The goal of the project is to create a realistic simulation of a number of Zombies and see what influences them to swarm. Everyone is more or less familiar with the idea that Zombies exhibit some kind of swarming behaviour at least superficially similar to soldier ants. Our simulation must obviously have internal logical consistency. Our simulation should as much as possible have a believable logical consistency

1 Types of Zombie

J. Pearse

The background material on Zombies is broadly split into two types of explanations for the existence of zombie humans;

Supernatural phenomenon are responsible for reanimating human corpses.[1] These animated corpse zombies are created with black magic and respond to the wishes of their creator. This type of Zombie is therefore not acting entirely independently and of little use in our simulation. Similarly the Zombie of Haitian Voodoo mythology, although not actually dead are enslaved to a Houngan sorcerer.

Biological albeit fantastic explanations, where the existence of zombies is the result of some type of virus or parasite, details vary from fiction to fiction but by picking and choosing between these ideas we can construct a Zombie conceit which is at the very least logically coherent.

1.1 Observations

- Zombies may be able to react differently to injury than non infected humans, ignoring pain. But a fatal wound, massive blood loss or damage to vital organs will eventually the kill the host.
- Zombies are attracted by sound and movement but are somehow deterred from biting already symptomatic individuals, in many accounts the Zombies use scent to search out prey.

1.2 Conclusion

The hypothesis of our simulation is that the attraction to movement and sound is what causes the Zombies to group together in hordes and this is what our simulation is designed to explore. The path-finding abilities of the zombies are very limited. Since zombies are not noted for their self-preservation instincts their motivations can be broken down as;

1. I can bite someone $\rightarrow$ Bite them.
2. I can detect someone to bite $\rightarrow$ Move towards them.
3. I can detect a suggestion of someone to bite $\rightarrow$ Move towards the stimulus.

[1] White Zombie (1932), The Evil Dead (1981)

2 Requirements Specification

J. Mitchard

2.1 High Level Requirements

We would like to create a program that simulates a possible zombie apocalypse using crowd-swarming algorithms in a concurrent manner. We aim for this to be realistic as far as we can make something that is still science-fiction realistic, based on research of the topic in science fiction and realistic biological effects considered. This simulation application would allow the user to change different parameters, such as changing the density of the populated area or the speed in which the Zombies can move, that would affect the way the simulation ran.

The software should be able to display a visual representation of a humans and zombies swarming together, attempting to survive. For each type of character, Zombie or Human, this will be achieved differently. Though different by nature, these two different characters will share similar qualities, or states, such as being hungry. However, they will differ in that Humans become tired, whereas a zombie uses less brain power and internal systems and act for longer period of time.

In terms of the intelligence of these two different entities, the human entities will be considerably more intelligent when it comes to things like finding the route of best fit to a location, or navigating obstacles. On the other hand, a zombie would just try and get from A to B without thinking about it in a particularly logistical or strategic manner. This should be represented within the simulation. We would like this application to be cross platform, though until further research has been carried out we will not limit ourselves to defining how this is achieved. It has been considered that we display the visualisation of the simulation through the means of a web browser or through a Java like system that uses a virtual machine.

2.2 Low Level Requirements

Based on the inherent lack of graphical apis with the majority of concurrent languages, we have considered that a decoupled client-server approach could be the best way in which to run the simulation. For example, carrying out the decision making and work in a concurrent language and sending this to a front end client written in a different language.

A message passing model of concurrency is going to be more efficient for our concept to be modelled than something like a thread based system, as quick communication between the individual processes will be a central point of our system. Certain requirements we have made about the languages used are that the language;

- Does it's own memory management
- Is efficient,in context to it's role, to a reasonable level

'Visual Representation Client'

- Has access to a well documented,complete 2D graphical animation and window management library.

