777 Interim Report: Sheep herding game

Alex Henriques[*]
Department of Computer Science
University of Auckland, Auckland, New Zealand

Abstract

The behaviour of sheep in a flock as they react to a dog is an interesting problem to understand and simulate. Various 2D Flash games simulate dog-sheep repulsion with simple response movements. Others model dog-sheep repulsion and flocking with spring forces. In this project we implement a 3D sheep herding game, aiming to simulate flocking and herding behaviour more realistically than currently available games. We also aim for our game to be enjoyable, and perhaps to teach players something about sheep herding.

1 Introduction and Motivation

The behaviour of sheep in a flock as they react to a dog – indeed the behaviour of any herd animal as it reacts to a predator – is an interesting problem to simulate. Does the sheep run, and if so in what direction? Does the sheep ignore the dog, confront the dog, or even attack the dog? How can the dog's behaviour and demeanour influence which action the sheep takes? How strong is the tendency and desire for each sheep to be close to other sheep? Do sheep tend to congregate in one flock, or multiple flocks as long as there are sufficient numbers? Would a sheep trying to get back to its flock charge right by a dog, or would it remain isolated in fear?

[*]e-mail: ahen045@ec.auckland.ac.nz

These are the problems we intend to investigate during this project. We will hopefully be able to make a fun game based on the sheep behaviour we discover. Everything discussed herein is my own work on the project.

2 Previous Work

Sheep herding is not exactly a popular game genre, but we have found some examples.

SheepGame is a simple 2 dimensional Flash game [3]. It moves the sheep directly away from the cursor by a small increment each timestep if the sheep is within a certain distance of the cursor. Three game modes are available: herding the sheep into a pen, splitting the flock into two separate pens, and guiding the sheep through an obstacle course.

The PCman's Arcade Sheep Herding Game is again a 2 dimensional Flash game [4]. In this game though instead of reacting to the cursor, sheep react to dogs. One or more dogs are attached to the cursor with a spring force. So when the user moves the mouse, the attached dogs are dragged along and oscillate back and forth around the cursor. It is not pure Hooke's law however, as the dogs seem to have a maximum speed. Dogs also have a slight repulsive force with respect to each other, to stop more than one dog bunching up in the same position. With a stationary cursor, four dogs reach jittery equilibrium about 5 metres away from each other. Sheep react to the dogs with a repulsive spring force, though again with a reasonably low maximum speed. Sheep also have a flocking tendency, so when no dogs are around the sheep tend to gather in a lump, with repulsive forces to prevent too much sheep-sheep collision. The object of the game is to herd sheep into a circular area on the game board.

Little shepherd is yet another 2 dimensional Flash game [1]. It is similar to *SheepGame*, but with some additional features. For example, in some levels there are obstacles to avoid such as water and cars on the road. Sheep tend to keep moving in the same direction even when the cursor is not around, which can be quite frustrating as they walk into hazards by themselves. There is no flocking tendency.

Doug Kavendek's flocking demo shows an interesting flocking algorithm for "boids" (bird-like objects) as they fly around obstacles to objectives [2]. We derived the basis of our flocking algorithms from here (see section 3.3), while adding significant improvements and tuning to adapt the method to sheep flocking.

3 Implementation

3.1 Terrain

The basic terrain engine that comes with Ogre automatically creates terrain based on three things.

- A *heightmap*. This is a greyscale bitmap. Each pixel represents a terrain vertex and its height, with 0 ground level and 255 the highest point. The bigger the bitmap, and the smaller terrain area it defines, the more detailed the terrain.

- A *terrain texture*. This is a coloured texture, mottled green for grass, or brown for dirt, etc. It is stretched over the whole terrain. As the texture size is generally much smaller than the whole terrain, the ground will look quite blurry with just a terrain texture applied. So the terrain texture is generally used for broad colour details of the terrain.

- A *detail texture*. This is generally stretched over only a small portion of the overall terrain, and tiled over the rest. If the tiling factor is set too low, the ground remains blurry; too high, and the repeating pattern is very visible. Generally quite a high factor is applied, to give the ground good detail. We looked in some real world games, and the repeating pattern is quite visible in the distance, though perhaps not so obvious as to be noticed when you don't know what you're looking for.

There are also more advanced terrain techniques supported by Ogre terrain plugins, for example *texture splatting*, which uses alpha maps to blend multiple detail textures over the terrain. We have not tried this yet but may investigate it later.

