

An interesting area of research is the food web of the African plains. The reason for this is that there are many prey that co-exist as well as predators which will hunt multiple prey. The *primary consumers* must migrate in order to find constantly-shifting nutrient rich areas to graze [REF 2]. This provides an interesting set of behaviors to model in a computer simulation.

The objective of this research is to implement and develop a predator-prey simulation generator which can be customizable to produce a variety of ecosystems. The simulation will feature a graphical user interface that will enable the user to modify the parameters during runtime. Animals will spawn at pre-determined spawn points and attempt to traverse the screen to reach a pre-determined finish. There will also be outposts where animals can be set to patrol. This allows the user to simulate such situations as a migratory herd passing through predator-dense lands.

The final simulation must be suitable to be used as an asset within a game. To succeed in this, the simulation will have handles that extend outside the code. Anyone who wishes to implement my solution into their game will only need to add the project into their code and initialize a new “PredatorPreySimulation” class. All of the parameters for the simulation will be accessable externally and the user will not need to edit any code to achieve the simulation that they require. I will return to this requirement at the end of this report to reflect on whether or not I have succeeded in creating an asset as opposed to a standalone program.

# Research

The first section of my research will concentrate on predator and prey interactions. More specifically, how predators hunt and how prey flee. The second part of my research will focus on previous simulations which are related to my project.

Predators and Prey

A predominant feature that defines the effectiveness of predators is their vision. Their eyes are generally located in a forward position and allow them to see large distances whilst having a narrow field of vision. This means that the predators can keep track of a prey from far away and they have little or no care for things in their peripheral vision, such as other predators. From this, I have deducted that the predators in my simulation will have a parameter to represent vision distance which a prey needs to be located within for the predator to begin hunting them. This range will increase for predators further up the food chain. This will enable higher predators to be more successful at hunting.

An area of interest which has a high number of studies is predator-prey balance within an ecosystem. The results from various studies point to the idea that as prey numbers increase, predator numbers increase and as predator numbers increase, prey numbers decrease. This keeps the ecosystem in constant flux. In the Princeton guide to ecology [REF 3] Denno and Lewis outline this “Reciprocal density effect”.

|  |
| --- |
| http://385888a2d8aaf2499c5f-64d142b35093ce125caba6c3b31274a9.r81.cf1.rackcdn.com/prge2012/fig_pt11_7_p205_fig002.jpg |
| **Fig. 2 – (A) Oscillating predator and prey populations and (B) a neutrally stable predator-prey limit cycle generated by the Lotka-Volterra equations** |

Boid Simulations

In my research I discovered that the founder of boid simulations was Craig W. Reynolds. Craig published a paper in 1987 titled “**Flocks, Herds, and Schools: A Distributed Behavioral Model**” [REF 4]. This was the first paper to talk about using computer graphics to simulate real-world flocking behavior. One specific section of his paper, which I think will heavily influence my boid system is the section labeled “**Simulated Flocks**”. In these few paragraphs Craig outlines the three behaviors which make up the flocking system:

*Collision Avoidance: avoid collisions with nearby flockmates*

*Velocity Matching: attempt to match velocity with nearby flockmates*

*Flock Centering: attempt to stay close to nearby flockmates*

|  |
| --- |
|  |
| **Fig. 3 – Grouping and fleeing** |

For each of these three behaviors I designed a section of code to replicate the behavior. I concluded that to simulate collision avoidance, each boid will have a force applied to it for each nearby boid. The force will be created by calculating the vector from the nearby boid to the original boid. I can then apply a scaled down version of this vector to the original boid to push it away from the nearby boid.

As for the velocity matching I will most likely take an average velocity of every boid in a localized area around each individual boid. I will then apply a scaled-down version of this to each boid.

I will perform a similar calculation for the flock centering. I will take an average location of all nearby boids and create a vector from the target boid to the average group location. I can then apply a scaled-down version of this vector to the target boid.

In Craig’s paper he also mentions collision avoidance. He listed a couple of methods of making sure boids avoid obstacles. Steer-to-avoid is one method that Craig outlines:

*“The boid considers only obstacles directly in front of it. Working in local perspective space, it finds the silhouette edge of the obstacle closest to the point of eventual impact. A radial vector is computed which will aim the boid at a point one body length beyond that silhouette edge.”*

This was the chosen method for Craig’s system and I am considering using it in my simulation. However, Craig also mentions another method of collision avoidance which I believe may work better in my system:

*“The force field model postulates a field of repulsion force emanating from the obstacle out into space; the boids are increasingly repulsed as they get closer to the obstacle.”*

The force-field model is most likely going to be the model of choice for my collision avoidance. The reason for this is that it uses the same principle of attractive and repulsive forces as the rest of the simulation. Therefore I can easily have the same functions for avoiding obstacles as avoiding predators.

