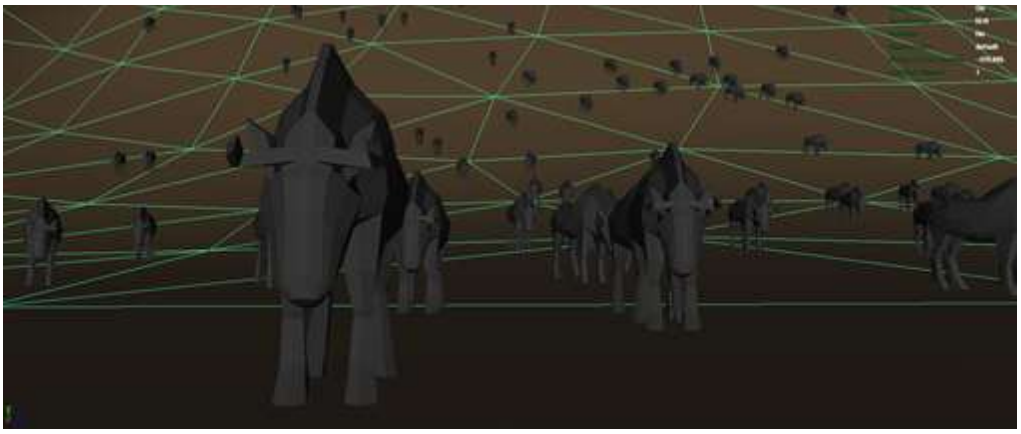
Master's Thesis

Animal Stampede Simulation



Akila Lakshminarayanan

Brian Tran

MSc Computer Animation and Visual Effects, **NCCA 2011-2012**

Abstract

Large crowd scenes with humans and animals are abundant in films. Shooting these shots can be very difficult, costly, or even nearly impossible. The problem of massive crowds has been previously addressed by various methods and programs. Presented here is a program that will specifically simulate an animal stampede. The goal of this project is to create a standalone program that will simulate an animal stampede, comprised of various animals, across an imported 3D terrain. The program will export the position and behaviour state data of each animal, which will be able to be imported into a 3D package such as Maya. The data will be used inside the 3D package to drive the animation of rigs using motion graphs. The end goal is to render out a realistic stampede scene. The program is written in C++ and OpenGL using the NCCA Graphics Library.

Table of Contents

List of Figures

1. Introduction

Crowd scenes are a common and an integral part of both live –action and animation films. Most of these scenes involve crowds of enormous size and variety hence making the process very complex and time consuming. Most of the live-action scenes manage to achieve the crowd effect by shooting with the real people or live creatures that are put to act or behave. On the other hand, simulating the same effect in computer graphics (CG) is a colossal task. Setting up motion for each character by hand for a crowd of large scale would take forever and the result may not look natural enough. Such group behaviours are hence programmed with help of complex math and physics and also with the use of Artificial Intelligence (AI) wherever required. This project addresses a specific problem of crowd behaviour, the simulation of animal stampede consisting of different types of animals.

One of the notable animal stampede scenes in films is the "Wildebeest stampede" sequence from Walt Disney's 2D animated film, The Lion King (Disney, 1994) which took two years to create a 2.5 minute sequence (Wikipedia, 2012).

Figure 1 : Wildebeest Stampede, *"The Lion King"* – (a)

Figure 2: Wildebeest Stampede, *"The Lion King"* – (b)
(Disney, 1994)

This thesis explains our approach to solve this problem of creating an efficient crowd system that simulates the animal stampede behaviour. The whole project is divided into several parts such as following a 3D terrain, flocking behaviour of a herd, avoiding obstacles on the terrain, adding natural looking paths and drag forces, importing the simulation into the 3D package Maya, adding models, animation and finally rendering. A crowd system is a widely researched area of computer graphics with a lot of work being done related to simulation of collective behaviour over the years. Some of such related works will be explained

in the next chapter (Chapter 2). The theory involved in the project along with the math and physics required are explained in Chapter 3. Design, implementation and the results are documented the next 3 chapters respectively.

2. Related work

2.1. Reynold's model

Any crowd will exhibit a basic collective behaviour which is common to the group and an individual behaviour underneath which is specific to each agent in the group. One of the earliest works in group behaviour was Craig Reynold's flocking algorithm which was a distributed behaviour model for flocks, herds and schools (Reynolds, 1987). The individuals of his flocking system are called "Boids" which are subjected to three rules cohesion, alignment and separation.

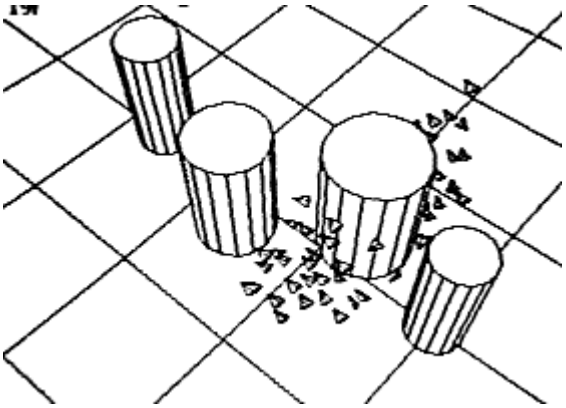


Figure 3: Reynold's Flocking model (Reynolds, 1987)

He also published a paper on steering behaviours where the behaviour of autonomous characters are explained using a hierarchical structure of containing the layers action selection, steering and locomotion. Steering behaviours such as seek, flee, pursuit, evasion, arrival, obstacle avoidance, wander, path following, containment along with the above mentioned flocking rules are explained and demonstrated in 2D.

2.2. Gompert's flocking on a 3D terrain

A slightly advanced form of flocking behaviour was created by Joel Gompert where a herd of animals are made to follow a 3D terrain. They also follow the three basic flocking rules while they move on the terrain. The herd is capable of going around the hills and other obstacles and they also follow natural paths that are defined considering their energy expenditure (Gompert, 2003). Here the animals' speed are modified based on their movement uphill or downhill using the slope of that particular area of the terrain there by keeping the complexity low. The implementation also focuses on goal seeking and obstacle avoidance.

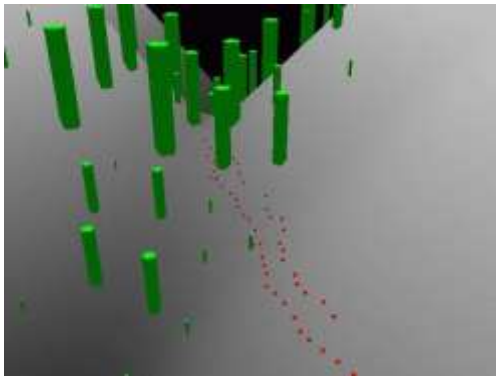


Figure 4: Flocking over 3D terrain (Gompert, 2003)

The program uses various parameters to drive the herd such as herd size, maximum speed, comfort-zone radius, danger-zone radius, visible flock make range, visible obstacle range, view angle and drag factor.