'Simulation'

- Is concurrent
- Can deal with a large amount of separate processes
- uses message passing

3 Client Requirements Specification

J. Mitchard,R. Hales

3.1 The Client

The client is the user facing part of our system, and as such it will need to not only clearly represent the information that the application simulates, but also look appropriate and aesthetically pleasing.

3.2 What We Need

This first section will explain the parts of the client that we feel are most critical to the workings of our system.

We need the client to:

- visually represents the current state of the simulation,
- display the continuous changing states within the system efficiently,
- be cross platform,
  - this includes looking and functioning the same across different systems,
- allow the user to start, reset and pause the simulation,
- allow the user to control certain parameters of the simulation.

3.3 What We Hope to Achieve

We would like the client to:

- visually represents the current state of the simulation in an aesthetically pleasing manner,
  - we would like the visualisation to display as much information as possible, without overcomplicating what is shown to the user,
- seamlessly display the continuous changing states within the system efficiently,
- provide intuitive controls for the user.

4 Coding Standards

`R. Hales`

The rules listed in this document are to be used as a guide during code production for our project. This is to ensure the code produced is consistent and of a good quality.

4.1 Versions

When milestones are reached within the code, the version number should be incremented to make this clear. The version number should reflect which iteration of the project the code is from as well as which version within that iteration. A changelog should also be kept to make clear the differences between each version.

4.2 Naming Conventions

All Modules, Functions, Atoms,etc. should be given a short (1-2 words) name that is descriptive of it's purpose to improve the readability of the code.

4.3 Formatting

As far as possible all formatting should be kept internally consistent. This includes bracket placement, indentation distance and messages.

4.4 Comments

All Functions,States, etc. should be clearly commented to explain their purpose.

4.5 OTP

The OTP framework should be used for erlang code where applicable.

4.6 Compiling

All code should compile successfully without warnings or errors.

5 Risks

`R Hales`

5.1 Deadlines not met

- If deadlines are not met then work on the project may be slowed or halted until the work is caught up on.
- If a deadline is not met the work should be completed as soon as possible, work plans and milestones should be reconsidered accounting for the setback and the reason for the failure to hit the deadline should be discussed.

5.2 Group members unavailable due to illness or injury

- If a group member is not available due to health issues work can fall behind and organising planning and meetings can become harder.
- The members work should be reallocated if they are unable to work, milestones and deadlines should be re-assessed, if there is a serious problem the member should seek concessions. If the member can't meet the group to discuss this issue discussions should be held online/via phone/etc.

5.3 Work is lost due to technical

- If files are lost then the work contained on those files will need redoing.
- To avoid this happening all work should have at least one back up copy that can be restored after a fault.

5.4 Concurrent simulation is not feasible

- If we can't design a concurrent simulation then we will need a different way of designing the simulation.
- We should have a backup non-concurrent design for the simulation we can implement in case our original design will not work.

6 Languages research

J. Mitchard

In order to achieve our goal of creating this simulation, we will need two separate systems. One that is written in a concurrent language that will act as a message switchboard and run the algorithms that control the intelligence behind the nodes in the the simulation, and a second that has a good, well documented, but not too heavy 2D animation library that we can display a visual representation of the simulation.

6.1 Simulation Languages

6.1.1 Go(Golang)

Though, in terms of programming languages, Go is very young, this has quickly become a popular concurrent language because of its similarities, in coding style, to C and because of its open source nature. This would be a good language to use for the simulation because of its asynchronous nature, but none of the members of the project have used it before so it would be a big risk to the completion of the project if it took longer to learn than expected.

6.1.2 Java

Java is an Object Orientated Programming language, so it may seem strange to include it in a list of concurrent languages. However, Java is a very adaptable language and through the use of Threads it is possible to synchronise multiple tasks so that they happen concurrently. For the scope of our project, this is not ideal however, as it doesnt allow proper communication between the threads.