Development

During development the simulation has been through multiple iterations. The general direction of development has been positive although there were points when newer versions had inferior methods to solving problems and I decided to revert to a previous version at a later time. I have documented the major iterations for each of the main sections of the simulation.

Class Iterations

The first iteration of my boid simulation had a ***predatorBoid*** class and a ***preyBoid*** class. I quickly realized that this was unnecessary as both classes were almost identical, except for the polarity of the forces applied being reversed. Therefore I merged both classes into a parent class ***Boid***. This class had a ***BoidType*** enumerator class which dictates the hierarchy of the food chain. For example, one simple setup of the ***BoidType*** enumerator could be the following:

[0] boid\_OBSTACLE, [1] boid\_WILDEBEEST, [2] boid\_LION

In this instance, any boid with their BoidType set to [1] or [2] will avoid any boid with their BoidType set to [0]. Furthermore, boids with BoidType [2] will be attracted to boids of BoidType [1]. This attraction is directly inversed for the relationship between BoidType [1] and BoidType [2]. This ultimately results in BoidType [2] being the sole predator type, which hunts BoidType [1].

After deliberating on this method of differentiating boids, I finally decided that there was a better way to do it. I created a new struct called **Type**. This struct holds each of the variables that **BoidType** previously represented. The simulation can then assign this **Type** to each boid as they are spawned. Another beneficial feature of adding a **Type** struct is that the simulation can load them in from an external file when the parameters are loaded. This allows the user to create custom **Types** to fit the simulation that they need.

Debug Iterations

The interface for the simulation has been through multiple iterations during development. The purpose of the interface is to allow the user to observe preset parameters and live variables during runtime. An additional feature is the ability to update parameters during runtime using keyboard and/or mouse input.

|  |
| --- |
|  |
| **Fig. 4 – First interface version** |

The first version of my simulation to include a debug interface listed each simulation parameter along the left of the screen. There were plus and minus buttons next to each parameter which incremented / decremented their associated value. These were achieved by using a large nested if statement. The program would check the cursor’s position upon clicking and check if it was within the bounds for any of the “buttons”.

|  |
| --- |
|  |
| **Fig. 5 – Second interface iteration** |

The next version of the GUI removed the clickable “buttons” and instead opted for a more keyboard-centralized interface. The debug menu started with a couple of true/false statements to display whether the simulation parameters have been loaded from file and if the mouse is being used as an obstacle. The next section listed all of the available controls for the simulation with a short description of what the action does. The next section listed all of the simulation parameters. This section became no longer interact-able via mouse. The user could instead press a key to open the data file which holds the simulation parameters. The user could edit these parameters and close the text file to have the changes take effect instantly within the simulation.

Another noticeable change in this rendition is the addition of force indicators for each boid. These use the boid’s current speed and current direction to create a line out from the boid. To create these the boids have a secondary vertex buffer which uses a line list format to draw a line between two vertices. Using the same line list vertex buffer the simulation also draws circles around each waypoint to indicate it’s area of effect.

After reviewing both of these versions of interface, I concluded that a combination of clickable interface and greater visual representation of forces e.t.c. was required. Therefore I merged both versions into a new interface.

|  |
| --- |
|  |
| **Fig. 6 – Final interface version** |

|  |
| --- |
|  |
| **Fig. 7 – Boid debug info** |

In this iteration, some settings are highlighted in red. This is because any Boolean values are indicated by either green (true) or red (false). This makes it easier to quickly glance and understand what the current setting is. Everything under the “Simulation Settings” section is clickable, including the non-Boolean variables. Left clicking increments the value and right clicking decrements it.

At this stage I also added a sight range indicator which allows the user to easily identify when a unit enters another unit’s field of vision. This is drawn using the same line list that the force indicators use. There is almost no decrease in performance when drawing the sight range indicator.