2.3. MASSIVE Software

MASSIVE (Multiple Agent Simulation System in Virtual Environment) is the most commonly used crowd-simulation software in the visual effects industry. It was first developed by Stephen Regelous for Peter Jackson's movie the *Lord of the Rings*. Massive uses fuzzy logic and pre-recorded animation clips to control the agents' behaviour. Each agent consists of a body and brain which helps them to react to each other and to their environment.

Figure 5: Battle scene from 'Lord of the Rings: Return of the King' for which MASSIVE won an award (3D Awards, 2004)

The body defines the physical aspects of an agent which is usually a hierarchy of connected segments

The brain part is a fuzzy logic network which controls the actions of the agents such as following an arbitrary terrain or avoiding an obstacle. Apart from Artificial Intelligence, the program also includes other features like rigid body dynamics (RBD) and cloth simulation that can be applied to the geometry used.

2.4. Moving through an Environment

Choi, Kim et al presented an interactive method to control the movement of characters through a cluttered environment. Their behavioural model was called Deformable motion where the characters' navigation is subjected to path planning, avoiding obstacles in a constrained environment. The program is a real-time interactive where the characters move through an environment with a lot of cylinders, steps and platforms by motion deformation. First the motion path which is considered as an elastic string undergoes deformation based on the constraints and then the poses at individual frames are squeezed into narrow passages (Choi, Kim et al 2011).

Figure 6: Deformable motion allowing characters move through constrained environment (Choi, Kim et al 2011)

2.5. Others

Numerous other contributions exist in the area of crowd simulation and multi-agents. Rossmann and Hempe et al created real-time crowd model which has several modules such as a finite state machine for the agents, their steering behaviour, locomotion, path planning and message passing ability (Rossmann and Hempe et al 2009). For the simulation to run interactively with more number of agents, some optimization techniques like multi-threading and cell space partitioning were implemented. This model uses modern hardware to optimize the results and it is therefore capable of simulating up to 2000 agents in real-time (Rossmann and Hempe et al 2009). Another multi-agent based model by Ruas and Marietto et al is a combination of "Distributed Artificial Intelligence" and "Computational Simulation". If focuses on social interactions such as co-operation, conflicts and so on (Ruas and Marietto et al 2011). In 2006, Petre

and Ciechomski et al published a paper on "Real-time navigating crowds: scalable simulation and rendering" where navigation graphs are pre-computed and are used in the simulation. This helps in spreading the crowd by individualizing the routes of the agents. The model also focuses on rendering of large crowds by using meshes and animations depending on their closeness to camera. Results show that their model can simulate up to 35,000 pedestrians in real-time.

3. Technical Background

3.1. Reynold's Flocking Algorithm

Craig W. Reynolds' distributed model for school, herds and flocks simulates the group behaviour by applying three simple rules to each agent known as a "Boid". The three steering behaviours cohesion, alignment and separation modifies the position and velocity of the boids to make then move as a unit at the same velocity at the same time avoid colliding with each other. Boids position and velocity are modified based on the position and velocity of its neighbouring boids. Only the boids that are within a certain distance are considered as neighbours and the rest are ignored while calculating the new position. The model was extended to avoid obstacles that are present in the environment and also exhibit goal seeking and path following behaviour (Reynolds, 1987).

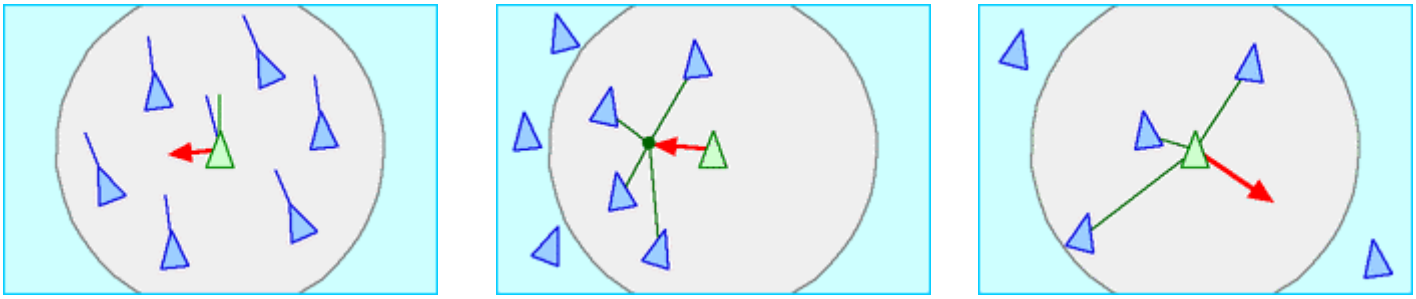


Figure 7: Alignment, Cohesion & Separation (Reynolds, 1987)

a) Cohesion – makes the agents move towards the centre of the flock so that they stay close to each other.

$$C\gamma = \frac{(\sum_{i=1}^{n} \rho_i)}{n}$$

$$C\gamma - Cohesion\ Velocity$$

$$n - Number\ of\ agents\ in\ the\ Neighbourhood$$

$$\rho_i - Position\ of\ Agent_i$$

b) Alignment – makes the agents move with the same velocity so that they move towards the average heading

$$A\gamma = \frac{(\sum_{i=1}^{n} \vartheta_i)}{n}$$

$$A\gamma - Alignment\ Velocity$$

$$n - Number\ of\ agents\ in\ the\ Neighbourhood$$

$$\vartheta_i - Velocity\ of\ Agent_i$$

c) Separation – makes the boids agents colliding with the neighbouring flock mates

$$S\gamma = \frac{\left(\sum_{j=1}^{c}\left(-\left(\rho - \rho_j\right)\right)\right)}{c}$$

$$S\gamma - Separation\ Velocity$$

$$c - Number\ of\ Colliding\ Agents$$

$$\rho - Current\ Agent$$

$$\rho_j - Colliding\ Agent_i$$

3.2. Barycentric Co-ordinates for finding the right triangle

The 3D terrain used here is a triangulated mesh imported from Maya. All the agents need to move based on the terrain. In order to make them follow the terrain, the first step is to find the triangle in which the agent is present. There are several methods that are available that finds if the given point is present in the triangle given its three vertices. One of the efficient methods is the use of Barycentric co-ordinates to test if the point lies inside a triangle. Barycentric co-ordinates are a form of homogeneous co-ordinates where the location of a point is specified as the centre of mass or Barycenter of masses placed at vertices of a triangle or a tetrahedron. Consider a set of points $\rho_0, \rho_1 \ldots . \rho_n$, an affine combination ρ for all the points can be written as

$$\rho = \alpha_0\rho_0 + \alpha_1\rho_1 + \alpha_2\rho_2 \ldots\ldots + \alpha_n\rho_n$$

Then, $\alpha_0 + \alpha_1 + \alpha_2 \ldots\ldots + \alpha_n = 1$ and $(\alpha_0, \alpha_1, \alpha_2, \ldots . \alpha_n)$ are called Barycentric co-ordinates of the points in space (Joy 1996)

For a triangle, 3 points say A, B and C form a plane in space. Any one of the points is chosen and all other locations on the plan are considered to be relative to that point. For example, we may consider A as the origin for the plane and consider the two edges that touch A which as C-A and B-A. These edges form vectors V0 and V1 that are used to move along their direction. To get to any point on the plane, we can move a certain distance along V0 and then along V1.

