FLAME GPU Tutorial

Mozhgan Kabiri Chimeh, Paul Richmond

1 Introduction

This document describes a hands-on tutorial exercise on how to use FLAMEGPU. An overview of the framework is provided separately in the HPCS17 [1] tutorial paper. You can download it from www.flamegpu.com/tutorial.

The tutorial will use Amazon EC2 G2 Instances, optimised for graphics and compute intensive applications. To access this system, you have been provided with a username/password as well as a public domain name.

This document will first introduce a predator prey model and then provide a set of instructions for how to execute this model on the EC2 system. A number of exercises will then guide you through the modification and experimentation of the predator prey model.

Note: If you have a laptop with CUDA-enabled GPU and have a CUDA 7.5 or CUDA 8.0 installed on it, feel free to skip Section 3. However, it is expected for all attendees to use AWS as these machines have been pre-configured for you.

2 Predator-Prey model

This section describes the FLAMEGPU implementation of a simple predator-prey model. The specification of the model is based on the previous implementations on FLAME [1] and NetLogo [2] but adds group cohesion and avoidance behaviours. Section 2.2 gives an overview of the model description (XMML) and behaviour scripting (functions.c) of the FLAMEGPU implementation of this model. For a complete guide to FLAME GPU functionality readers are directed to the FLAME GPU Technical Report and user guide available from the FLAMEGPU [3] website documents page.

2.1 Model description

Predator prey models capture the dynamic behaviours of competing species within an environment to demonstrate population dynamics as species try to survive. The model is important as it captures emergent macro level phenomenon which can be observed within other domains such as economics.

In its most basic form, a two species system consists of both prey and predators. Preys reproduce but are eaten by predators and predators reproduce and eat prey. Typically such systems can be represented using non linear differential equations known as Lotka-Volterra equations. In a more advanced form of the model microscopic behaviours of the species can be considered by using agent based representation. E.g. The predators can have energy which is depleted as they move. Prey can form cohesive groups (flocking) and dynamically avoid predators, while predators can move towards preys. If a predator and prey are close enough (i.e. within a fixed kill radius), the predator is able to eat the prey and increase the amount of energy it has. Predators die when they run out of energy.

The model has been implemented in various frameworks Agent Based Modelling frameworks (e.g:NetLogo and FLAME). For this tutorial the model which will be implemented have the following behaviours.

- Predator and Prey agents move and act in a sequence of iterations (time steps).
- Predator and Prey agents have an real valued (x, y) position and velocity which is integrated over time.

[1] FLAME GPU: Complex System Simulation Framework

- Predator agents have an initial velocity. A Predator adjusts its velocity to follow the nearest prey within a synthetic limited vision.
- A predators energy is reduced by 1 unit of energy each time it moves.
- Prey are eaten by the closest predator agent within a fixed (kill) radius.
- A predators energy increases each time it catches a prey
- A predator dies if it has run out of energy.

More specific details of the model features are described below.

Prey catching There is no limit on the amount of prey a predator can eat per iteration. If a predator is closest to multiple prey within the specific (kill) radius, a predator will kill/eat them all. A prey cannot be shared between multiple predators. If there are multiple predators within the kill radius then only the closest will eat it.

Reproduction Each species has a rate of reproduction. When an agent reproduces the parents energy is shared evenly between parent and child, so both will have half of the parents energy for the next iteration.

2.2 FLAMEGPU implementation

The FLAMEGPU implementation of the predator prey model is based on the assumption made in previous section. All agents have a real valued position (`x`, `y`) within a continuous environment. The environment bounds are wrapped to form a continuous torus space in which agents can move. Agents have a velocity (`vx` and `vy`) in which they will move and a steering velocity (`steer_x` and `steer_y`) which dictates how they should adjust their velocity according to local interactions.Predator agents have a variable (`life`) which contains the level of energy. An agent variable type is used for visualisations of the simulation where unique type values correspond to different colours within the visualisation. The type is set by the initial values of the agent and does not change over time.Each agent has a unique identifier variable (`id`). When a predator kills a prey, it is necessary to communicate identifiers to ensure correct behaviour as will be explained later in this section.