# Boid Iterations

|  |
| --- |
|  |
| **Fig. 8 – VBCube boids** |

|  |
| --- |
|  |
| **Fig. 9 – CMOGO boids** |

As previously mentioned, the Boid class has been through multiple iterations. The class as it stands has very few necessary links to other classes. This has allowed me to switch between visual representations multiple times. The original rendition inherited from the pre-made VBCube class and therefore each boid was represented by a small cube in world space. I then applied a transform to the cube to achieve the design shown in Fig. 8. Due to the low reliance on other classes I decided to change the boid’s to allow different types to be easily distinguishable. I changed the polymorphism to inherit from CMOGO as opposed to VBCube. I used <http://www.turbosquid.com> to download two unique models which I felt would represent the level of the boid. The first two free .fbx models I found were a cockroach (http://www.turbosquid.com/FullPreview/Index.cfm/ID/851549) and a crab (http://www.turbosquid.com/FullPreview/Index.cfm/ID/762723). Changing the boid from a cube to a model was as simple as changing the initializer for the boid class to load in a given model.

A little later into the development I decided to switch back to using a vertex buffer object. The reason for this was that I could easily add extra vertices to the vertex buffer to display additional information such as the force arrows and sight range indicators. The class is still easily changeable to be able to use imported models although the user would have to either scrap the debug information or create a new object that inherits from VBGO and is parented to each boid.

# Class Diagram

## Gameobject

The **GameObject** class is used for any object which may or may not appear in 3D space. Below is a diagram representing the inheritance of all classes below **GameObject**.

|  |
| --- |
| C:\Users\David\Desktop\Doxy\html\class_game_object.png |
| **Fig. 10 – GameObject Diagram** |

|  |  |  |
| --- | --- | --- |
| **Attributes** | | |
| **Data Type** | **Variable Name** | **Purpose** |
| Vector3 | m\_pos | Coordinates of object in 3D space |
| Float | m\_pitch | Pitch of object |
| Float | m\_yaw | Yaw of object |
| Float | m\_roll | Roll of object |
| Vector3 | m\_scale | Scale of object on each axis |
| Matrix | m\_worldMat | World matrix for object |
| Matrix | m\_rotMat | Rotation matrix for object |
| Matrix | m\_fudge | Fudge matrix for object |

|  |  |  |
| --- | --- | --- |
| **Functions** | | |
| **Return Type** | **Function Name** | **Purpose** |
| Vector3 | GetPos() | Returns m\_pos |
| Void | SetPos(Vector3 \_pos) | Sets m\_pos |
| Void | SetPitch (float \_pitch) | Sets m\_pitch |
| Void | SetYaw (float \_yaw) | Sets m\_yaw |
| Void | SetRoll (float \_roll) | Sets m\_roll |
| Void | SetPitchYawRoll (float \_pitch, float \_yaw, float \_roll) | Sets m\_pitch, m\_yaw and m\_roll |
| Void | SetScale (Vector3 \_scale) | Sets m\_scale |
| Void | SetScale (float \_scale) | Sets m\_scale using float |
| Float | GetPitch () | Returns m\_pitch |
| Float | GetYaw () | Returns m\_yaw |
| Float | GetRoll () | Returns m\_roll |

## BoidManager

The **BoidManager** class inherits from the **GameObject** class and is used to manage everything to do with the **Boids**. This class handles the spawning, killing, fleeing, chasing and reacting for every **Boid** in the simulation.

|  |
| --- |
| C:\Users\David\Desktop\Doxy\html\classboid_manager.png |
| **Fig. 11 – boidManager Diagram** |

|  |  |  |
| --- | --- | --- |
| **Attributes** | | |
| **Data Type** | **Variable Name** | **Purpose** |
| Vector<Boid\*> | toSpawn | Contains all **Boids** to be spawned |
| Vector<Boid\*> | myBoids | Contains all alive **Boids** |
| Boid\* | cursor | **Boid** to be used as cursor |
| Vector<Waypoint\*> | m\_waypoints | Contains all **Waypoints** |
| Vector<Type\*> | m\_types | Contains all **Types** |
| ULONGLONG | lastSpawnTime | Time at last **Boid** was spawned |