Hence any point on the plane can be represented by the equation

$$\rho = A + u * (C - A) + v * (B - A)$$

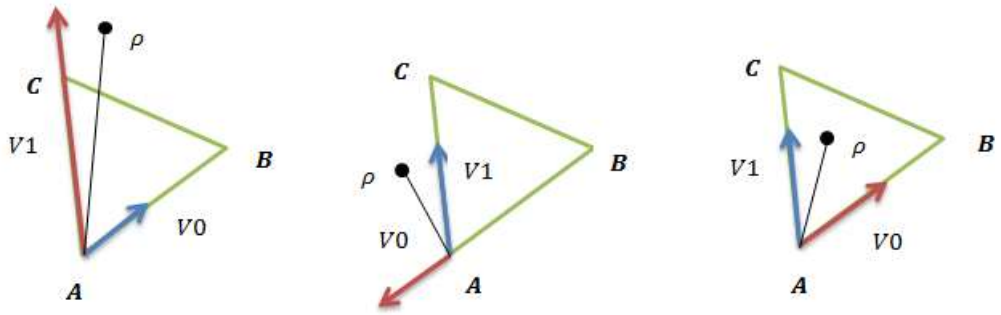


Figure 8: Point P lies inside or outside based on the direction in which vectors V0 & V1 move

Any point used with the above equation is outside the triangle,

$$\text{If } u \text{ or } v < 0 \quad \text{or}$$

$$u \text{ or } v > 1 \quad \text{or}$$

$$u + v > 1$$

The values of u and v are calculated by solving two equations formed from the above equation. Once these values are calculated we can easily find whether the given point lies in the triangle by checking for the above conditions.

4. Design & Implementation

4.1. Overall Pipeline

A stand-alone system is created in C++ & OpenGL which uses imported mesh from Maya and also exports the simulation data back into the 3D package to incorporate animation. A 3D terrain is modelled in Maya and the mesh is triangulated before being exported out as .obj file. The imported mesh is processed and its face information is written out to a text file to reduce the computation every time the same terrain data is used inside the program. The behaviour of different agents is scripted in Python and they are integrated with the C++ application. This is mainly to add flexibility to the application where different agents can be added to the system just by including their behaviour scripts. The vertex and fragment GLSL shader files are included which are used by C++ to render using OpenGL in real-time. NGL (NCCA Graphics Library) is used which is a library of classes with basic graphics functions implemented. Most of the OpenGL functions here are accessed through these NGL libraries.

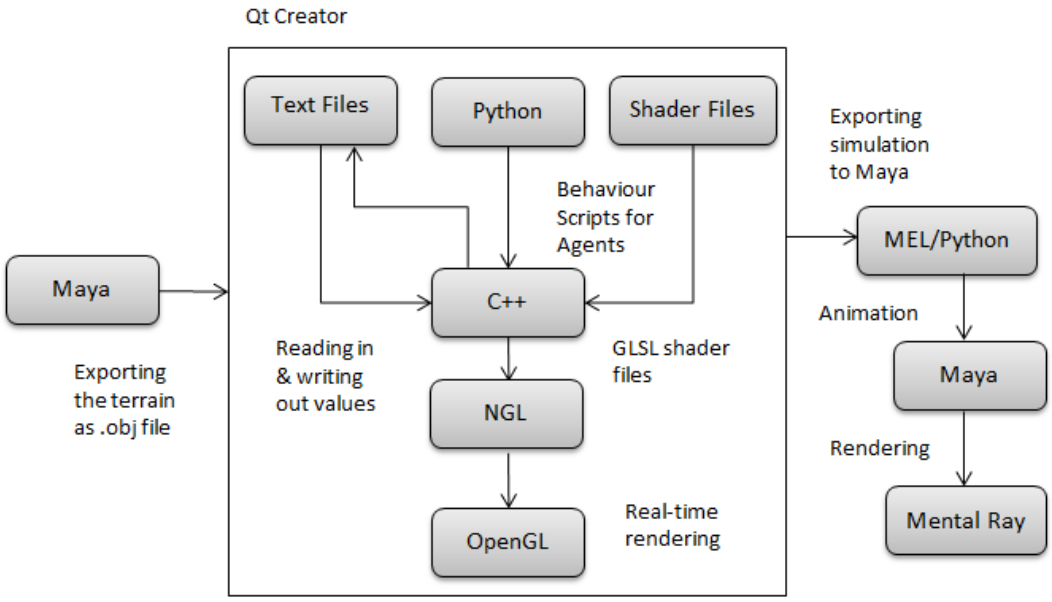


Figure 9: Overall System Pipeline

The output of the program or the simulation data is exported to Maya using MEL/Python API. A motion path is generated for each animal based on the positions from the simulated data. This is added to a scene containing the above 3D terrain in Maya. The agent geometry with animation is attached to the motion path generated. Finally, the stampede scene is rendered using Mental Ray renderer.

4.2. Classes

4.2.1. Herd class

This class maintains the information about the herd such as its size, neighbouring radius and speed. It takes care of the creation of different types of agents, make them flock together, avoid obstacles and adjust their speed. It also handles the terrain following by using the agent and the terrain data.

4.2.2. Terrain class

This holds the mesh data such as the mesh size, faces, vertices and normals and it draws the mesh at each frame. It also contains a function that finds the face in which the agent is currently present and returns the face number to the herd class.

4.2.3. Agent class

Agent class contains the agent position, direction, speed, type and colour. It also uses the Python scripts for the behaviour of different agents. It has a number of accessor and mutator methods for the other classes to use and modify agent attributes.

4.2.4. Face class

This class is used to store the vertices and normals of each face and a list containing the neighbouring faces of each face of the terrain.

4.2.5. Parser class

Parser class handles the text parsing functions for reading in the initial values from a text file. It also saves and reads the face information to and from text files.

4.2.6. GLWindow class

GLWindow handles the OpenGL rendering by calling the draw methods of various other classes. It calls the move method of the herd and updates the OpenGL window every frame.

4.2.7. MainWindow class

This class is has the main method that starts the application. It creates the GLWindow and also handles user interface by connecting the parameters in the UI with the values inside the program.

4.3. 3D Terrain

4.3.1. Processing imported terrain mesh

One of the important parts of the project is making the herd follow an uneven terrain. The terrain is modelled in Maya and the triangulated mesh is imported into the C++ program as .obj file using *ngl::Obj* class.