All communication between agents in FLAMEGPU is indirectly performed via messages. In in this model we have three message types which are required to perform the mechanism for killing. E.g. ,

1. Both prey and predator output their (x, y) coordinates to a prey or predator location message respectively.
2. Preys read predator locations and perform evasive behaviour to avoid them
3. Predators read prey locations and move towards the closest visible prey
4. Prey read each others location and move towards each other to form cohesive groups.
5. Both predators and prey read each others (same species) location and avoid each other within close proximity to avoid collisions.
6. Prey reads locations of predators and calculates the nearest predator within the kill distance. If a predator is within the kill distance then it notifies this predator by sending a `prey_eaten_message` message with the predators `id` and it dies (is removed for the simulation).
7. Predators read the `prey_eaten_message` messages and checks the to see if their `id` has been communicated by any dead prey. If the message indicates that the predator ate some prey, then the predator increases its energy/life accordingly.

2.3 The FLAMEGPU Model

A FLAMEGPU model specification consists of; **environment** which holds global information related to the simulation such as constant variables and behaviour script file names, **agents** which describes variables and functions of the agent and **messages** which represent information communicated between agents. Each aspect is introduced in the following subsections with examples. If you are confident of the models behaviour and the FLAMEGPU syntax then feel free to move onto the hands on material (4).

2.3.1 FLAMEGPU environment

For the Predator-Prey model a number of self explanatory model parameters are described within the environment section. Each has a default value which may be overwritten by the initial agents states file used at runtime. The environment section dictates that the agent function behaviours are located in the `functions.c` file and that there are named CPU functions for performing initialisation, simulation step and simulation exit behaviours. In this case these functions are used for logging population data from the simulation. This is much more efficient than outputting the entire agent state list each iteration for post processing.

```
<gpu:environment>
    <gpu:constants>
        <gpu:variable>
            <type>float</type>
            <name>REPRODUCE_PREY_PROB</name>
            <defaultValue>0.03</defaultValue>
        </gpu:variable>
        <gpu:variable>
            <type>float</type>
            <name>REPRODUCE_PREDATOR_PROB</name>
            <defaultValue>0.03</defaultValue>
        </gpu:variable>
        <gpu:variable>
            <type>int</type>
            <name>GAIN_FROM_FOOD_PREDATOR</name>
            <defaultValue>15</defaultValue>
        </gpu:variable>
        <gpu:variable>
            <type>int</type>
            <name>GAIN_FROM_FOOD_PREY</name>
            <defaultValue>5</defaultValue>
        </gpu:variable>
    </gpu:constants>
    <gpu:functionFiles>
        <file>functions.c</file>
    </gpu:functionFiles>
    <gpu:initFunctions>
        <gpu:initFunction>
            <gpu:name>initLogFile</gpu:name>
        </gpu:initFunction>
    </gpu:initFunctions
    <gpu:exitFunctions>
        <gpu:exitFunction>
            <gpu:name>closeLogFile</gpu:name>
        </gpu:exitFunction>
```

```
    </gpu:exitFunctions>
    <gpu:stepFunctions>
        <gpu:stepFunction>
            <gpu:name>outputToLogFile</gpu:name>
        </gpu:stepFunction>
    </gpu:stepFunctions>
</gpu:environment>
```

2.3.2 Agent memory

An agent tagged by `<xagents>` consists of memory and functions. An example of the Prey agents variable declarations is provided below. Each agent variable has a type (of either integer, float or double) and a name which will be used to build a C struct data member representation during the code generation process.

```
<memory>
    <gpu:variable>
        <type>int</type>
        <name>id</name>
    </gpu:variable>
    <gpu:variable>
        <type>float</type>
        <name>x</name>
    </gpu:variable>
    <gpu:variable>
        <type>float</type>
        <name>y</name>
    </gpu:variable>
    <gpu:variable>
        <type>float</type>
        <name>type</name>
    </gpu:variable>
    <gpu:variable>
        <type>float</type>
        <name>fx</name>
    </gpu:variable>
        <gpu:variable>
        <type>float</type>
        <name>fy</name>
    </gpu:variable>
    <gpu:variable>
        <type>float</type>
        <name>steer_x</name>
    </gpu:variable>
        <gpu:variable>
        <type>float</type>
        <name>steer_y</name>
    </gpu:variable>
    <gpu:variable>
        <type>int</type>
        <name>life</name>
    </gpu:variable>
</memory>
```

2.3.3 Agent functions

The Functions aspect of the agent model description describes the properties of the behaviours that the agent will have. Note: This is not where the actual behaviour is described but the properties relating to the behaviour functions. Table 1 and Table 2, shows the list of defined functions per agent in the predator-prey model.