|  |  |  |
| --- | --- | --- |
| **Functions** | | |
| **Return Type** | **Function Name** | **Purpose** |
| Void | Tick(GameData\* \_GD) | Performs all updates for each **Boid** |
| Void | Draw(DrawData\* \_DD) | Draws each **Boid** |
| Boid\* | SpawnBoid(int type) | Spawns a **Boid** and returns it |
| Boid\* | RespawnBoid(Boid\* \_B, bool keepPos) | Respawns a **Boid** |
| Boid\* | GetHighestBoid() | Returns highest tier **Boid** |
| Void | BreedBoids(Boid\* a, Boid\* b) | Spawns a new **Boid** copying stats |
| Void | RespawnAllBoids(bool keepPos) | Respawns all **Boids** |
| Void | DeleteBoid(Boid\* b) | Deletes a specific **Boid** |
| Void | DeleteBoid(int type) | Deletes first **Boid** of given type |
| Void | DeleteAll() | Deletes all **Boids** |
| Void | AddWaypoint(wayPointType \_type, int \_affects, Vector \_pos, float \_aoi) | Creates a **Waypoint** |
| Void | AddWaypoint(Waypoint\* w) | Creates a **Waypoint** |
| Void | DeleteAllWaypoints() | Deletes all **Waypoints** |
| Void | AddType(Type\* t) | Adds new **Type** to m­\_types |

## Camera

The **Camera** class inherits from the **GameObject** class and is used to give the user a viewpoint into the simulation. The **Camera** has a position and a target and will always look toward that target.

|  |
| --- |
| C:\Users\David\Desktop\Doxy\html\class_camera.png |
| **Fig. 12 – Camera Diagram** |

|  |  |  |
| --- | --- | --- |
| **Attributes** | | |
| **Data Type** | **Variable Name** | **Purpose** |
| Matrix | m\_projMat | Projection matrix |
| Matrix | m\_viewMat | View matrix |
| Vector3 | m\_target | 3D coordinates of location to look |
| Vector3 | m\_up | Determines which way is up |
| Float | m\_fieldOfView | Field of view |
| Float | m\_aspectRatio | Aspect ratio |
| Float | m\_nearPlaneDistance | Min view distance |
| Float | m\_farPlaneDistance | Max view distance |

|  |  |  |
| --- | --- | --- |
| **Functions** | | |
| **Return Type** | **Function Name** | **Purpose** |
| Void | Tick(GameData\* \_GD) | Performs update |
| Void | Draw(DrawData\* \_DD) | Draws **Camera** (not used) |
| Matrix | GetProj() | Returns m\_proj |
| Matrix | GetView() | Returns m\_view |
| Vector3 | GetTarget() | Returns m\_target |
| Vector3 | GetUp() | Returns m\_up |

## PredCamera

The **PredCamera** class inherits from the **Camera**. It uses the **BoidManager** class to find the highest tier **Boid** and then sets that as it’s target and follows it around.

|  |
| --- |
| C:\Users\David\Desktop\Doxy\html\class_pred_camera.png |
| **Fig. 13 – PredCamera Diagram** |

|  |  |  |
| --- | --- | --- |
| **Attributes** | | |
| **Data Type** | **Variable Name** | **Purpose** |
| Boid\* | m\_targetObject | Projection matrix |
| Vector3 | m\_dpos | View matrix |

|  |  |  |
| --- | --- | --- |
| **Functions** | | |
| **Return Type** | **Function Name** | **Purpose** |
| Void | Tick(GameData\* \_GD) | Performs update |
| Boid\* | GetTarget() | Returns target **Boid** |
| Void | ChangeTarget(Boid\* \_target) | Sets target **Boid** |

## Light

The **Light** class inherits from the **GameObject** class. It is used to add colour to all game objects which use a vertex buffer. It has an ambient colour and a flat colour.

|  |
| --- |
| C:\Users\David\Desktop\Doxy\html\class_light.png |
| **Fig. 14 – Light Diagram** |

|  |  |  |
| --- | --- | --- |
| **Attributes** | | |
| **Data Type** | **Variable Name** | **Purpose** |
| Color | m\_colour | Base colour of the light |
| Color | m\_ambientColour | Ambient colour of the light |

|  |  |  |
| --- | --- | --- |
| **Functions** | | |
| **Return Type** | **Function Name** | **Purpose** |
| Void | Tick(GameData\* \_GD) | Performs update |
| Void | Draw(DrawData\* \_DD) | Draws **Light** |
| Color | GetColour() | Returns colour |
| Color | GetAmbCol() | Returns ambient colour |
| Void | SetColour() | Sets colour |
| Void | SetAmbCol() | Sets ambient colour |

## VBGO

The VBGO class inherits from the GameObject class. It is used to create an object using an index buffer and a vertex buffer.