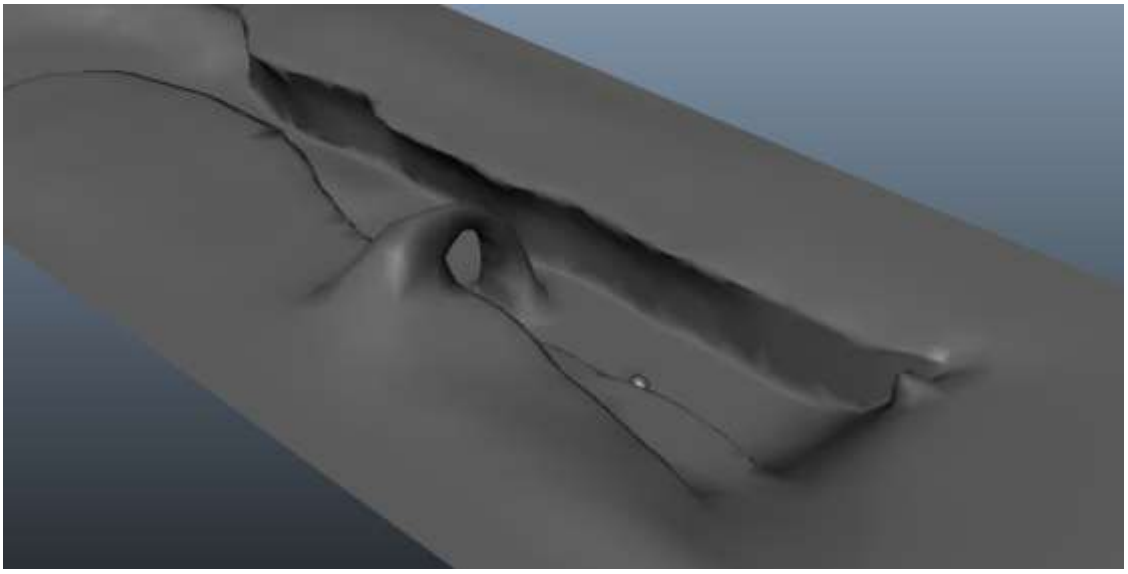


Figure 10: Terrain modelled in Maya

This ngl class also provides all the faces and vertices of the imported terrain along with other useful information like the minimum and maximum values of the mesh boundaries. The terrain in the Figure 10 is made up of thousands of triangular faces whose vertices and normals are frequently accessed and used in the program to move the agents. For example, to make an agent stick to the terrain the first step is to find the right face on which the agent currently lies. Checking every face in the terrain for each agent at each frame would be unnecessary and totally inefficient. Hence to increase the computation speed of the program, the terrain is pre-processed and the face information are computed and stored in text file for further use until a new terrain is added.

For each face in the terrain, its three vertices, the surface normal and a list of its neighbouring faces are computed and written out to a text file. So every time when the program searches for the face where agent lies, it checks the neighbour faces first instead of scanning through the whole mesh. If the agent is not found in a face's neighbours, the program checks the next level of neighbours (i.e.) all the neighbours of the previously checked neighbours.

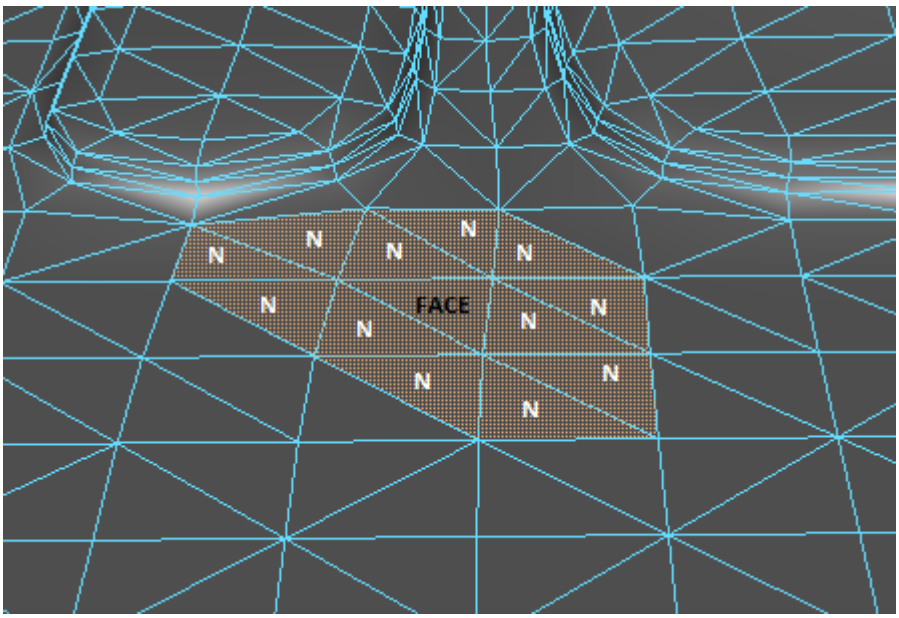
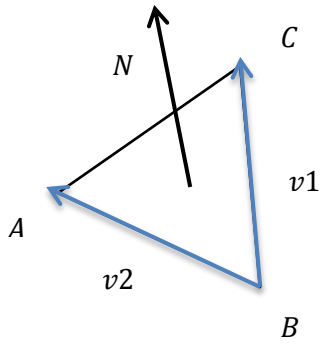


Figure 11: A face and its neighbouring faces

The surface normal for each face is calculated by taking the cross product between the vectors forming any two edges of the triangle.



Normal $N = v1\ X\ v2$

For each face,

Face Number	face0
Vertices	vertex0, vertex1, vertex2
Normal	normal
Neighbouring faces	nf1, nf2, nf3…

This whole process takes place only for a terrain and once the values are calculated and stored, they can be accessed at a very quickly from the text file using text parsing methods. The *Parser* class handles this task of reading the values from a text file at each frame for each agent.

4.3.2. Terrain following

The terrain following is implemented in a series of steps to make the agents stay on the terrain as they move. At each frame, the current position of the agent is used to find in which part of the terrain it is present. Once the right face is found, the agent is positioned on the terrain based on the normal of the face found. To handle this efficiently, each agent stores the previously hit face in a variable and if the agent is present in the same face as the previous frame then the program breaks out of the loop right there without checking any further. Parser class reads in the face information from the text file and stores them in a std::vector for the other class to access. The method to find if a position lies inside a faces uses Barycentric co-ordinates which was explained earlier in section 3.2. The method takes in the current face number and the position of the agent as input and returns a Boolean value to say if the point is inside or not. As discussed earlier, the method finds the values of the Barycentric co-ordinates u & v using the below equations (Joy, 1996),

$$u = (v1.v1 * v0.v2 - v0.v1 * v1.v2)/(v0.v0 * v1.v1 - v0.v1 * v0.v1)$$

$$v = (v0.v0 * v1.v2 - v0.v1 * v0.v2)/(v0.v0 * v1.v1 - v0.v1 * v0.v1)$$

Where,

$$v0 = C - A$$
$$v1 = B - A$$
$$v2 = P - A$$

A, B & C are the three vertices of the triangle and P is the point that is checked

The point lies in the given triangle only if all of the following conditions are satisfied,

$$u \geq 0 \ \ \&$$
$$v \geq 0 \ \ \&$$
$$u + v \leq 1$$

Algorithm: *Finding the face*

a) Get the ***last_hit_face*** of the agent

b) Check if the ***agent->position*** lies on the same face.

