David Dunnings  
13015063  
Simon Scarle

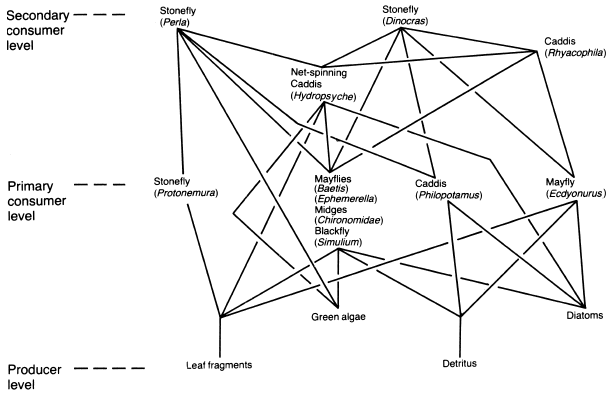
Simulated Worlds 2



# Introduction

Across the globe, every day, millions of prey are hunted by millions of 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*) [1].

**Fig.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 [3]. 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.

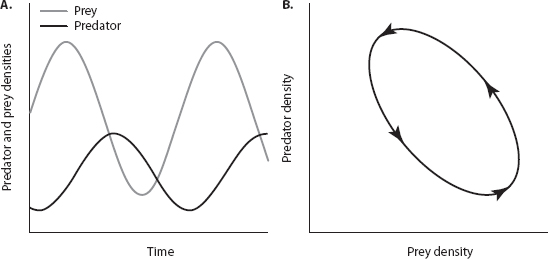
# Research

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 similar 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 [5] Denno and Lewis outline this “Reciprocal density effect”. The diagram below (A) illustrates this perfectly.



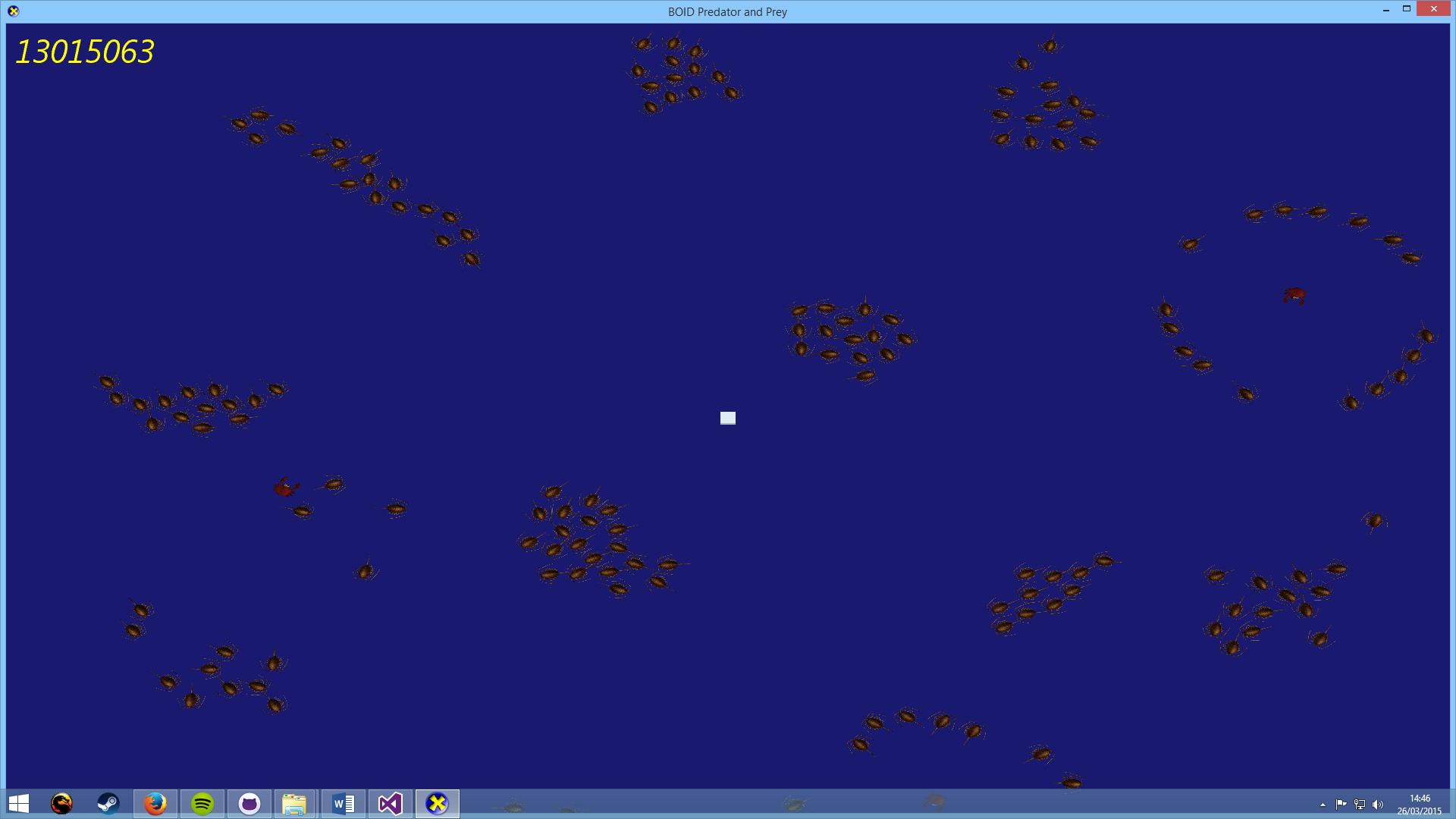
From this research I can deduce that if the prey in my simulation have the ability to reproduce then