6.1.3 Erlang

Erlang is a functional and concurrent programming language that deals very well with a high load of processes. Concurrency is handled through message passing between individual processes, which makes for very flexible and dynamic concurrency designs possible in simple manner. This is the language we have decided on using for this part of our project because the members of the project group are all competent Erlang programmers and we feel that this language meets our requirements.

6.2 Visualisation Languages

6.2.1 Java

Java meets most of our requirements quite well here; it is cross platform through the use of its virtual machine, we all know it and it has access to a lot of different graphics and animation languages. However, most of these libraries are very heavy duty and may be a little overkill for what is needed for the program.

6.2.2 Javascript(and the D3 Library

Being a web-based scripting language, Javascript has the inherent advantage of being totally cross platform as all modern browsers can run Javascript. On top of this is includes a lot of individual libraries that allow for animation, object control and networking abilities. This makes it a very strong candidate for our project as it meets all our requirements and is a language that we all have a lot of experience in using.

D3 is a Javascript library that allows you to manipulate documents based on data. It is lightweight and is entirely cross platform as it is web-based and follows strict compatibility with web standards. We have decided to use Javascript, and other web based markup languages such as HTML and CSS to build our visual client as we feel that it is the most effective means to achieve our goal.

6.2.3 Python

Similar to Java, Python is primarily an Object Orientated Programming language. Its open sourced and community based ethos has led to an abundance of very good graphical and animation libraries. One library in particular that we have looked into is calle Pygame, whilst this library is very well documented and maintained, it is far more than we really need for this program. Another problem is that Python is not crossplatform in the same way that the previous two languages are, however compilers for all widely used platforms are available.

6.3 Conclusion

So, now it has been decided that we will use an Erlang based server that will deal with all simulation handling and process management, and will be represented in Javascript to create a cross platform system that negates the problem of having to install it onto multiple operating systems.

7 Intelligence Research

R. Hales

One of the most important decisions we had to make about the simulation was exactly how intelligent to make the human and zombie entities.

7.1 Factors in determining intelligence

The two biggest factors involved in deciding on an intelligence level for the entities, realism and time. Realism is how similar the simulated entities act compared to their real versions or in the case of zombies, a realistic model. Ideally our simulation should be as realistic as possible, but there is nearly no limit to how much attention to detail can go into making the entities realistic which is why we have to take into account our time restraint. Because of this we decided to create two models for intelligence, a base model that is intelligent enough to satisfactorily complete the project, and an advanced model that contains more features we would like to include but may not have time for.

7.2 Zombie Base Model

The zombie entities must be aware of their surroundings, be lightly attracted(ie move towards) other zombies within their surroundings and strongly attracted to humans within their surroundings. They also need to have collision detection to avoid clustering into one spot rather than forming a horde.

7.3 Zombie Advanced Model

As zombies are typically not shown to be intelligent, we actually intend to decrease the zombies intelligence if there is enough time. In particular we plan to decrease the zombies ability to detect collision when they are chasing humans, this will allow zombies to get stuck on obstacles while trying to catch people as often happens in zombie fiction.

7.4 Human Base Model

The human base model will be a modified version of the zombie base model. They must be aware of their surroundings, be lightly attracted to other humans and strongly repelled(ie move away from) zombies,they also need collision detection.

7.5 Human Advanced Model

If there is enough time , we plan to improve the path-finding of the human entities. As Intelligence is usually one of humanities biggest advantages against zombies we intend to make humans more capable of navigating obstacles, allowing them the possibility of outsmarting the zombies.

8 Updated Client Requirements Specification

J. Mitchard,R. Hales

8.1 The Web Client

Now we have decided to implement the client using Javascript and D3, we can make our requirements specifications more specific to the target platform.

8.2 What We Need

This first section will explain the parts of the client that we feel are most critical to the workings of our system.

We need the client to:

- visually represents the current state of the simulation,
- display the continuous changing states within the system efficiently,
- be cross platform,
  - this includes looking and functioning the same across different browsers,
  - this includes avoiding browser specific tools and API's,
  - this also means avoiding system specific API's such as Adobe Flash and Microsoft Silverlight that work differently or are not available on certain Operating Systems.
- allow the user to start, reset and pause the simulation,
- allow the user to control certain parameters of the simulation,
- meet current web standards.