Table 1: Summary of the Prey agent functions for Predator Prey model - no grass included

Function	Description
prey_output_location	each prey agent outputs information to be read by other agents
prey_avoid_pred	prey agents avoiding predator agents
prey_flock	flocking between prey agents
prey_move	movement of prey agents
prey_eaten	prey agents get killed by predator agents
prey_reproduction	regeneration of preys

Table 2: Summary of the Predator agent functions for Predator Prey model

Function	Description
pred_output_location	each predator agent outputs information to be read by other agents
pred_follow_prey	predator agents follow prey agents
pred_avoid	predator agents avoid each other
pred_move	movement of predator agents
pred_eat_or_starve	predator agents gain energy or starve
prey_reproduction	regeneration of predators

Functions are applied to agents in a specified initial_state and will result in the agent moving into a next_state (or the same state). Furthermore, functions can have conditions. For example, a function which simulates agent death might check a variable incremented each iteration called life-cycles to see if it has reached a maximum number. Only agents meeting this condition would perform the behaviour and move into a dead state. The predator-prey model is a stateless model (each agent has only a single state) as functions do not require any function conditions. The dependencies and order of functions is described within the models function layers which dictate the ordering of execution.

Functions can have either a single input or output (or neither). Inputs and outputs are in the form of messages which are a collection of variables which are persistent from the point in which they are output to the end of the simulation iteration (at which point they are destroyed). A function description within a model requires that any inputs or outputs are fully specified.

Listing 2.3.3 shows the structure of `<pred_output_location>` Function description. In this example, the agent function has a message output. The reallocate flag indicates that the agent function will not result in agent agent deaths (and so the process or removing dead agents can be bypassed. The inclusion of a random number generator for the function is dictated by the `<gpu:RNG>` element.

```
<gpu:function>
    <name>pred_output_location</name>
    <currentState>default1</currentState>
    <nextState>default1</nextState>
    <outputs>
        <gpu:output>
            <messageName>prey_location</messageName>
            <gpu:type>single_message</gpu:type>
        </gpu:output>
    </outputs>
    <gpu:reallocate>false</gpu:reallocate>
    <gpu:RNG>false</gpu:RNG>
</gpu:function>
```

Note that for each message type defined within the XMML model definition, the dynamically generated simulation API will create a message output function. Agents can only output a single message (or an optional message) per agent function. In Listing 2.3.3, the pred output location function is shown for completeness. The arguments of the `add_pred_location_message` function are the ordered variables defined for the message type (see Agent Messages sub section).

```
__FLAME_GPU_FUNC__ int pred_output_location(xmachine_memory_predator* xmemory,
    xmachine_message_pred_location_list* pred_location_messages)
{
  add_pred_location_message(pred_location_messages, xmemory->id, xmemory->x, xmemory->y);
  return 0;
}
```

For agent functions which have message inputs, iterating over messages within the function is also possible by the use of dynamically generated message API functions. Listing 2.3.3 shows a complete agent function with input messages demonstrating the iteration of a message list and outputting a different message. The while loop (`while (pred_location_message)`) continues until the `get_next_pred_location_message` return a NULL or a false value. Each agent and message variable can be accessed by name from a C structure which is code generated from the model file.

```
__FLAME_GPU_FUNC__ int prey_eaten(xmachine_memory_prey* xmemory, xmachine_message_pred_location_list*
    pred_location_messages, xmachine_message_prey_eaten_list* prey_eaten_messages)
{
  int eaten = 0;
  int predator_id = -1;
  float closest_pred = PRED_KILL_DISTANCE;

  //Iterate the predator location messages until NULL is returned which indicates all messages have been read.
  xmachine_message_pred_location* pred_location_message = get_first_pred_location_message(
     pred_location_messages);
    while (pred_location_message)
  {
    //calculate distance between prey and predator
    float2 predator_pos = float2(pred_location_message->x, pred_location_message->y);
    float2 prey_pos = float2(xmemory->x, xmemory->y);
    float distance = length(predator_pos - prey_pos);

    //if distance is closer than nearest predator so far then select this predator as the one which will eat the prey
    if (distance < closest_pred)
    {
      predator_id = pred_location_message->id;
      closest_pred = distance;
      eaten = 1;
    }

    pred_location_message = get_next_pred_location_message(pred_location_message, pred_location_messages);
  }

  //if one or more predators were within killing distance then notify the nearest predator that it has eaten this prey via a
      prey eaten message.
  if (eaten)
    add_prey_eaten_message(prey_eaten_messages, predator_id);

  //return eaten value to remove dead (eaten == 1) agents from the simulation
  return eaten;
}
```

After adding the function definition to the XML model file, the new function should be added to the function layers. Layers represent synchronisation points in the execution of the agent functions. It is assumed that functions on the same layer execute simultaneously so functions which have a dependency via messages should not be within the same layer. The `prey_output_location`, `prey_follow_prey` and `prey_flock` functions are in different layers for this reason.