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 1). 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*

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.

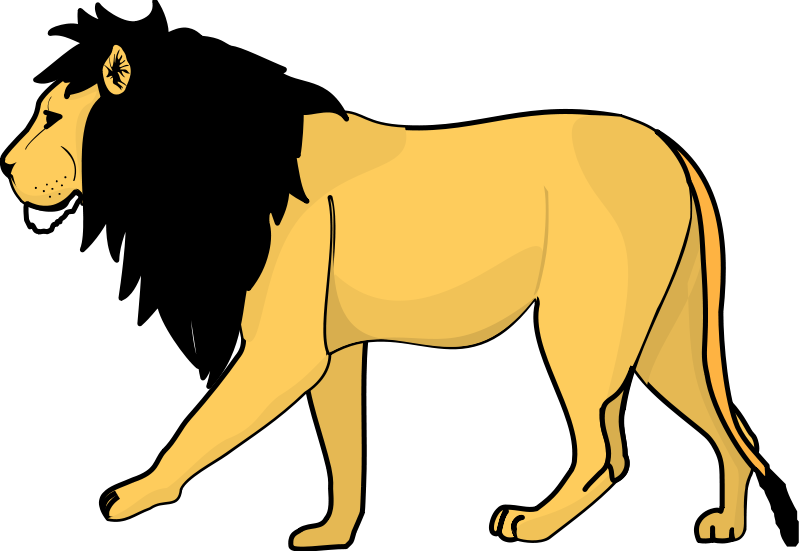
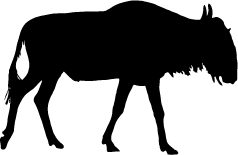
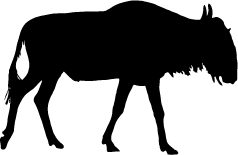
Iterations