8.3 What We Hope to Achieve

We would like the client to:

- visually represents the current state of the simulation in an aesthetically pleasing manner,
  - we would like the visualisation to display as much information as possible, without overcomplicating what is shown to the user,
  - because this is going to be a web based system, we think that scalability and efficiency are a higher priority than a graphically intensive visualisation,
  - include a visual indication of current actions, including it's current attraction or object it's avoiding,
  - show a visual representation of family units within human entities,
  - provide a visual representation direction of entities,
- seamlessly display the continuous changing states within the system efficiently,
- provide intuitive controls for the user,
- provide the user with option to change how the data is represented, for example:
  - heat maps for the grid,
  - graphs showing population over time,
  - normalised population graphs,

9 The evolution of concurrent system design

J. Pearse

This document provides a description of the process for developing a coherent and effective strategy for using concurrency to simulate a spatial environment, providing data required for navigation to the entities within it. Rather than a more traditional technique of maintaining a canonical model of each entities position in space and repeatedly dividing it to provide relevant data relative to position, we wanted to use smaller concurrent regions of space each maintaining it's own data internally.

9.1 Initial design

Our original (naive) idea for the system was to use concurrent processes to represent each position in the system, if a position in space was occupied; a process would be created to represent it. By modelling space as a set of occupied points we intended to keep the system overhead low. As we explored the system it became apparent that we were in fact doubling the number the required processes; one entity and an additional position for each entity. As this early system