In the case of `prey_output_location` function, it will be added to the layer so that it executes in the same layer as `pred_output_location`. The other functions are are added after the output location function to ensure they begins executing after all prey agents have completed outputting each prey's location. E.g.

```
<layer>
    <gpu:layerFunction>
        <name>prey_output_location</name>
    </gpu:layerFunction>
    <gpu:layerFunction>
        <name>pred_output_location</name>
    </gpu:layerFunction>
</layer>
<layer>
    <gpu:layerFunction>
        <name>prey_flock</name>
</gpu:layerFunction>
<gpu:layerFunction>
        <name>pred_avoid</name>
</gpu:layerFunction>
</layer>
<layer>
...
</layer>
```

2.3.4 Agent messages

Messages are defined as a simple collection of variables, a partitioning type and a buffer size indicating the maximum number of messages which may exist during any single simulation step. The `pred_location` message is shown in the below example. It corresponds with the message used in Listing 2.3.3 and Listing 2.3.3. All functions within the predator prey model use a partition type of `<partitioningNone>`. This indicates that the message reading functions will iterate all messages in the list (i.e. 2.3.3 will iterate over every message).

```
<messages>
    <gpu:message>
        <name>pred_location_message</name>
        <variables>
            <gpu:variable>
               <type>int</type>
               <name>id</name>
            </gpu:variable>
            <gpu:variable>
               <type>float</type>
               <name>x</name>
            </gpu:variable>
            <gpu:variable>
               <type>float</type>
               <name>y</name>
            </gpu:variable>
        </variables>
        <gpu:partitioningNone/>
        <gpu:bufferSize>262144</gpu:bufferSize>
    </gpu:message>
```

..
<messages>

2.4 The FLAMEGPU flow diagram

Figure 1 shows a dependency graph for the Predator-Prey model. This indicates the order in which functions will be executed and shows any dependencies functions have on messages.

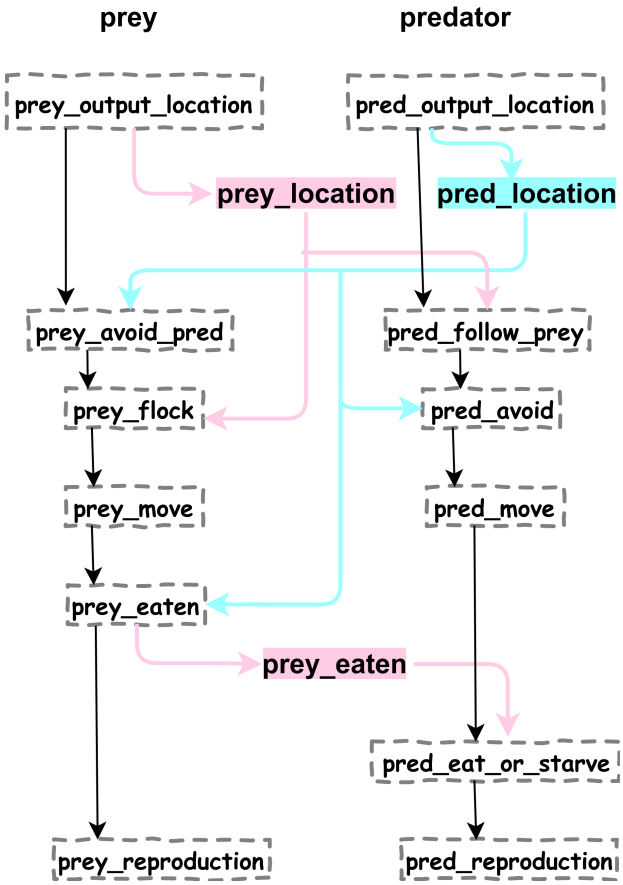

Figure 1: Flow diagram for Predator-Prey model without grass

2.5 Input data generation and Setup

The initial data for the model is generated using a simple C++ program. The program generates the specified number of predator and prey agents in random positions (`x,y`) between [-1,1] with random velocities (`fx,fy`) between [-1,1]. Each predator is given an amount of energy (`life`) which is randomly selected from the interval of [0,40].

The program generates an `0.xml` output file containing the initial data. Table 3 shows the various parameters used for the implementation of the model in FLAMEGPU.

Table 3: Summary of parameters in Predator Prey model without grass

Parameter	Value
Initial number of prey agents	800
Initial number of predator agents	400
Amount of energy gained from eating prey	75
Probability of predators reproducing	0.03
Probability of preys reproducing	0.05

3 Connecting to the Linux Instance

To get started with FLAMEGPU tutorial, you first need to connect to the AWS G2 instances. These are optimised for graphics-intensive application[2]. We have previously installed Ubuntu 16.04 and other required applications.