   If yes, return ***last_hit_face*** and break out of the loop

c) If not in ***last_hit_face***,

   Add the ***last_hit_face*** to the ***already_checked_faces*** list

   Check if the ***agent->position*** lies in ***last_hit_face->neighbour_faces***

   If yes, return ***hit_face*** and break out of the loop

d) If not in ***last_hit_face->neighbour_faces***

   Add the ***last_hit_face->neighbour_faces*** to the ***already_checked_faces*** list

   Search the neighbours of the

   last_hit_face->neighbour_faces - already_checked_faces

e) Recursively repeat the above step until the face is found

f) If found break out of the loop

   Set ***agent->last_hit_face*** = ***hit_face***

g) Else if ***total_number_faces*** = ***already_checked_faces***

   'Agent is not in the terrain'

Once the face is identified, using the three vertices of the triangular face an equation for that plane can be calculated

$$Ax + By + Cz + D = 0$$

Where (x, y, z) lie on the plane

Then, using the current position and the normal of the current agent an equation for a line can be formed.

$$P = P1 + u(P2 - P1)$$

Where $P1$ -> current position

$P2$ -> Normal of the agent

Solving both the equations, the point of intersection between the line and the plane can be found which is set as the new position for the agent.

As the agent moves along the terrain, its direction needs to be updated based on its terrain alignment. The direction is obtained by getting the angle between the agents' normal and the face normal and the axis of rotation. This rotation is applied to the current direction vector to get the new direction which is usually along the plane of the face.

4.3.3. *Herding*

A basic implementation of Reynold's flocking algorithm works well for a flock of birds or a school of fish which are free to move in a 3D space. However a herd of animals do not have that freedom as they are supposed to move along the terrain in addition to flocking. Also, the animals are supposed to head in a direction together in a natural way unlike birds or fishes while flocking change directions without any real target. Hence in herd it is important in which way the herd is headed at each frame. This will be the result of combination of other factors like obstacle avoidance, terrain following with the flocking rules.

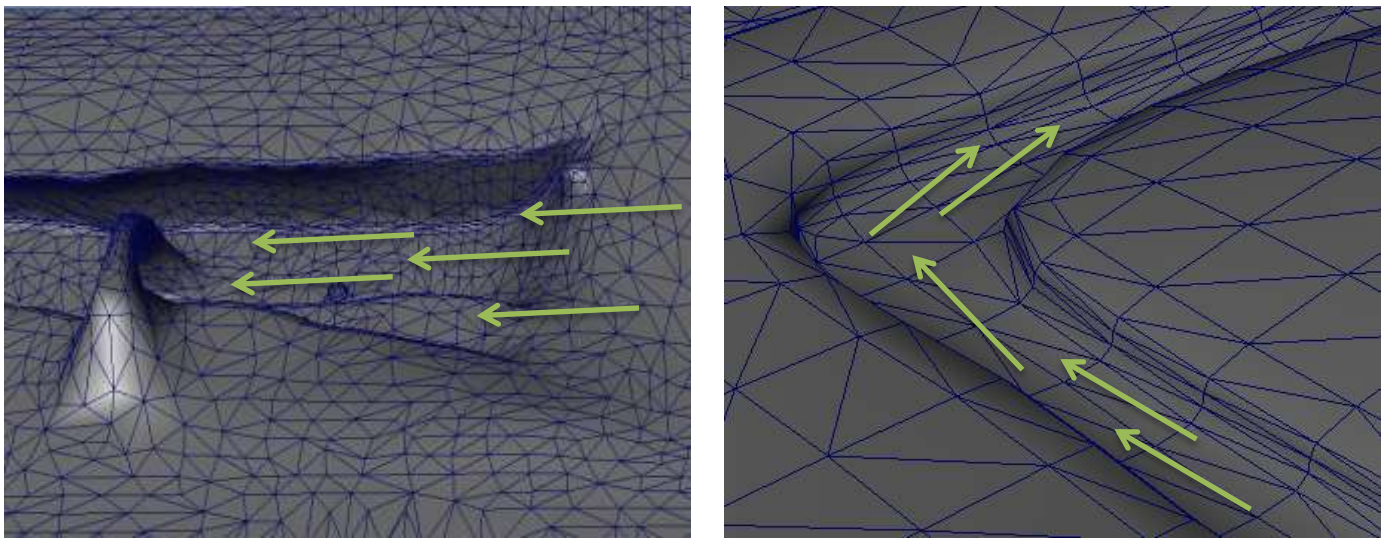


Figure 12: Heading direction of the herd

Initially, once the initial positions of the agents are set up by the Herd class a *std::vector* is created for each agent to store its neighbours. A visibility radius is provided to all the agents of the herd within which the agents choose their neighbours. Any agent outside this radius will not have any effect on the flocking properties of this agent.

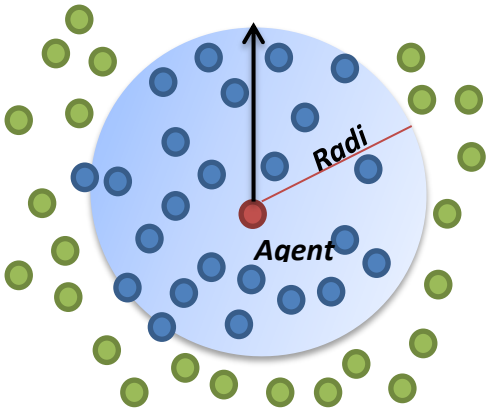


Figure 13: Viewing radius for the agents

The neighbour update is run for all the agents when they are created and later at each frame for each agent based on its new position calculated by the flocking methods. The flocking rules cohesion, alignment and separation are calculated as per the explanation in section 3.1. A sum of the three rules provides the resultant flocking vector.

$$flocking\ direction = cohesion + alignment + separation$$

This flocking direction is added with the current direction and the resultant is normalised forming the new direction of the agent.

$$new\ direction = current\ direction + flocking\ direction$$
$$agent \rightarrow direction = new\ direction$$
$$agent \rightarrow update\ neighbours$$

During the course of herding through the terrain, the agents may form different groups and those groups may not stick together due to the visibility radius. Even if the agents group up to smaller herds, the groups are supposed to move together for it to look like one huge stampede. In order to make them stay together a global cohesion is implemented where the average position of all the agents is calculated and added to the flocking vector. The vector is weighted based on how strong the cohesion is needed without changing the basic herding behaviour of the agents.

4.3.4. Obstacle Avoidance

In any crowd system, the agents are introduced into an environment where there are a lot of obstacles on their path. Obstacles can be of different types such as walls, pillars, rocks, other agents and so on depending on the type of environment. Since the environment here is a terrain, the obstacles are usually rocks, huge bumps or tiny hills on the surface. The agents are expected to look ahead and detect the presence of an obstacle. Once the obstacle is detected, they should be able to avoid it by going around or jumping over based on the size of the obstacle and the nature of the animal behaviour. In order to make them go around the obstacle, the steering direction (D) should be changed by deflecting it through an angle (theta). Before the start of simulation, the terrain faces are categorised into traversable and non-traversable based on their slope. This is determined by the angle between the face normal and the gravity vector. If that angle is too big then that face becomes non – traversable and is assigned a value 2 and all the neighbouring faces are set as 1. Using the current position and the direction of the agent, a “look ahead” position is determined. If that position is 1 then it is closer to a non-traversable face and hence is turned by a certain angle. If in case an agent hits a non-traversable face then it is turned by a 90 degree.