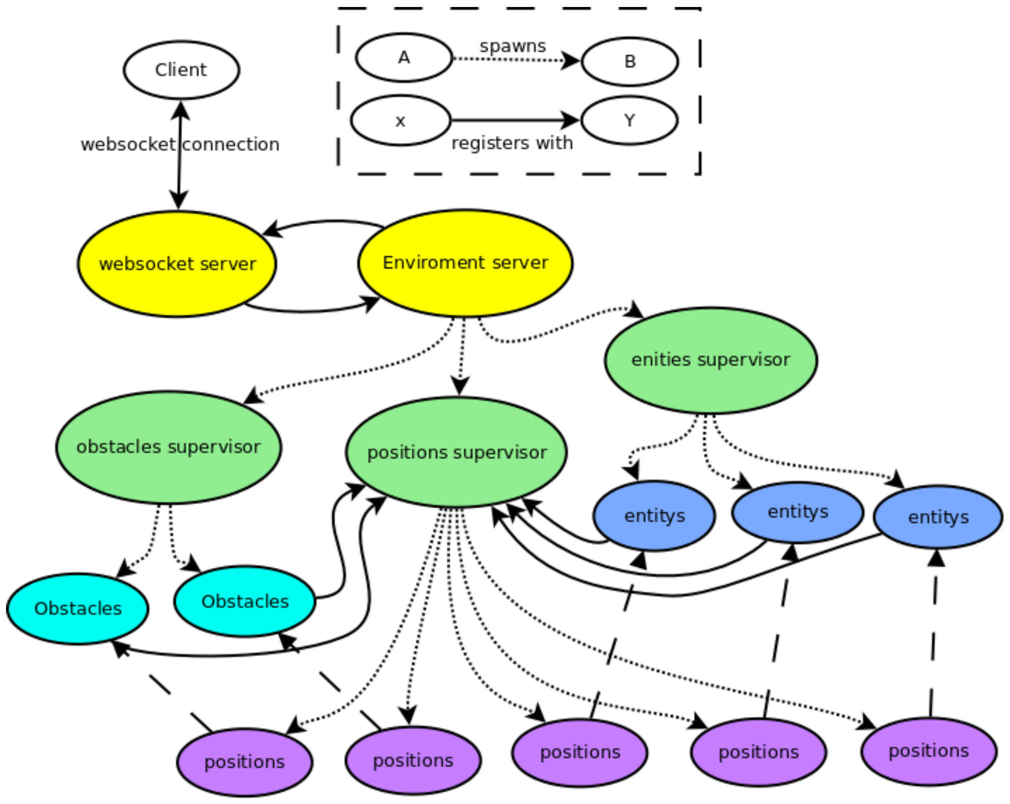


Figure 1: Initial, obsolete, system design

diagram shows, we had planned for positions (represented as some kind of co-ordinate data) to be separate processes which would be generic and used by different ”classes” of entities. Thankfully we revised this design further before we began to implement such a system. The group would like to thank Dr F Barnes, for his input in our system revision. Dr Barnes, remarked that our design was similar to an early iteration of a concurrent flocking program he had worked on and provided us with a description of the system used by his program which greatly influenced our subsequent revisions.

9.2 The Tile-Viewer model

Our revised design employed a system of concurrent tiles used to model space, we abandoned the idea of positions-as-processes, each tile would keep a list of the positions of entities on it and each entity would keep a record of it's own position and the address of the tile it currently occupies. Additional viewer process would be created to contain lists of cross-tile data, allowing entities to "see" into neighbouring tiles. An entity requests it's sensory data from its tile's corresponding viewer and a tile updates all its neighbouring viewers with the positions of entities it contains. This duplication of data allows us to maintain a high level of concurrency in the simulation, reducing potential message-passing bottle necks. Each tile has its own corresponding viewer, but this viewer additionally contains the data of neighbouring tiles which report to it. The viewer corresponding to Tile:A (which would be Viewer:A) contains all the

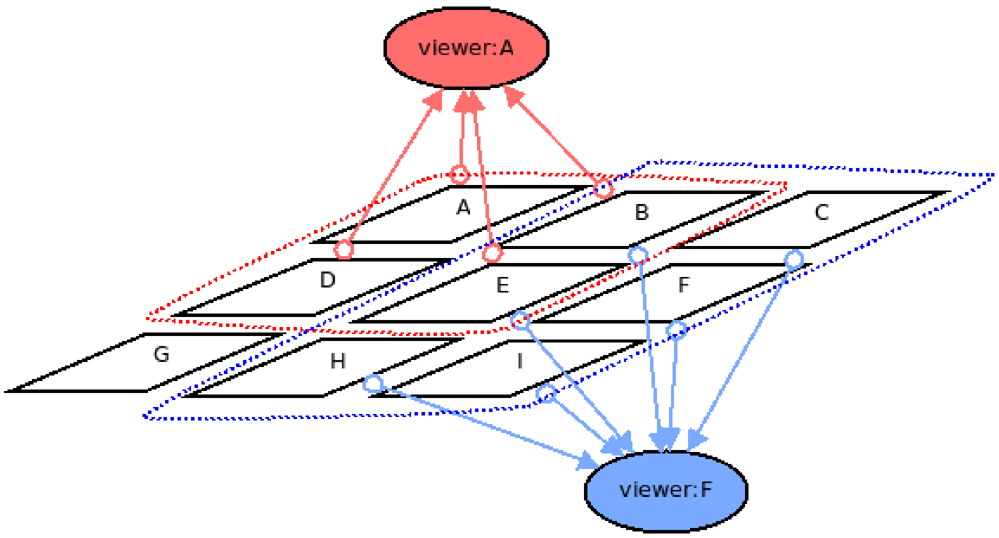


Figure 2: Tile-Viewer System

data from tiles A,B,D and E. When an entity on Tile:A needs information about its surroundings it calls on Viewer:A. This system allows an entity in the bottom-left corner of Tile:A to be aware of the contents of Tile:E. Viewer:F duplicates the data from Tiles B and E but entities calling on Viewer:F never receive data about the contents of Tile:A.