3.1 Using SSH in Linux

Connect to the Linux Instance using SSH command and the provided usernamepassword. Replace the *public_dns_name* with the ones given to you by the instructors.

```
ssh username@public_dns_name
```

3.2 Using SSH in PuTTY

1. Putty is a free software application and can be downloaded from https://www.chiark.greenend.org.uk/ sgtatham/putty/ and run putty.exe
2. In the Category pane, select Session and complete the following fields:
   - In the Host Name box, enter `username@public_dns_name`.
   - Under Connection type, select SSH.
   - Ensure that Port is 22.
3. (Optional) If you plan to start this session again later, you can save the session information for future use. Select Session in the Category tree, enter a name for the session in Saved Sessions, and then choose Save.
4. In order to view images, we will need to configure X11 window forwarding so that the remote display windows are opened on our local machine. If you are using Linux as your host operating system then you do not need to install anything. If you are using Windows you will need to install an X11 window server so that the remote machine where we are executing FLAME GPU can display on your local machine. The XMing software is free and is recommended. From the command line, X11 forwarding can be enabled with the ssh X option. From Putty in Windows, this can be enabled from the Connection-SSH-X11 dialogue.
5. Choose Open to start the PuTTY session
6. If this is the first time you have connected to this instance, PuTTY displays a security alert dialog box that asks whether you trust the host you are connecting to.
7. Choose Yes. A window opens and a terminal prompt asking for your username: `login as:` Enter your username.
8. Enter your password and you are connected to the instance.

3.3 Transferring files

Linux users can transfer files from your computer to the `username` home directory via `SCP` command.

```
scp /path/yourfile.txt username@dns_name:~
```

Windows users can use the PuTTY Secure Copy client (PSCP) (a command-line tool) to transfer files between Windows computer and the Linux instance.

```
pscp C:\path\yourfile.txt username@public_dns:/home/username/yourfile.txt
```

[2]Each GPU has 1,536 CUDA cores

4 Getting started with FLAMEGPU

To get started with FLAMEGPU, connect to the Amazon instance by following the given instructions in Section 3. Then, download the tutorial version of FLAMEGPU from GitHub:

```
git clone https://github.com/FLAMEGPU/Tutorial.git
```

Note: that this is not the full FLAMEGPU version as all examples other than the predator prey model are omitted. The full version of FLAMEGPU can be downloaded from the website.

A typical top-level directory layout is as below:

- **FLAMEGPU:** contains the templates and XML schemas that are used to generate CUDA GPU code. These should not be modified by the users.
- **bin/x64** and **bin/linux-x64:** The location of the console and visualisation binaries for each of the examples. There is a Linux shell script for each example which will start the simulation with an initial states file (and the number of iterations to simulation in console mode)
- **doc:** The FLAMEGPU technical report and user guide in addition to reports for specific example models.
- **examples:** The location of the model files for FLAMEGPU examples and the location to create your own models.
- **include:** Some common include files required by FLAMEGPU
- **lib:** Any library dependencies required by FLAMEGPU
- **media:** 3D models used for some of the visualisations
- **tools:** A number of tools for generating function script files from XML model files and running template code generation in windows.

We are going to work with the predator-prey model (Section 2.1) in the examples folder. Navigate to the `examples/PreyPredator` directory and call `make` to perform all the the FLAMEGPU code generation and code compilation stages.

```
cd examples
cd PreyPredator
make
```

This will process the XML model and build a console [3] version of the model in release mode. To run the executable, simply run `make run_console`. Alternatively, navigate back to the FLAMEGPU bin directory and call the *run script* which will have been generated by the make process.

```
cd ../../bin/linux-x64/}
./PreyPredator_console.sh
```

Both the `make run_console` and *run script* commands will call the executable by passing the `iterations/0.xml` starting stats file as the program augment with a default single simulation iteration. The output will be a csv file `iterations\PreyPred_Count.csv` which will have logged the population counts for the initial state and for the simulation step. Note: XML output is disabled but can be re-enabled by setting the `XML_OUTPUT` definition in the automatically generated `src/dynamic/main.cu` file to 1. After rebuilding and running the simulation again this will create an XML file (saved in the location of the initial input file) for each iteration which will contain the state of the agents after applying a single simulation iteration to the agents (in the same formal

[3] In the original version of FLAMEGPU, calling *make* would build both console and visualisation version of the model in release mode

as `0.xml`. You can view this file (via `cat` command) to see how the agent positions and other properties have changed.