Class iterations

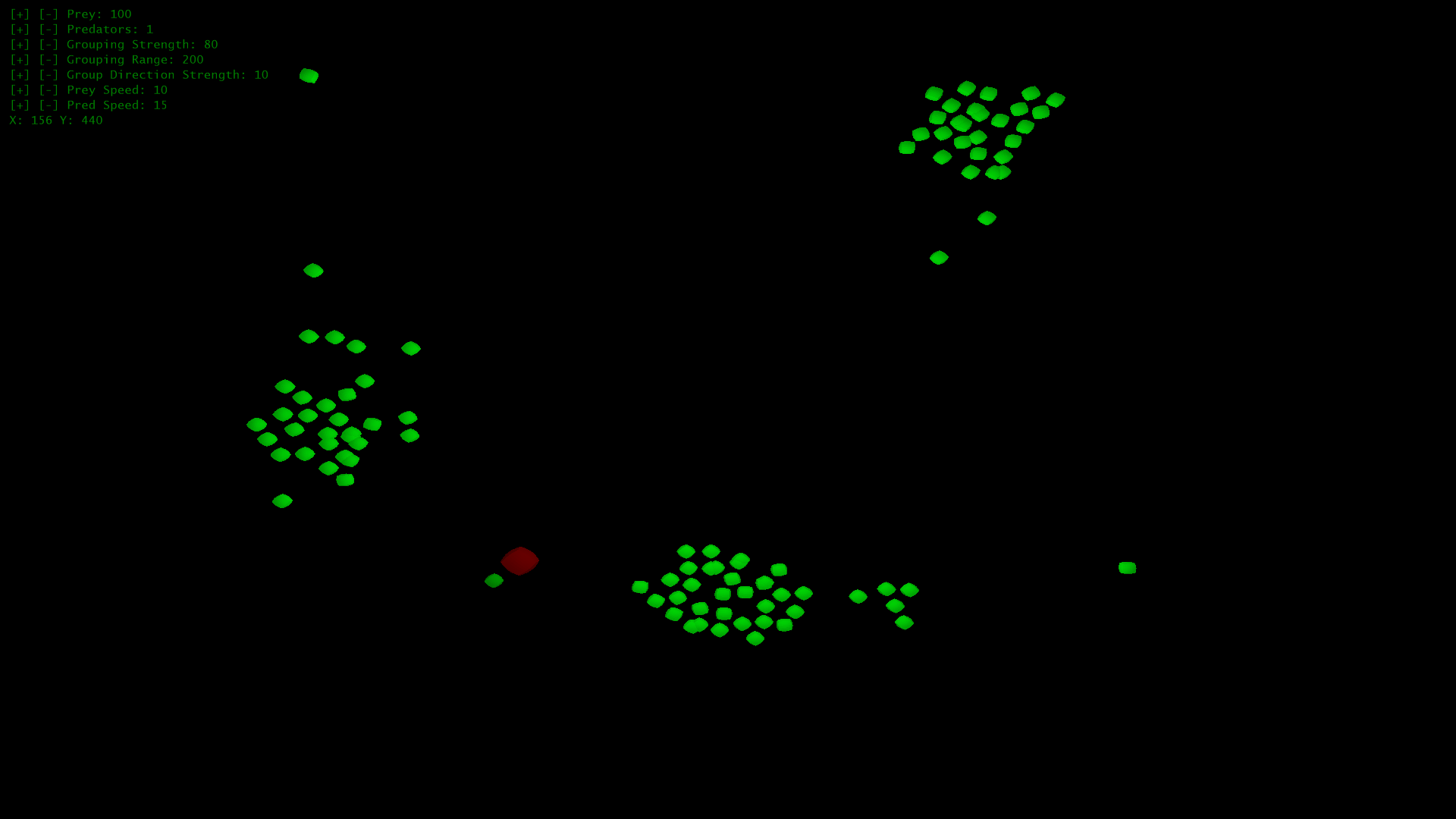
The first iteration of my boid simulation had a ***predatorBoid*** class and a ***preyBoid*** class. I quickly realized that this was unnecessary as both classes were almost identical, except for the polarity of the forces applied being reversed. Therefore I merged both classes into a parent class ***Boid***. 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:

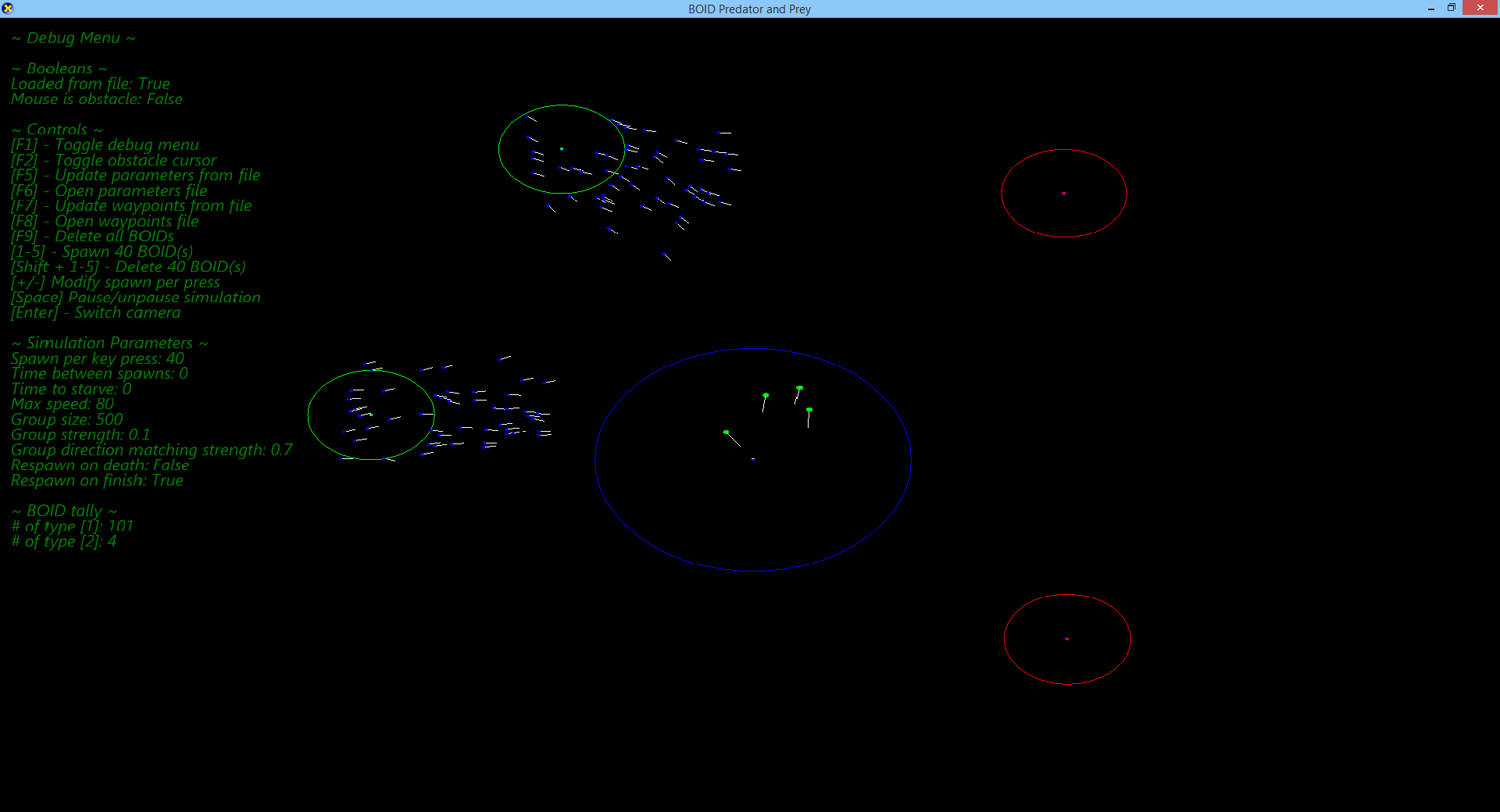
[0] boid\_OBSTACLE, [1] boid\_WILDEBEEST, [2] boid\_LION

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].