When an entity decides to move a negotiation takes place between the Tile and the entity; details of this negotiation can be found in section 18.3 but are summarised below.

1. Entity informs its Tile of its preferred move
2. Tile informs entity of new position and which tile corresponds to that position.

crucially the tile computes the final position of the entity, this system allows the tile to handle collision. The potential for two entities to move simultaneously into the same space is avoided. This is in-keeping with our design goal of realism.

Our final system architecture (fig:3) also follows the general Open Telecom Platform (OTP) pattern using a supervisor-process hierarchy, further details of this OTP design pattern can be found in Section 16.1

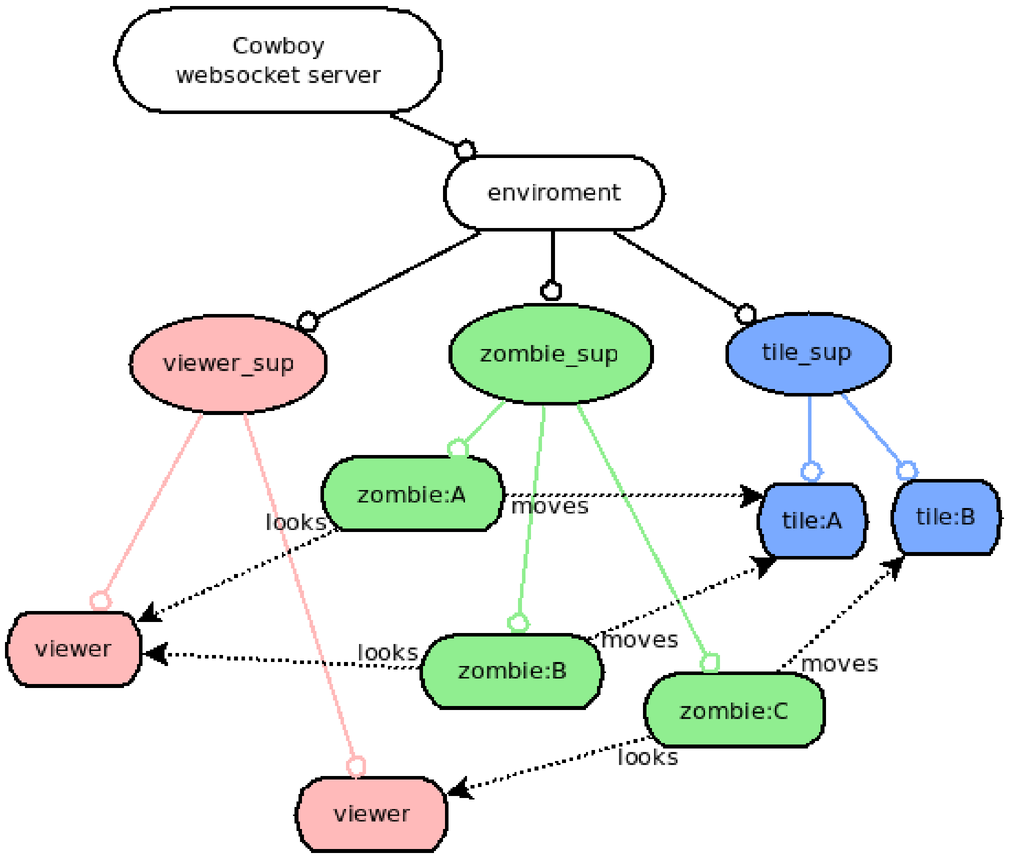


Figure 3: Zombie, Tile-Viewer Relationship

10 Phase 1 Development Evaluation

J. Pearse

After the lengthy research phase the whole team were keen to begin programming, this first development cycle was conceived as something of a experimental/prototyping exercise and a chance to get to grips with the Open Telecommunications Protocol (OTP). We distributed the development of the modules specified in the first draft among the team members. Throughout the two week iteration it became clear that we had underestimated the high level of dependencies among the modules especially with regard to integration, but by maintaining a good level of contact we were able to exceed our target for this iteration in all but one area. This success despite the difficulty of learning to satisfy the constraints of OTP development left the team in positive spirits with good morale going into the next phase. The knock-on effect of the unexpected programming difficulty was that the code QA fell short of our intended targets with methods uncommented and messy source files, in response we have decided to take a short break from coding at the end of this cycle and do a mini documentation and code clean-up iteration before we begin the next cycle.