Examine the run script by looking at the parameters passed to the simulation. The parameters are the initial model file and the number of simulation runs (iterations). Note that by default, the number of iterations is set to 1. In order to modify the number of iterations, simply pass an argument to the shell script as follows:

```
make run_console iter=500
```

5 Exercise 01

In exercise one, we are going to build and execute the simulation program for the basic Predator-Prey model, followed by plotting the output results. Navigate to the `examples/PreyPredator` directory.

```
cd examples/PreyPredator
make
```

Now, we run the simulation for 150 iterations:

```
make run_console iter=150
```

The generated csv file contains the number of prey and predator agents per iteration. Navigate to the `iterations` folder and plot the result:

```
cd examples/PreyPredator/iterations/
gnuplot make_plot_PreyPred.gp
```

You can transfer the `.png` file to your local machine to view it or you can use the `display` command. e.g.

```
display PreyPredator.png
```

Your plot should be similar to Figure 2.

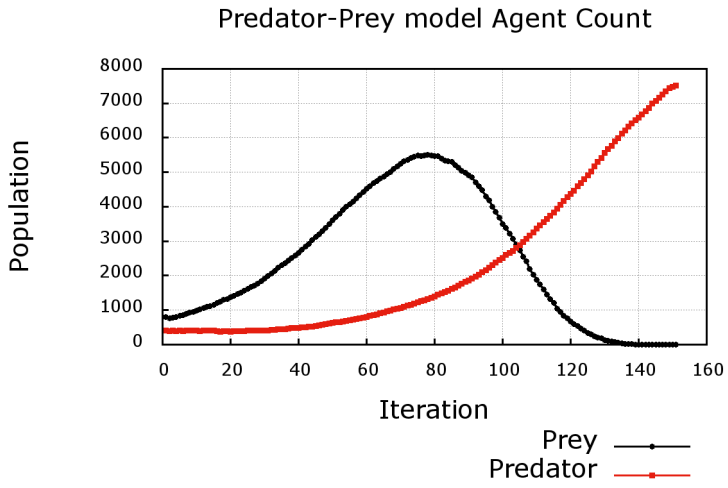

Figure 2: Examples of predator-prey model simulation - (after 150 iteration)

6 Exercise 02

Now, let's change the parameters in the initial data and see how it affects the behaviour. To generate a new randomised set of initial data (0.xml), navigate to `XMLGenerator` folder.

2.1 Compile and re-run the executable with different parameters.

```
cd /examples/PreyPredator/XMLGenerator/
g++ -std=gnu++11 xmlGen.cpp -o xmlGen
./xmlGen ../iterations/0.xml 800 400 0.05 0.03 50
```

, where 800 is the number of predators, 400 is the number of preys, 0.05 and 0.03 are the reproduction rates for both prey and predator, and 50 is the predator's energy gain.

2.2 Re-run the executable again for 300 iterations and plot the results. Your plot should be similar to Figure 3. You can observe the predator prey behaviour where both species become extinct after certain number of iterations.

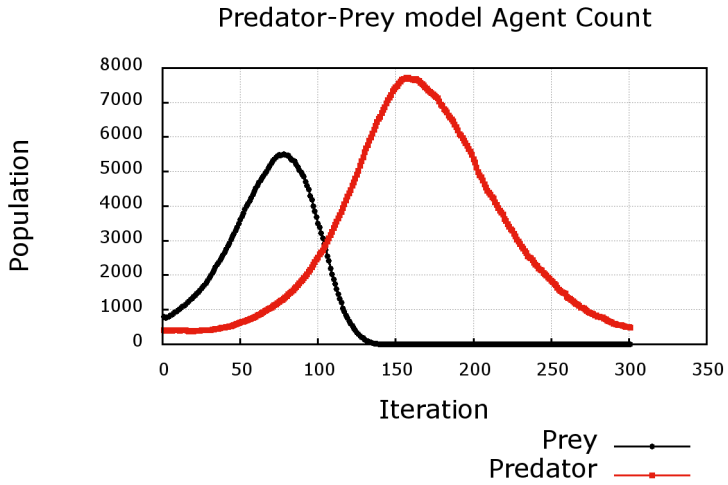

Figure 3: Examples of predator-prey model simulation - no grass included (after 300 iteration)

2.3 Change the other parameters or the iteration number to see how the behaviour changes. Try to modify the simulation parameters to produce oscillating population dynamics.