3.2 Grass

To create grass in Ogre, we follow this process (derived from the Ogre grass demo):

1. Create the grass mesh.
2. Create a static geometry that fits the desired terrain, and add to it as many instances of the grass mesh as desired.
3. Set the static geometry *region dimensions* and *rendering distance*.
4. Create a shader function for smooth fade in/out.

In step 1, the grass mesh we create is a simple mesh consisting of three planes at 60 degree angles to each other. A grass texture with alpha values is applied to each plane.

In step 2, we generate the grass positions in a grid like fashion, with random positional offsets. To get the correct y position for each grass, we cast a collision ray to the terrain under its (x, z) coordinate. After this step, the entire terrain is covered with grass. This can look good, but there are severe frame rate implications. The number of grasses rendered is proportional to the square of the far clip plane. With a distant clip plane, there can be tens of thousands of grasses to render, slowing things to a crawl. To solve this we would like to only render grass reasonably close to the camera, which is where the next step comes in.

Step 3 requires some background information. In Ogre, each *region* of a static geometry is sent as one rendering batch. Larger batches are generally more efficient to render, however this has to be balanced against the ability to *cull*. If an entire region is far away from the camera, it can be culled, i.e. not sent for rendering. Regions are culled by setting the static geometry *rendering distance*, so when the distance from the camera to the region is greater than the rendering distance, the region is culled. We found that a good rendering distance seemed to be the region size (length). So, when the camera is at the edge of one region, the adjacent region is visible, but the next region over is only visible when the camera crosses over into the adjacent region. After this step, only nearby regions of grass are rendered, solving the frame rate problems. However, these large square regions of grass are quite unseemly as they suddenly pop into view when we get near, and pop out of view when we leave (see Figure 2). This is where the next step comes in.

The motivation behind step 4 is that rather than have large square regions of grass suddenly pop into place as we wander around, we would like to see only and all the grass in a set circular region around the camera. A simple way to do this is to add a line the grass's vertex shader program,

```
colour.w = oPosition.z < 60 ? 1 : 0;
```

where `colour.w` is the vertex's alpha transparency, `oPosition` is the object position, and `60` is the radius of the clipping distance. This gets us what we wanted: we only see the grass in a 60 metre radius around the camera. But, it is still quite unsightly when every individual grass suddenly pops into view at the 60 metre edge. Instead a gradual fading into view would be better. We tried

```
colour.w = saturate (1.0 - (oPosition.z / 100));
```

where saturate clamps to the $[0 \ldots 1]$ interval. This worked better, but not brilliantly – at 50 metres when the grass is still large enough to be quite visible, it had an alpha of 0.5 and looked slightly deformed. We tried increasing the radius above 100, but a smooth alpha function from the camera to the maximum radius seemed wrong. After some experimentation, currently the best function seems to be something like,

```
colour.w = saturate (2.0 * (1.0 - (oPosition.z / 100)));
```

This results in an alpha of 1.0 up to 50 metres, which smoothly drops to 0.0 at 100 metres. Further tweaking is probably necessary to get the best looking results.

3.2.1 Wind effects

We did some experimenting with wind effects, i.e. swaying grass. The simple vertex shader Ogre example for swaying grass assumed each grass had a base y position of 0. But in our program grass y height is not constant – it depends on the terrain height. We need to investigate how to pass each grass's base y height into the vertex shader program.

3.2.2 Grass density

There are different "styles" of grass one can use. Sparse grass meshes, mixed perhaps with bushes and flowers, tend to add some visual plausibility to the environment. Grasses in this case are perhaps taken as being of a different, taller growing breed than the surrounding ground. Another style is to cover the ground completely with grass, such that the grass meshes are taken as being the actual grass all over the ground. We tried this style (see Figure 1) and rendering speed was fine. But memory usage and CPU time to create the meshes was unacceptably high. With terrain 1km by 1km, one grass every square metre results in a million grasses. Clearly current games with full grass cover use a different method – we intend to investigate which.

One idea that might allow denser grass is to create only 9 regions of grass. As the player crosses a region border, the regions behind him are moved in front, keeping the player in the centre region. For each region change, this involves modifying the position of three regions, then updating the y value of each grass mesh in those positions. Suppose grass visibility and region length is 100m, with one grass mesh per square metre. That makes 10000 meshes per region, or 90000 overall. Filling 1.5km by 1.5km of terrain at the same density would give 2.25 million meshes, so it is a big improvement. However because the 90000 meshes move as the player does, they cannot be as optimized as static geometry and rendering speed takes a big hit.

Another option is to use 9 regions as above, but when the player crosses regions, create three new static geometries of grasses in the correct position. This would speed up rendering time while still requiring only 90000 grass meshes at any one time. We have not yet tested this, but building the static geometries would probably introduce notable lag at each region cross.