Target	**Status**	**Notes**
Environment server	Complete	Additional OTP supervisor module also created
Tile Server	Complete	OTP supervisor
Viewer Server	Complete	OTP supervisor
Zombie State Machine	Unfinished	OTP supervisor present

As well as the unexpected supervisor modules we also implemented;

- A basic but fuctional websocket server,
- a JSON library for Erlang researched, compared and integrated,
- a basic JavaScript Client implemented with the D3.js data visualisation library.

Although we had hoped to have a visualisation of Zombies moving moving around at the end of this iteration, given the difficulty we were pleased to have them visualised, albeit static.

11 System Architecture for Iteration One

J. Mitchard

In order to begin writing the modules for the system, an effective design was needed to allow for all the neccesary communication to take place in the most effective means possible. This was something that had to be carried out before anything else could take place, as it would define how the system would work from the ground up.

11.1 Design

The system we had initially intended to design was considerably less complicated than the one we have at present, and took a number of attempts to get a working solution. The architecture we have designed splits the system into a number of different sections, each having a process parent called a Supervisor in Erlang. The sections are as follows:

- Environment - This is intended to act as the communications switchboard between all of the other processes, that allows a lot of the neccesary control over the system. This handles calls to and from the websocket, and provides what control the system offers over the processes and their supervisors.
- Tile - This module controls the grid that the entities will navigate. Providing means of communicating between the Environment module and the population of the tile.
- Viewer - This is looked upon as an overseer of the population, designed around the grid system that we have implemented with the Tile Module. Each viewer can see into a surrounding neighbourhood of tiles, and as such knows about the processes within. An example of this is depicted here: 2
- Zombie - This module controls the functioning of the Zombie entities, particularly the desicion making and inteligence of them.
- Human - As with the Zombie module, the Human module controls the desicion making of the Humans, though this will be considerably more complicated than the Zombie moule.

12 Process Evaluation

R. Hales

12.1 Iteration One

For our first coding iteration we chose to split the workload based on modules, so that each person would be able to individually work on their modules without having to rely on the others progress and allowing work to be done when we were not able to meet up. This process proved to have several problems. One of the major issues was the fact that when it was time to connect the modules together a fair amount of code had to be changed or replaced due to the modules being incompatible or because of unforeseen bugs. The other major problem that became clear is that not all of the group fully understood how every module worked due to not being involved in the coding process for some of the modules, this compounded with the first issue making it even more difficult to link the modules together.

12.2 Iteration Two

For the second iteration we re-evaluated our coding method and decided it would be best if we ensured the majority of our programming was done as a group,utilising pair programming. Having all code programmed in the presence of and with the input of the whole group, helped solve the problems from iteration one,making it a lot easier to make sure all the modules are compatible and allowing all the group to understand how the code works. The other reason for this change was, due to the nature of the concurrent system, the code for the system had started to become too interlinked for individual modules to be changed without it affecting the entirety of the system. If we had carried on attempting to work on modules individually we would likely have ended up with multiple incompatible versions of every module which would then require more time and work to make compatible again. We believed it would be more efficient to try to keep multiple versions of the code to a minimum to prevent incompatibility.