7 Exercise 03 (Optional)

In this exercise, we are going to extend our model to include grass. When grass or preys' source of food is included in the model [4]:

- A preys energy is reduced by 1 unit each time it moves.
- A preys energy increases each time it eats grass.
- A prey dies if it has run out of energy.
- Once eaten, grass regrows after a fixed number of iterations.

With respect to the implementation this will require , a fourth and fifth message type as grass should only be eaten by prey within a certain certain radius. The mechanism for "grazing" is as follows:

1. Grass agents read the positions of the prey from the location message. Then, each pick out the messages within the minimum distance and then post the ID of a prey within that distance to the `grass_eaten_message` message. Note that if there is more than 1 prey on within the minimum distance, the grass agent will be eaten by the closest prey.
2. Grass agents then modify the `active` agent variable to indicate they no longer have grass at this time. The colour also changes from "green" to "navy".

[4] based on NetLogo's implementation

3. Prey agents read the `grass_eaten_message` message which contain their ID. They then increase their energy accordingly.

4. Prey agents die if they do not have enough life/energy

Figure 4 shows a dependency graph for the Predator-Prey model with grass.

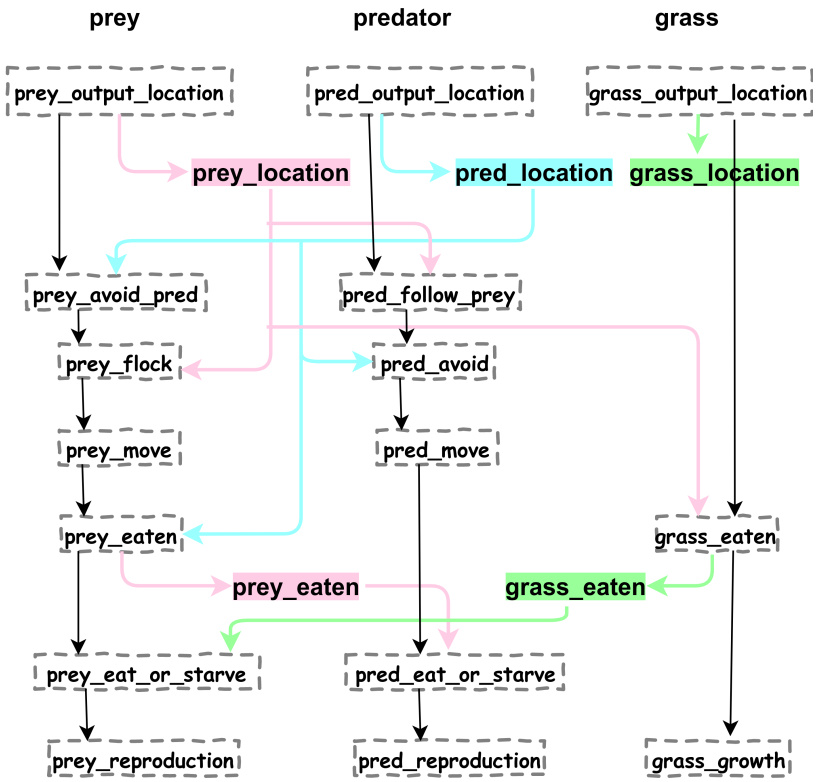

Figure 4: Flow diagram for Predator-Prey model with grass

For simplicity a Grass agent description has already be added to the XML model file with function descriptions matching the behaviour described above. In this exercise, you need to modify the `functions.c` file to add the following behaviour to the functions stubs described below:

3.1 **grass_output_location**: each grass agent outputs a *grass_location* message to provide information. In a more advanced model prey agent can use this information to migrate to areas of high food concentration.

3.2 **grass_eaten**: each grass agent should iterate over `prey_location_messages` and checks the distance between its location and the prey agent. If the grass is available and the distances less than `GRASS_EAT_DISTANCE`, then the grass is eaten by the closet prey and the regrowth cycle starts. Note that if there are multiple preys within the `GRASS_EAT_DISTANCE`, then the closet prey to the grass, eats it and outputs a message `grass_eaten` containing the ID of the prey who ate it.

Once the grass is eaten, its colour changes (`type` variable is set to a different colour) and it no longer will be available until the `death_cycles` reaches `GRASS_REGROW_CYCLES`).

3.3 **prey_eat_or_starve**: each grass agent iterates over `grass_eaten_messages` and checks the ID against it ID. If the grass eaten message indicates that this prey ate some grass then increase the preys life by adding energy. Moreover, if its life is less than 1, it dies.