3.3 Sheep behaviour

There are three main steps determining a sheep's behaviour (call her Dolly). This discussion is based on the implementation as of the time of writing.

1. Calculate the individual influences acting on Dolly. These include the dog, flocking attraction, close proximity repulsion from other sheep, goals (e.g. green grass), anti-goals (e.g. an evil forest), etc.

2. Combine the individual influences into an overall force $\mathbf{f}$. The simplest method is to calculate a weighted sum of all individual influences.

3. Update Dolly's current velocity using $\mathbf{f}$.

Most of step 1 is reasonably straightforward, though some influences are more difficult to calculate than others. The dog's influence is a force linearly decaying to 0 beyond a certain distance. The flocking attraction was initially a force capped beyond e.g. 50m, and decaying to 0 at 10m. These values can be also dependent on flock size. Close proximity repulsion has been implemented in naïve $O(n^2)$ fashion, which does not harm performance much with only around 100 sheep. For better scaling, some form of spatial hashing or binning would be necessary. Goal calculation can vary, but is generally a constant force acting toward the goal.

The most difficult influence to get right so far is the flocking attraction. If influence decays to 0 at a certain radius r, sheep tend to quite unrealistically form a ring of radius r facing inwards. If on the other hand influence decays to 0 at the flock centre, sheep all try to go to the centre, resulting in a maximally dense flock with lots of jittery repulsion behaviour. One solution is to always keep sheep moving, so they are never able to make patterns around a particular point. A better solution, and the current implementation, is to leave Dolly satisfied if a certain number of other sheep are nearby. For example, if 30% of the sheep are within 30 metres of Dolly, she feels no flocking attraction. If fewer than 30% of the sheep are nearby, she is attracted to the centre of the flock. This results in much more natural behaviour, and depending on the % of sheep specified, can allow stable separation into multiple subflocks. Again we use the simple $O(n^2)$ method with negligible performance impact.

Step 2 is simple. The main difficulty here is tweaking weights such that for example Dolly does not run right through the dog toward a small patch of green grass (even if she's hungry).

Step 3 is more difficult. The method described in *Boids* calculates the angle θ between $\mathbf{v}$ and $\mathbf{f}$, and turns Dolly by some portion of θ each timestep, keeping speed constant [2]. This produces a nice effect of more rapid turning when θ is large. Our sheep have variable speed however, so we require something more complex.

To simplify things, if we turn Dolly as above (by a fraction of θ each timestep), we just need to reconcile $||\mathbf{f}||$ with $||\mathbf{v}||$. The first difficulty here is that $||\mathbf{v}||$ is dependent on Dolly's maximum speed, which the AI module should not be basing its calculation of $\mathbf{f}$ on. One solution might be to ensure $0 \leq ||\mathbf{f}|| \leq 1$, with $||\mathbf{f}||$ signaling the "urgency" with which the force is applied. 0.0 would mean no urgency and decelerate the sheep to 0.0 speed, while 1.0 would accelerate the sheep to maximum speed. We implemented this, and found it worked well.

The main difficulty now is tuning parameters to produce plausible results. For example flocking attraction–if to within too small a distance–can bunch up the flock. Combined with large repulsion radii, this can also result in some jittery, hyperactive sheep.

4 Results

Figures 1- 2 show some of the results so far.

5 Conclusion

In this project so far we have implemented a basic sheep-herding game. The player controls a dog and runs around in first or third person, while sheep react in real time. Sheep react to the dog with a repulsive force, and a flocking tendency attracts sheep to each other.

The terrain is generated by a heightmap, a terrain texture and a detail texture. There is also grass which fades in and out in a set radius around the camera.