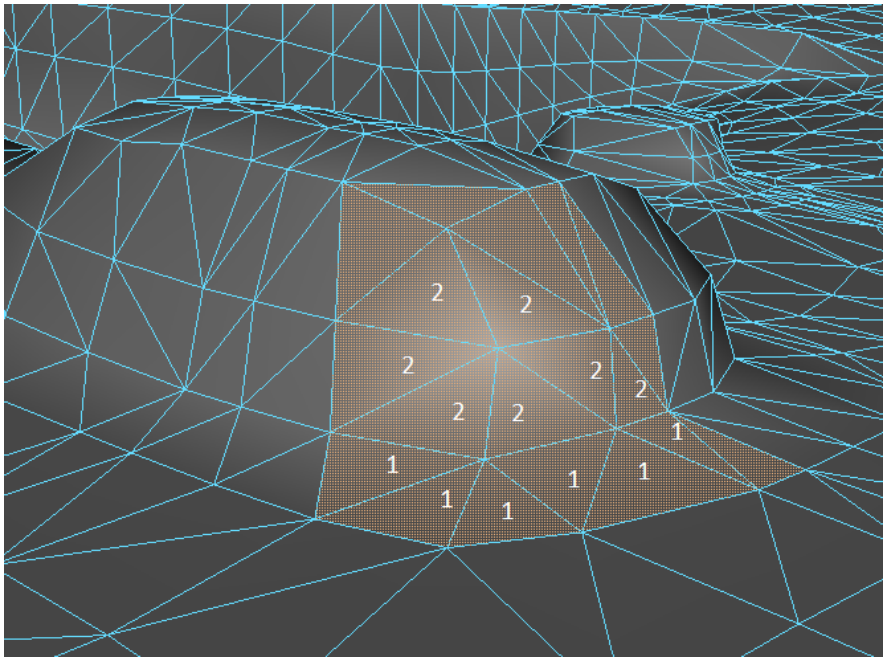


Figure 14: Faces with value 2 are non-traversable & ones with 1 are their neighbours

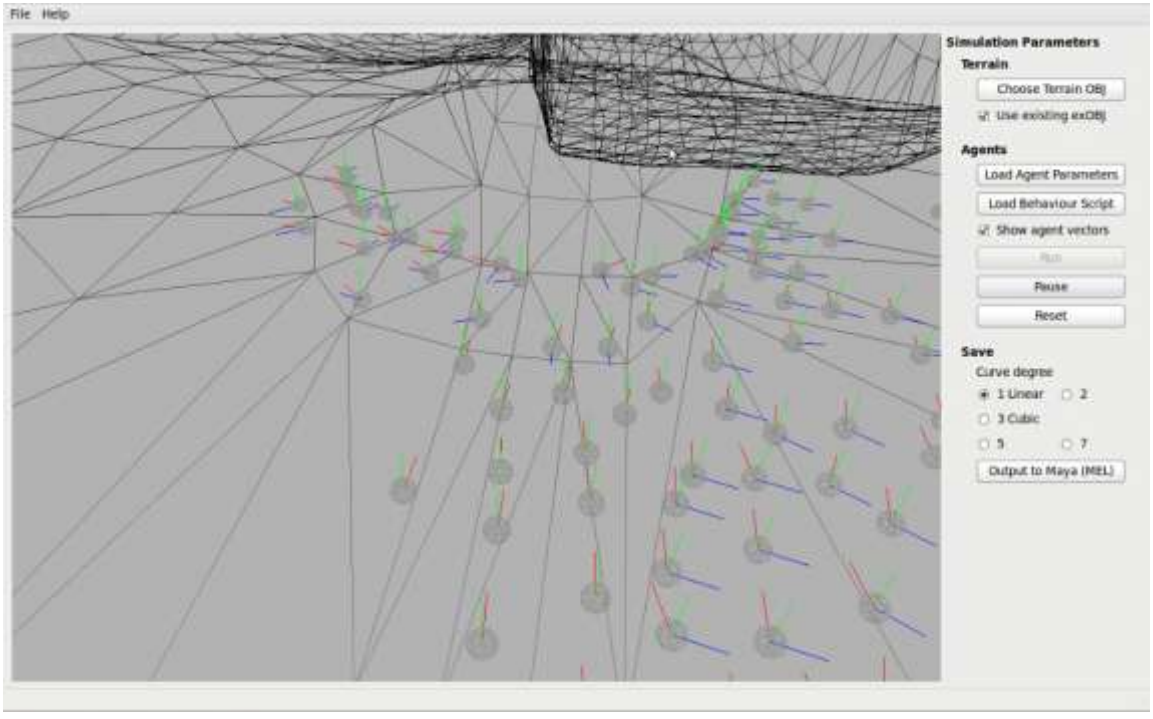


Figure 15: Agents avoiding obstacles and steering away

4.3.5. Drag force

Agents moving on an arbitrary terrain navigate at different speeds at different parts of the terrain based on the shape of the terrain at that position. Gravity has a significant effect on the agents when they are moving on hills. For instance, the speed of the agents when they run downhill will be different from that when they climb uphill. Each face will have a normal and an up vector which is always (0, 1, 0). The angle between this up vector and the normal gives the rotation angle which is calculated using the dot product of the two vectors. The cross product between the same two vectors gives the axis of rotation. Using this rotation, the drag vector is set to align with the face. This gives the direction in which the drag force will act on an agent. The magnitude of this is calculated by getting the dot product of the drag force vector and the gravity.

4.3.6. Python Integration

To add more flexibility to the application, python has been integrated with C++ code. While the main C++ part handles the collective herd behaviour, the python part is responsible for the individual behaviour especially in cases when there is more than one type of agent in the herd. Such agent-specific behaviours are written as a python file and that file is passed to the program where the agents are created. Here the python part also handles the position update using the current direction and speed of the agent. This way, new agent behaviours can be added to the simulation at any point just by adding the behaviour script file. To demonstrate this feature, a simple behaviour script is included with the program which makes the agents wander around with random directions on top of the basic herding behaviour. Similarly any kind of behaviour can be scripted in python and added to the program by passing the file through the UI.

Figure 16: Agents wandering due to the python script Wanderer.py

4.4. Maya visualisation

4.4.1. Exporting simulation data to Maya

Simulation once run is exported to Maya for adding models and animation. This is carried out by storing the positions of each agent at each frame in a vector list where every agent is defined by a *std::vector* of its positions from the start frame to the end. These positions combine to form one curve for each agent that would serve as the motion paths for the corresponding agents in Maya. The data from C++ is written out into a MEL file with a script that draws curves in Maya using the positions. This generated MEL script is executed in the Maya API which draws the curves for each agent in the herd. Any model is imported into Maya as an OBJ file. This animal geometry is attached to the motion paths using an automated python script which key frames the agent on to the corresponding positions which is calculated by the distance travelled and the overall distance. The resulting simulation is rendered in Maya with a basic lighting.