3.4 **grass_growth**: If the the `death_cycles` variable is equal to `GRASS_REGROW_CYCLES`, then the grass agent becomes available and the `death_cycles` restarts and the colour will be set to green again. If the grass is not available (meaning the `death_cycles` variable is not equal to `GRASS_REGROW_CYCLES`), then we only increase the `death_cycles` variable.

In the case where grass is included, prey agents require an energy (`life`) variable similar to predators. This variable is randomly selected from the interval of [0,50]. The grass agent initial colour is set to "green". Once eaten, the `available` variable is set to 0 and it takes up to `GRASS_REGROW_CYCLES` iterations till the grass re-grow.

Now, generate a new initial data file using below parameters. Then re-build the model via `make` and run the simulation for 600 iterations `make run_console iter=600`.

```
cd /examples/PreyPredator/XMLGenerator/
g++ -std=gnu++11 xmlGen_IncGrass.cpp -o xmlGenEx3
./xmlGenEx3 ../iterations/0.xml 800 400 2000 0.05 0.03 75 50 100
```

, where 800 is the number of preys, 400 is the number of predators, 2000 is the number of grass, 0.05 and 0.03 are the reproduction rates for both prey and predator, 75 is the prey's energy gain, and 50 is the predator's energy gain.

Plot your results by running `gnuplot make_plot_PreyPred_IncGrass.gp`. Your plot should be similar to Figure 5.

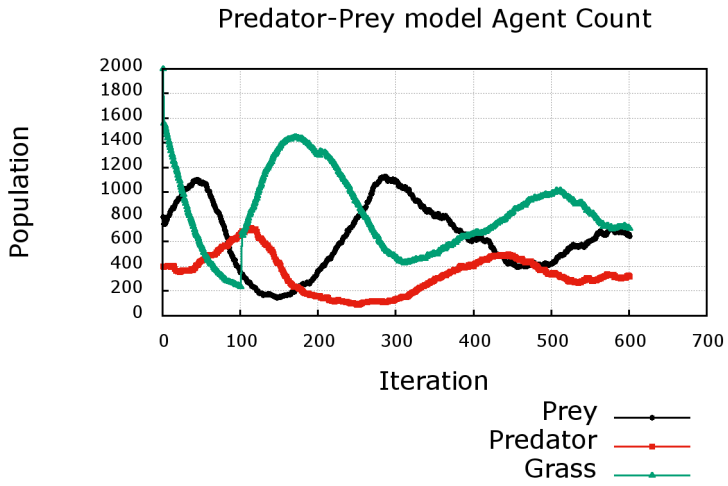

Figure 5: Examples of predator-prey model simulation with grass included

8 Experimenting with the Model

Try changing the parameters to see how this will change the behaviour of the agents causing the behaviours to change. If you have completed all the above exercises then try modifying the model file and functions file to allow prey agents to migrate to areas of high concentrations of food. You can implement this by adding an additional velocity term to the preys steering velocity and by allowing the prey agents to query the `grass_location` messages.

For more information on FLAMEGPU see the FLAMEGPU website[5] and the documentation which gives detailed instructions on all aspects of FLAMEGPU modelling. More examples can be found on FLAMEGPU GitHub repository (`https://github.com/FLAMEGPU/FLAMEGPU.git`).

You can download the solutions from GitHub by checking out the Exercise_3_solution branch. E.g.

```
git clone https://github.com/FLAMEGPU/tutorial -b Exercise_3_solution
```

[5] www.flamegpu.com

9 OpenGL Rendering of the Model

Note: This will not work on Amazon AWS images as OpenGL remote rendering is not easily supported. Skip this section unless you are running the examples on your own Ubuntu system. [6]

On your own machine, navigate to the FLAMEGPU bin folder (`bin/linux-x64/`) and run the *vis script* visualisation shell script. This will launch the simulation in visualisation mode.

You can build the model in visualisation mode rather than console (by default), and then run the executable as follows:

```
make Visualisation_mode
make run_vis
```

References

[1] G. L. Poulter, C. Greenough, and H. Oxford, "FLAME Tutorial Examples : Predation - a simple predator-prey model," Tech. Rep., 2014.

[2] U. Wilensky. (1997) Netlogo wolf sheep predation model. [Online]. Available: http://ccl.northwestern.edu/netlogo/models/WolfSheepPredation

[3] P. Richmond, "FLAME GPU Technical Report and User Guide (CS-11-03)," Department of Computer Science, University of Sheffield, Tech. Rep., 2011.

[6] Ubuntu users can install VirtualGL and connect to AWS via *vglconnect*. For more info, please refer to their website as this is beyond the scope of this tutorial.