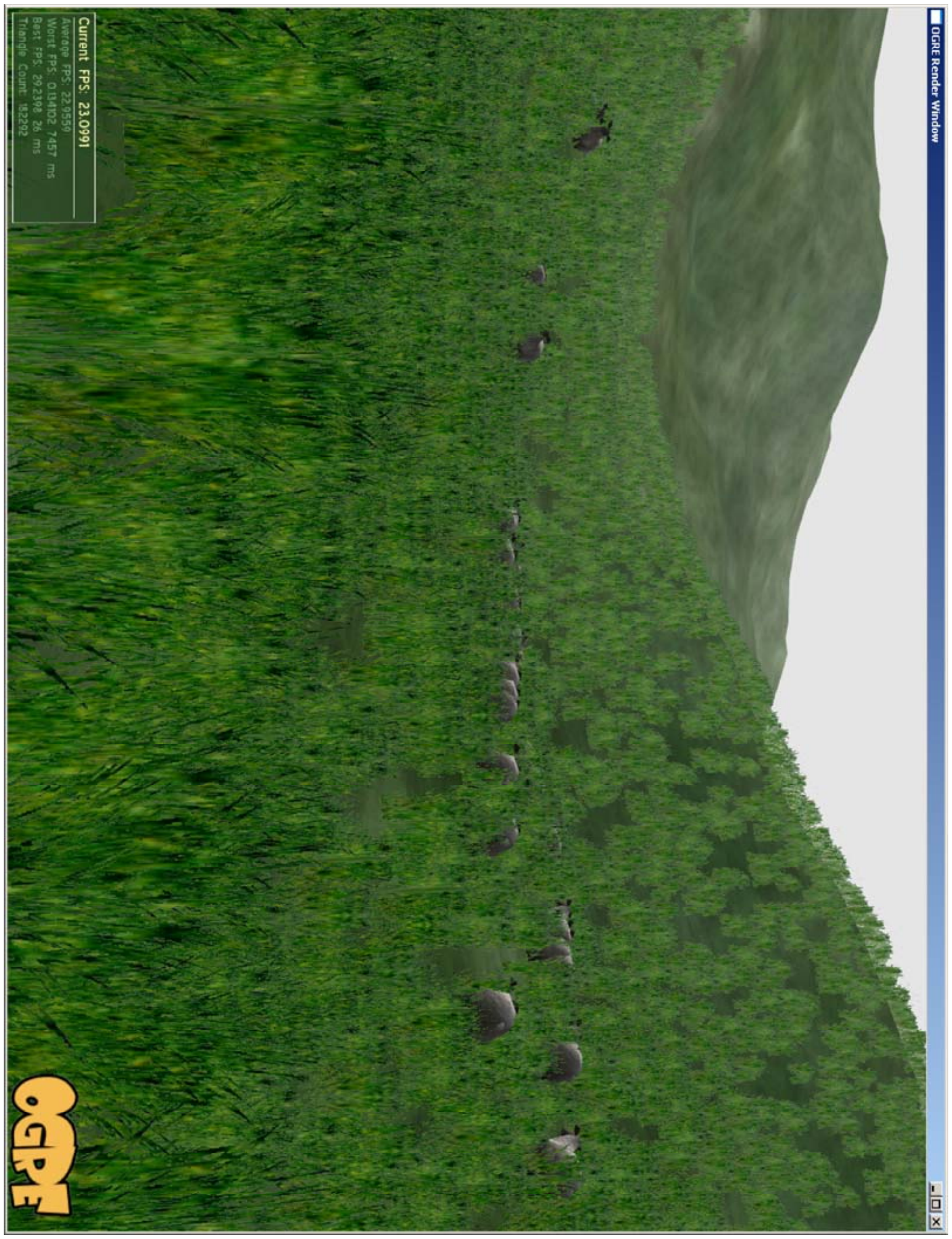
6 Goals for rest of project

We have several goals for the rest of the project.

- Create a few different, unique levels with their own terrain, obstacles and objectives.
- Improve look of grass, and experiment with better grass fade in/out techniques.
- Add some trees.
- Improve sheep behaviour, e.g. flocking and response to the dog.
- Implement "special powers" for the dog: e.g. barking and biting.
- Add our own textured and animated sheep and dog models.

References

[1] FreeOnlineGames.com. Little shepherd, 2006. [Online; accessed 13-September-2006].



```
Current FPS: 23.0991
Average FPS: 22.9559
Worst FPS: 0.134102 7457 ms
Best FPS: 29.2398 26 ms
Triangle Count: 182292
```

Figure 1: A dense, nearly full-coverage grass style.