Debug Iterations

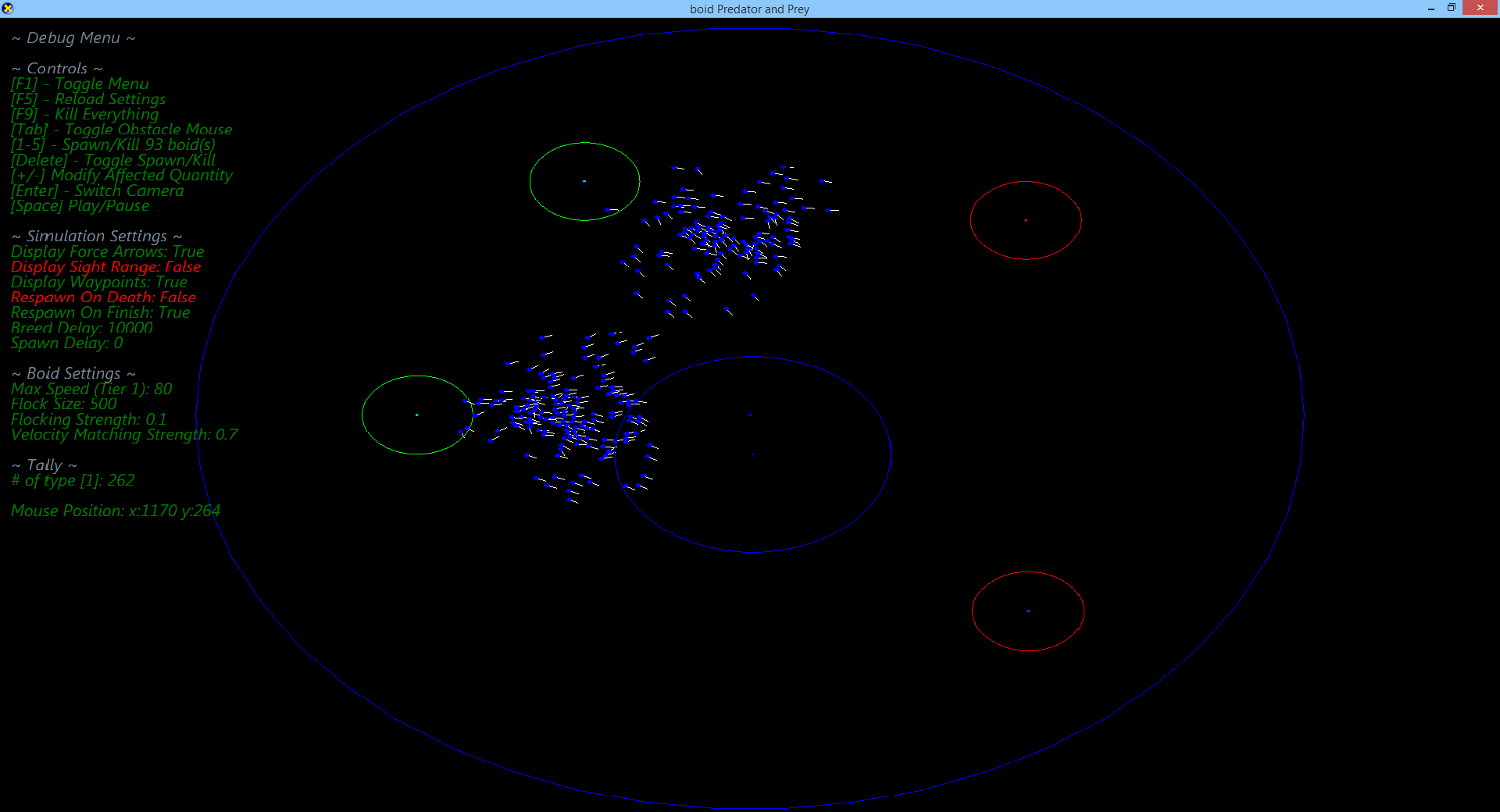
The first version of my simulation to include a debug interface listed each simulation parameter along the left of the screen. There were plus and minus buttons next to each parameter which incremented/decremented their associated value. These were achieved by using a large nested if statement. The program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