|  |
| --- |
| C:\Users\David\Desktop\Doxy\html\class_v_b_g_o.png |
| **Fig. 15 – VBGO Diagram** |

|  |  |  |
| --- | --- | --- |
| **Attributes** | | |
| **Data Type** | **Variable Name** | **Purpose** |
| ID3D11Buffer \* | m\_VertexBuffer | DirectX vertex buffer (triangle topology) |
| ID3D11Buffer \* | m\_LineVertexBuffer | DirectX vertex buffer (line topology) |
| ID3D11Buffer \* | m\_IndexBuffer | DirectX index buffer |
| UINT | m\_numPrims | Number of primitives |
| D3D\_PRIMITIVE\_TOPOLOGY | m\_topology | DirectX topology of object |
| ID3D11VertexShader \* | m\_pVertexShader | DirectX vertex shader |
| ID3D11InputLayout \* | m\_pVertexLayout | DirectX vertex layout |
| ID3D11PixelShader \* | m\_pPixelShader | DirectX pixel shader |
| ID3D11ShaderResourceView \* | m\_pTextureRV | DirectX shader resource |
| ID3D11Buffer \* | m\_pConstantBuffer | DirectX constant buffer |
| void \* | m\_pCB | Constant buffer |
| ID3D11SamplerState \* | m\_pSampler | DirectX sampler |
| ID3D11RasterizerState \* | m\_pRasterState | DirectX raster state |

|  |  |  |
| --- | --- | --- |
| **Functions** | | |
| **Return Type** | **Function Name** | **Purpose** |
| Void | BuildIB (ID3D11Device \*\_GD, void \*\_indices) | Builds index buffer |
| Void | BuildVB (ID3D11Device \*\_GD, int \_numVerts, void \*\_vertices) | Builds triangle vertex buffer |
| Void | BuildLineVB (ID3D11Device \*\_GD, int \_numVerts, void \*\_vertices) | Builds line vertex buffer |

## Boid

The Boid class inherits from the VBGO class. Every boid in the simulation uses the Boid class, including obstacles.

|  |
| --- |
| C:\Users\David\Desktop\Doxy\html\class_boid.png |
| **Fig. 16 – Boid Diagram** |

|  |  |  |
| --- | --- | --- |
| **Attributes** | | |
| **Data Type** | **Variable Name** | **Purpose** |
| Type\* | m\_type | Defines the type of boid and it’s behavior |
| Waypoint\* | finish | Where the boid is attempting to go |
| Waypoint\* | outpost | Limit on how far the boid can see/move |
| Int | m\_weight | How many boids has this boid eaten |
| Vector3 | m\_direction | Which direction is it facing |
| Bool | m\_alive | Is this boid alive |
| Float | m\_health | Current health |
| Float | m\_speed | Current speed |
| Float | max\_speed | Max speed |
| Float | m\_sight | Sight range |
| Ulonglong | lastUpdateTickCount | Last time this boid ate or starved |
| Ulonglong | lastBreedTickCount | Last time this boid bred |
| Float | breedDelay | How long in between this boid breeding |
| Myvertex\* | m\_vertices | Vertex list as array |
| Vector<Myvertex> | lineVertices | Vertex list as vector |

|  |  |  |
| --- | --- | --- |
| **Functions** | | |
| **Return Type** | **Function Name** | **Purpose** |
| Void | Tick(GameData \*GD) | Tick the boid |
| Void | Draw(DrawData \*DD) | Draw the boid |
| Void | initialize() | Initialize vertices |
| Void | Damage(Float \_dmg) | Reduce boid health and kill it if health < 0 |
| Void | Eat() | Increase size and weight |
| Void | Starve() | Decrease size and weight |
| Void | SetSpeed(Float speed) | Set current speed |
| Void | SetMaxSpeed(Float speed) | Set max speed |
| Void | SetSight(Float sight) | Set sight range |
| Void | SetDirection(Vector3 dir) | Set heading |
| Void | SetType(Type \*type) | Set type |
| Void | SetBreedDelay(Float \_d) | Set how long delay is between breeding |
| Void | SetWeight(Int w) | Set weight |
| Void | Breed() | Breed this boid |
| Waypoint\* | GetFinish() | Return the finish waypoint |
| Waypoint\* | GetOutpost() | Return the outpost waypoint |
| Float | GetSpeed() | Return current speed |
| Float | GetMaxSpeed() | Return max speed |
| Float | GetSight() | Return sight range |
| Vector3 | GetDirection() | Return current direction |
| Type\* | GetType() | Return type |
| Bool | GetAliveState() | Return true if boid is alive |
| Bool | GetBreedingStatus() | Return true if boid can breed |
| Int | GetWeight() | Return current weight |
| Float | GetHealth() | Return current health |
| Vector3 | GetScale() | Return current scale as Vector3 |
| Float | GetFloatScale() | Return curren scale as Float |