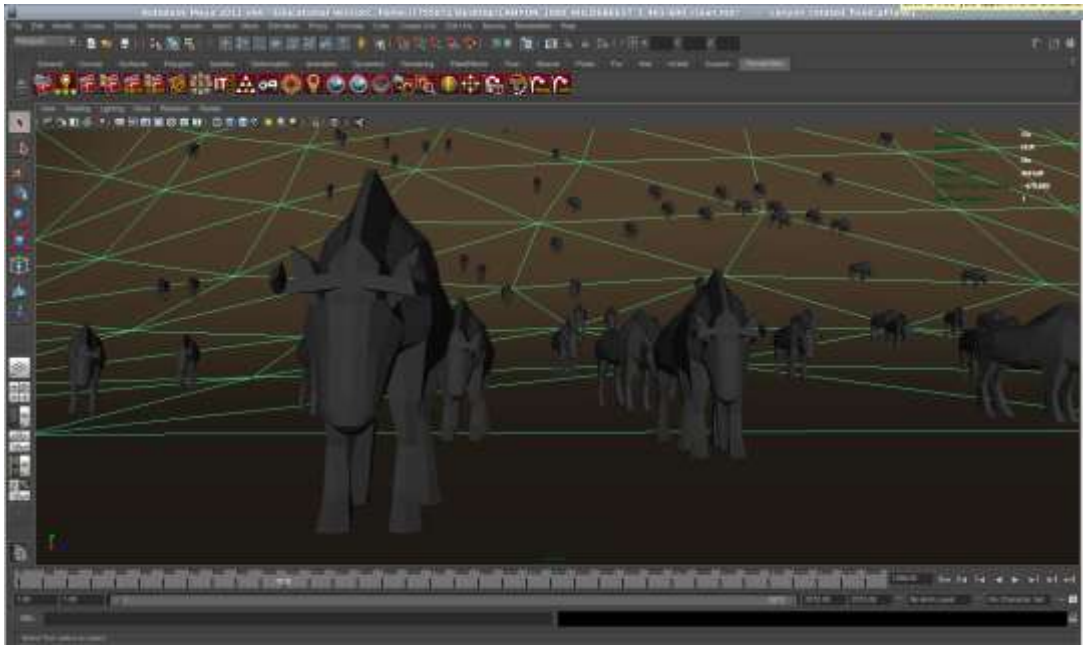


Figure 17: Maya Visualisation - Herd moving down the terrain

5. Results

The simulation works well with as a whole and the simulated agents that are capable of herding, following terrain, avoiding obstacles and also adjusting their speed based on the slope of the terrain. Most of the costly operations that are required for the above tasks are pre-processed and the values are read in from a text file. This has significantly improved the efficiency of the whole application. The program can comfortably handle number of agents that render 1000 at an average frame rate of 2 fps. But theoretically if can handle agents up to the size an integer if the required processing power is available on the machine used.