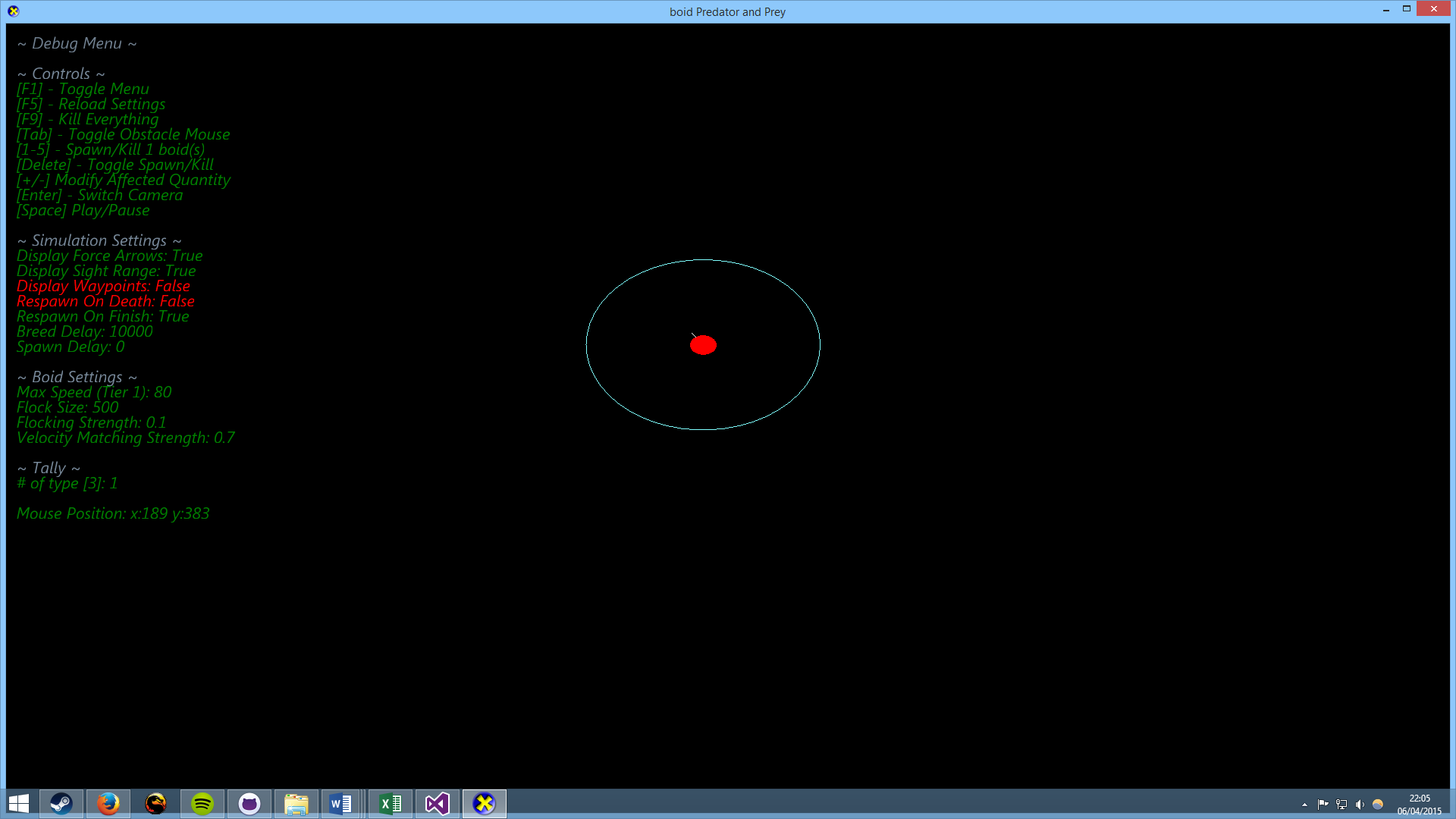


The next version of the GUI removed the clickable “buttons” and instead opted for a more keyboard-centralized interface. The debug menu started with a couple of true/false statements to display whether the simulation parameters have been loaded from file and if the mouse is being used as an obstacle. 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 the following interface:

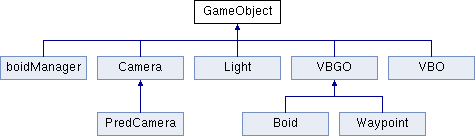


As you can see, some settings are highlighted in red. This is because any Boolean values are indicated by either green (true) or red (false). This makes it easier to quickly glance and understand what the current setting is. Everything under the “Simulation Settings” section is clickable, including the 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 a unit’s field of vision.

# 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**.



|  |  |  |
| --- | --- | --- |
| **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.

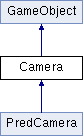


|  |  |  |
| --- | --- | --- |
| **Attributes** | | |
| **Data Type** | **Variable Name** | **Purpose** |
| Vector<Boid\*> | toSpawn | Contains all **Boids** to be spawned |
| Vector<Boid\*> | myBoids | Contains all alive **Boids** |
| Boid\* | cursor | **Boid** to be used as cursor |
| Vector<Waypoint\*> | 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.

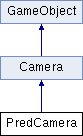


|  |  |  |
| --- | --- | --- |
| **Attributes** | | |
| **Data Type** | **Variable Name** | **Purpose** |
| Matrix | m\_projMat | Projection matrix |
| Matrix | m\_viewMat | View matrix |
| Vector3 | m\_target | 3D coordinates of location to look |
| Vector3 | m\_up | Determines which way is up |
| Float | 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.



|  |  |  |
| --- | --- | --- |
| **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.

|  |  |  |
| --- | --- | --- |
| **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.

|  |  |  |
| --- | --- | --- |
| **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 |

# References

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

2. <http://www.cs.toronto.edu/~dt/siggraph97-course/cwr87/>

3. Wildebeest Migration (2008). [Online]. In C. Love & C. Stamps (eds.). Animals: A visual encyclopedia. London, United Kingdom: Dorling Kindersley Publishing, Inc.. Available from: http://search.credoreference.com.ezproxy.uwe.ac.uk/content/entry/dkanimals/wildebeest\_migration/0 [Accessed 6 April 2015].

4. <http://link.springer.com.ezproxy.uwe.ac.uk/content/pdf/10.1140%2Fepjb%2Fe2013-40210-5.pdf>

5. Denno, R. and Lewis, D. (2012) Predator-Prey interactions. [Online]. In S. Levin , S. Carpenter & H. Charles J. (eds.). The Princeton guide to ecology. Princeton, NJ: Princeton University Press. Available from: http://search.credoreference.com.ezproxy.uwe.ac.uk/content/entry/prge/predator\_prey\_interactions/0 [Accessed 6 April 2015].