Waypoint

The waypoint class inherits from the VBGO class. Every boid is given a start, outpost and finish waypoint when it is spawned.

|  |
| --- |
| C:\Users\David\Desktop\Doxy\html\class_waypoint.png |
| **Fig. 17 – Waypoint Diagram** |

|  |  |  |
| --- | --- | --- |
| **Attributes** | | |
| **Data Type** | **Variable Name** | **Purpose** |
| Int | typeToAffect | The type of boid this waypoint affects |
| waypointType | myType | The type of waypoint this is |
| Float | areaOfInfluence | How far does this waypoint influence |
| myVertex\* | m\_vertices | Every vertex for this waypoint |
| vector<myVertex> | lineVertices | Used to temporarily hold debug line vertices |

|  |  |  |
| --- | --- | --- |
| **Functions** | | |
| **Return Type** | **Function Name** | **Purpose** |
| Void | Tick(GameData \*GD) | Tick the waypoint |
| Void | Draw(DrawData \*DD) | Draw the waypoint |
| Void | initialize() | Initialize vertices |
| Vector3 | GetToward(Boid\* b) | Returns a vector from the given boid to the waypoint |
| Vector3 | GetNormalizedToward(Boid\* b) | Returns a normalized vector from the given boid to the waypoint |
| Float | GetAreaOfInfluence() | Returns areaOfInfluence |
| Int | GetTypeToAffect() | Returns typoToAffect |
| Int | GetMyType() | Returns myType |
| Void | SetAreaOfInfluence(int \_new) | Sets areaOfInfluence |
| Void | SetMyType(waypointType \_new) | Sets myType |
| Void | SetAreaOfInfluence(float aoi) | Sets areaOfInfluence |

# Critical Evaluation

During development I encountered several situations which required me to completely re-think my approach to the solution. One such instance was when I decided to make every boid into a variant of a single class as opposed to having multiple classes. I started a new solution and used my previous iteration as a reference in order to write the new classes. I also took this opportunity to heavily optimize the Tick() function in the BoidManager class. This was around the same time as the updated code base was released by my lecturer which I was able to use due to starting again.

Issues with optimization arose from the very beginning and framerate loss was noticeable when as little as fifty boids were spawned. The first step I took to optimizing this was by analyzing the code to find the highest strain functions which took the longest to perform. I found that the slowest section of code was the checking between each boid. The loop was set out like so:

Loop through each boid in array

Loop through each boid in array

Compare first boid with second boid

End Loop

End Loop

As you can see the number of comparisons increases exponentially for each boid in the simulation. At twenty boids there are 20 x 19 = 380 comparisons. This was causing severe frame-rate loss so I implemented a small change in attempt to reduce the number of comparisons per tick. I changed the loop to only check boids from the current boid onwards. Here is an outline of this in pseudo code:

Loop through each boid in array

Loop through each boid in array starting after first boid

Compare first boid with second boid

End Loop

End Loop

This means that the comparisons for twenty boids are changed from 380 to 178. This is an optimization of 2.13 times. This resulted in a dramatic framerate increase. I later changed back to looping through the set of boids twice again. The reason for reverting was that the time taken to perform the calculations for both predator and prey in the second iteration took slightly longer than just doing the calculations separately and looping through twice.

Another optimization that I carried out was to add a preliminary check at the beginning of each comparison to check if the second boid is within the sight range of first boid. The reason behind this was that there will never be an interaction between boids that cannot see one another. This was a change that had no impact on the functionality of the code but made a significant impact on the speed of the simulation. Soon after implementing this change however, I noticed that some boids were ignoring collisions and also not getting eaten by their predators. After experimenting for a while, I found that if one boid had a larger sight range than another and a check had already been performed between the two using the smaller sight range, the check with a larger sight range would not be performed. In order to fix this issue I had to perform the check if either of the boids were in either sight range.