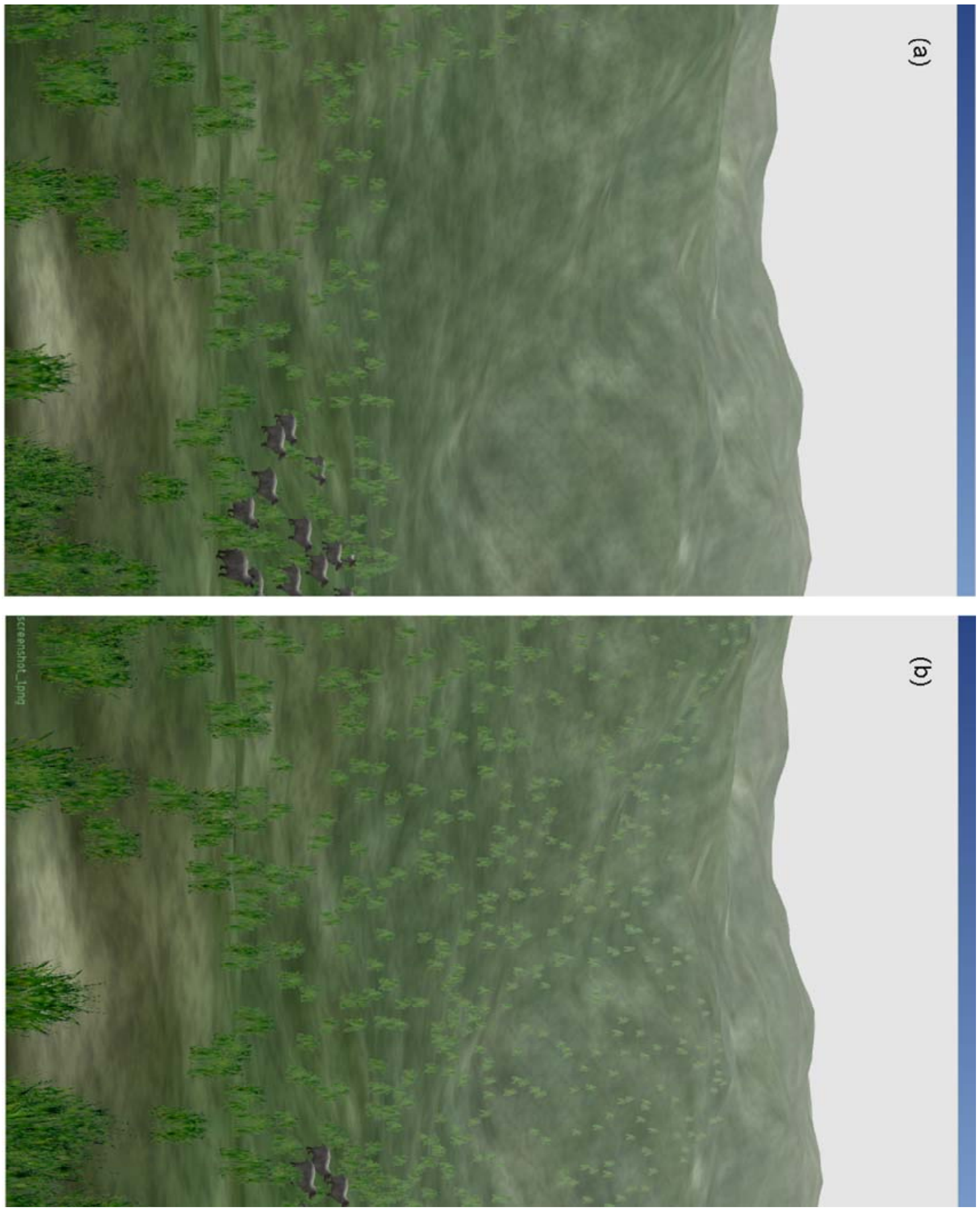


Figure 2: In (a), a region of grass has been culled. In (b), the user steps forward, and the region of grass pops into view.

[2] Doug Kavendek. Boid flocking, 2006. [Online; accessed 13-September-2006].

[3] David Lewis. SheepGame, 2006. [Online; accessed 13-September-2006].

[4] Time Tripper. The PCman's Arcade Sheep Herding Game, 2006. [Online; accessed 13-September-2006].

Learning Objectives

3D modeling

- Skeleton animation
- Texturing
- Modeling
- Exporting into Ogre compatible format (integration with Ogre)

AI

- Intelligent agents
- Perform decisions based on environment info
- Attractive and Repulsive forces
- Genetic algorithms for population generation
- Character motion
- Flocking algorithms

Ogre Libraries

- Terrain generation
    - Grass
    - Height maps
- Camera
- Environment effects
    - Animated skybox
    - Weather

Main References

Makino Kohji and Matsuo Yoshiki. Control of shape and internal movement of a homogeneous autonomous mobile robot herd employing simple virtual interactive forces. In *SICE 2003 Annual Conference*, volume 3, pages 2912–2915, 2003.

Jyh-Ming Lien, O.B. Bayazit, R.T. Sowell, S. Rodriguez, and N.M Amato. Shepherding behaviors. In *Robotics and Automation*, 2004. Proceedings. ICRA ’04. 2004 IEEE International Conference, volume 4, pages 4159–4164, 2004.

Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *SIGGRAPH ’87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 25–34, New York, NY, USA, 1987. ACM Press.