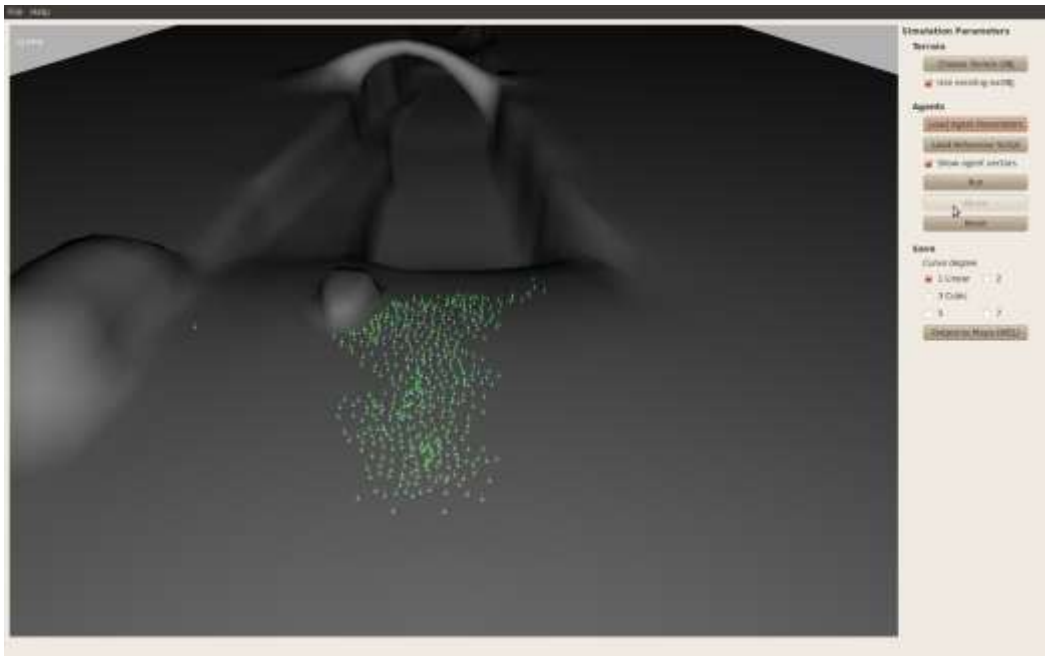


Figure 18: A basic herd behaviour

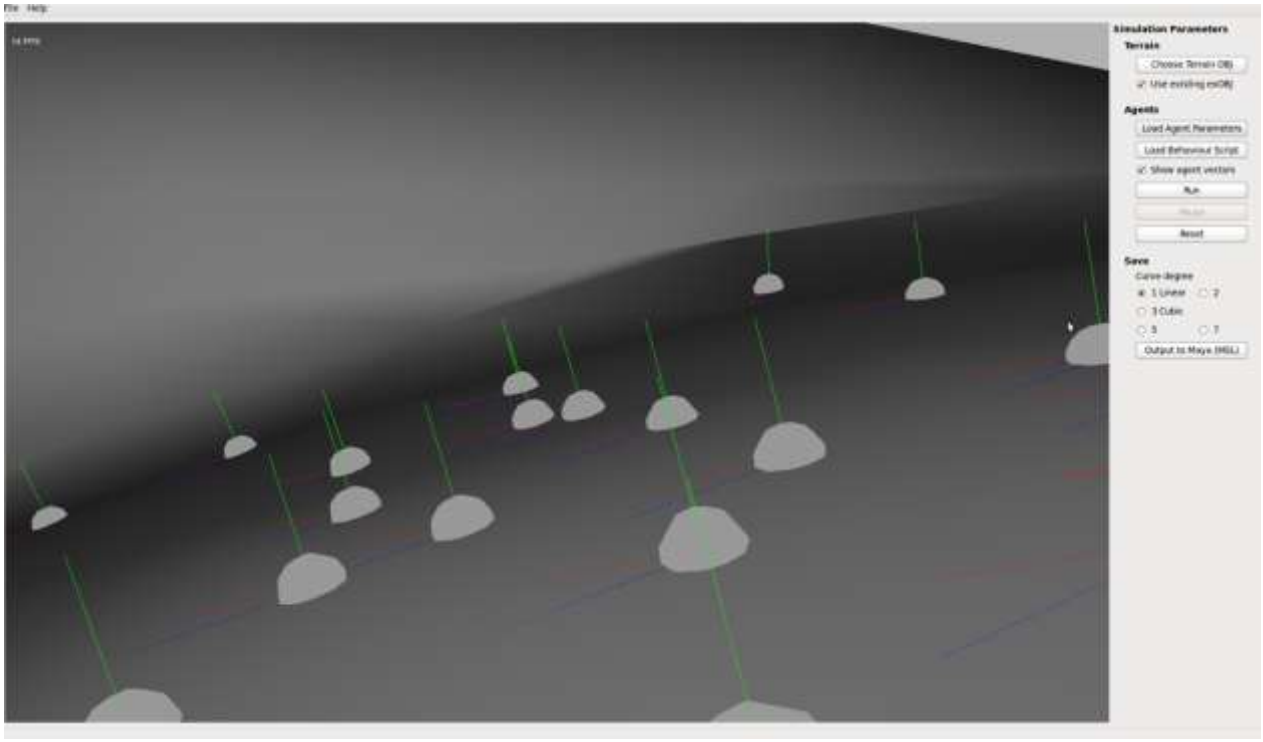


Figure 19: Agents following the terrain

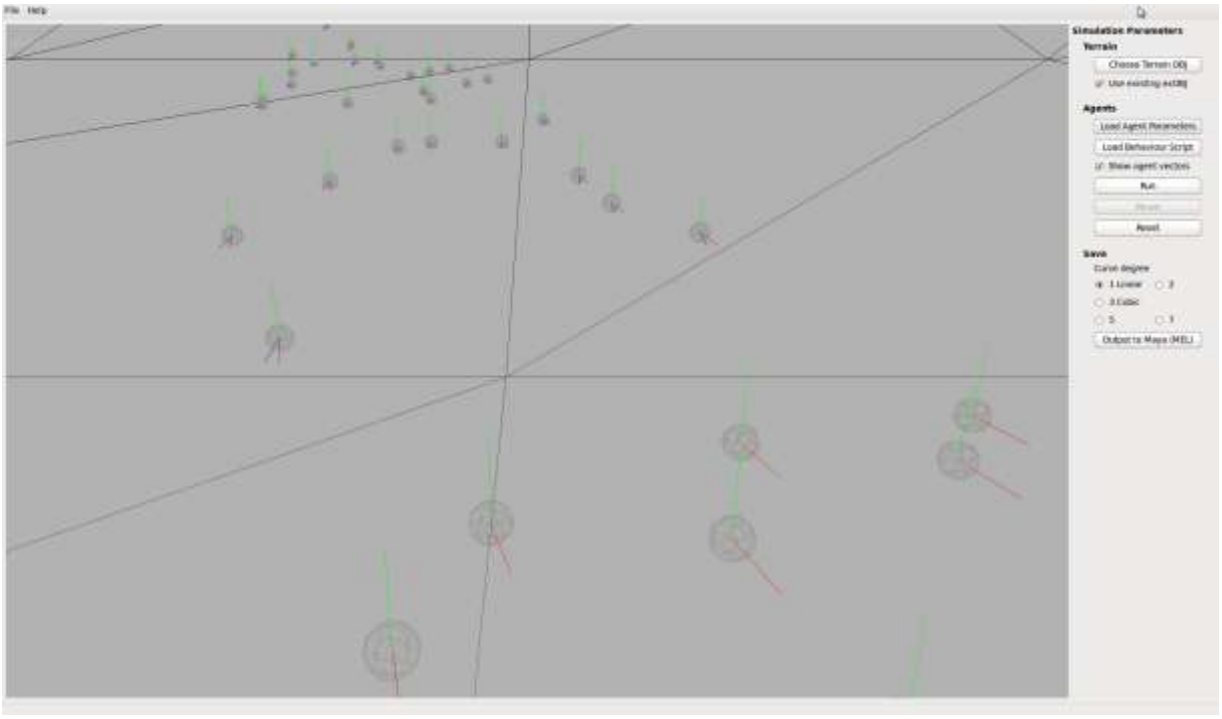


Figure 20: Agents showing their steering direction and up vector

6. Conclusion & Future Work

Thus this simulation program can be used to generate any type of flocking or herding behaviour with multiple agents and different behaviours by passing in the required attributes and script files. Considering the features and the flexibility of the program, the overall goal of the project has been satisfied to a good extent. More functionality can be added to the program in future such as incorporating artificial intelligence, avoiding moving obstacles and so on. The visualisation in Maya can be further improved by adding animation to the geometry whose motion graph is modified based on its movement in the herd and in relation to the environment.

References

[i] Reynolds, C. W., 1987. Flocks, herds and schools: A distributed behavioral model. In: SIGGRAPH '87 Proceedings of the 14th annual conference on Computer graphics and interactive techniques, July 1987, New York, USA

[ii] Gompert, J., 2003. *Flocking over 3D terrain*. Available from: http://digitalcommons.unl.edu/cgi/viewcontent.cgi?article=1127&context=csetechreports [Accessed Jun 2012]

[iii] Reynolds, C. W., 1997. *Steering behaviors for autonomous characters*. Available from: http://www.red3d.com/cwr/steer/ [Accessed Jun 2012]

[iv] Choi, M. G., Kim, M., et al. Deformable Motion: Squeezing into Cluttered Environments. *Eurographics*, 30(2011). Available from: http://mrl.snu.ac.kr/publications/deformable_motion.pdf [Accessed Jun 2012]

[v] Macey, J., 2012. *Massive documentation*. Available from: http://nccastaff.bournemouth.ac.uk/jmacey/Massive/Docs/learning/ [Accessed Jun 2012]

[vi] Ruas, T. L., Marietto, M. G. B., et al., 2011. *Modeling artificial life through multi-agent based simulation*. Available from: http://www.intechopen.com/books/multi-agent-systems-modeling-control-programming-simulations-and-applications/modeling-artificial-life-through-multi-agent-based-simulation [Accessed Jun 2012]

[vii] Rossmann, J., Hempe, N., et al 2009. A flexible model for real-time crowd simulation. In: *IEEE International Conference on Systems, Man, and Cybernetics*, Oct, 2009, San Antonio, TX, USA. Available from: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5346308 [Accessed Jun 2012]

[viii] Disney, 1994. *Wildebeest Stampede*. Image. Available from: http://lionking.wikia.com/wiki/Wildebeest [Accessed Jul 2012]

[ix] Wikipedia, 2012. *The Lion King*. Available from: http://en.wikipedia.org/wiki/The_Lion_King [Accessed Jul 2012]

[x] Joy, 1996. *Point in triangle* test. Available from: http://www.blackpawn.com/texts/pointinpoly/default.html [Accessed Aug 2012]