## Suitability as a Game Asset

My primary concern when designing and creating this simulation was to keep it suitable to be imported into a game as an external asset. The main method I employed to adhere to this concern was to make all variables defined outside of a function. Using fixed variables, otherwise known as “magic numbers”, means that certain behaviors are unchangeable outside of the code. By keeping parameters in an external file, game designers can modify the variables of the simulation to what they require for their game. For example, if the designer wishes the boids to travel from one waypoint to another and then restart when they reach their destination, they can do this by setting the parameters file to include a START waypoint, a FINISH waypoint and by setting RespawnOnFinish to true.

All of the functionality of the simulation can be obtained through a single class. This means that the user can just load this project into their solution and include the “BoidManager” class. The solution can then create a new instance of the BoidManager class and tick it every time their game ticks.

# Data File Layout

Fig. 17 shows an example set of parameters found in “SimulationParameters.dat”. The section starts with a <PARAMETERS> tag which tells the simulation that until it encounters an ending tag (</PARAMETERS>) that the following input will be parameters for the simulation. The program then reads in each line and checks for a parameter with the name given on the left of the equals. If it finds a matching parameter it set’s the value to the value given on the right of the equals.

|  |  |
| --- | --- |
| <parameters>  groupStrength = 0  groupDistance = 200  groupHeading = 0.7  boidMaxSpeed = 20  mapSize = 500  starvationTime = 5000  spawnDelay = 200  canBreed = 0 | showDebugMenu = 1  showDebugForces = 1  showDebugSight = 0  showDebugWaypoints = 1  cursorObstacle = 0  respawnOnFinish = 1  respawnOnDeath = 0  obstacleSize = 40  </parameters> |
| **Fig. 17 – Example parameters** | |

The next type of element that can be used in the parameters file is the <type> tag. This is how the user defines a new type. Fig. 18 demonstrates an example type which applies to any boid with an ID of ‘2’. The definitions are similar to the definitions for the parameters although instead of using an equals character to separate the parameters and the values, this section uses tags. The reason this section uses a different type of setup is that I was experimenting with different methods of saving and loading to/from a file and I kept in the multiple methods.

|  |
| --- |
| <type>  <id>2</id>  <scale>1</scale>  <breedDelay>20000.0</breedDelay>  <speed>20.0</speed>  <sight>50.0</sight>  <health>100.0</health>  <prey>1</prey>  </type> |
| **Fig. 18 – Example type** |

In this example, the scale of any boid with this type will be set to 1, the delay between breeding will be 20 seconds, the maximum speed will be 20, the sight range will be 50 and the starting health will be 100. If at any point the boid can see another boid of type 1, it will move towards it. To add more prey to the declaration the user can simply add more <prey></prey> tags with the ID of the prey.

The final type of declaration is for a waypoint. The layout for these declarations are exactly the same as the type declarations. An example waypoint setup can be seen in Fig. 19.

|  |
| --- |
| <waypoint>  <type>finish</type>  <size>500.0</size>  <typeToAffect>1</typeToAffect>  <x>1000.0</x>  <z>1000.0</z>  </waypoint> |
| **Fig. 19 – Example waypoint** |

This waypoint will be a finish waypoint, it will affect any boid with type ID of 1, will have an area of effect of 200 and it will be located at X: 1000, Y: 1000. The user can add as many different waypoints as they want to the parameters file and they will all be loaded in on startup.

# References

1. Thomas, D and Goudie, A. (2000) Food chain, food web. [Online]. *The dictionary of physical geography.* [online]. Available from: http://search.credoreference.com.ezproxy.uwe.ac.uk/content/entry/bkphsgeo/food\_chain\_food\_web/0 [Accessed 6 April 2015].

2. Love, C. (2008) Wildebeest Migration. *Animals: A visual encyclopedia*. [online] Available from: http://search.credoreference.com.ezproxy.uwe.ac.uk/content/entry/dkanimals/wildebeest\_migration/0 [Accessed 6 April 2015].

3. Denno, R. and Lewis, D. (2012) Predator-Prey interactions. *The Princeton guide to ecology. Princeton, NJ: Princeton University Press.* [online] Available from: http://search.credoreference.com.ezproxy.uwe.ac.uk/content/entry/prge/predator\_prey\_interactions/0 [Accessed 6 April 2015].

4. Reynolds, W, C. (1987) Flocks, Herds, and Schools: A Distributed Behavioral Model. *Computer Graphics* [online]. 21. (4). Available from: <http://www.cs.toronto.edu/~dt/siggraph97-course/cwr87/> [Accessed 01 April 2015]