13 Prototyping movement using Particle swarm optimisation

J. Pearse

By treating each zombie in a swarm as an individual particle and implementing Particle Swarm Optimization (PSO) we can test that neighbourhood awareness is functioning correctly. PSO is relatively simple and requires most of the features which can later be used to implement a flocking algorithm. The way in which space is divided up by the Tile-Viewer system (see section:9.2) means the algorithm functions as though there are multiple overlapping particle swarms, the only things which had to be implemented that are not useful in flocking is the fitness function and storing the previous fittest position in the zombie state. For this we used a simple distance measurement to a human, the 'optimisation' is to try and reduce the mean distance to the human across the swarm.

13.1 Awareness of neighbours

A zombie gets a list from the viewer associated with the current tile containing all the positions of entities on neighbouring tiles, this list is first sorted by distance then truncated to remove all entities outside the zombies sensory range. The resulting list is an Erlang map with the entity type as the key, allowing efficient retrieval of all the target entitys first. For the PSO, target entity's are of fitness zero (there is zero distance between a target and itself). If no target is visible the next lowest fitness individual becomes the focus of the optimisation. The zombie uses the (clamped) difference between the fittest neighbour and its own previous fittest position as its new position.

13.2 Usefulness

Implementing this simple PSO allowed us to uncover and remove bugs from the neighbourhood awareness system and prove that it was functioning as expected, we were forced to make the state of entities much more generic and update the system of state reporting, to incorporate state messages with an arbitrary number of variables. All of the required trigonometry and improvements to the tile-viewer message system will be invaluable to the next stage of development.

13.3 Conclusion

The PSO prototyping implementation was successful in it's aim of testing the system with a reasonably simple movement algorithm [2], it took a week to implement and almost all of it is usable going forward. We didn't fix those bugs which were deemed specific to PSO fitness and velocity as the entire PSO fitness function is to be discarded in the next iteration. Trigonometric functions were placed into a separate module as these will be reused later as were the functions for the PSO fitness optimisations. We enjoyed watching the results of the PSO running as the zombie particle swarm gave a believable impression of coordinated movement. There's nothing intrinsically wrong with using PSO to provide the swarm movement, but the emergent behaviour is lacking. The ability of an individual zombie to be 'aware' of the fitness of it's neighbours is questionable in our stated aim of realism, combining this with the simplistic nature of PSO we have evaluated that a better option would be a flocking algorithm.

[2]The specific function used to calculate movement in our PSO was $V = IV_c + R_i(B - C) + R_j(T - C)$ where V is the velocity along a given vector, I is inertia, V is the new velocity, V_c is the current velocity R_x is a random number such that $R_x \leq 1$, B is the previous recorded fittest position ,C is the current position and T is the targets position.

14 The Boids Model

J. Mitchard

After deciding that Particle Swarm Optimisation was not the most appropriate means of achieving our goal, we decided to implement an algorithm based on Boids.

14.1 Boids

14.1.1 What are Boids?

The Boids model is an example of an individual-based flocking algorithm used to simulate the behaviour of large numbers of autonomous entities. First designed in 1986 by Craig Reynolds[3], the model incorporates three simple steering behaviours in order to flock to other boids near itself.

14.1.2 The Boids Model

The Boids Model defines a number of behaviours that allow for different entities to flock together.

- Steering behaviours
  - *Separation* - ensures that the entities will not overcrowd local flockmates. This prevents the flock converging on one entity and getting stuck in a localised swarm.
  - *Alignment* - allows the flock to maintain a heading, by aligning other members of the flock to navigate to an average heading of nearby entities.
  - *Cohesion* - the opposite of separation, this allows the independent entities to flock together by navigating towards other nearby entities.
- Advanced navigation
  - *Collision Avoidance* - ensuring that the members of the flock do not bounce into other entities or obstacles. This is the ability to change heading instead of hitting an obstacle.
- Attractors and Repulsors
  - *Flock Attraction* - an attraction to entities of similar type. This will be used by the Cohesion and Alignment steering behaviours.
  - *Super Attractors* - other entities considered higher priority to flock towards.
  - *Super Repulsors* - other entities considered a 'threat', flocks will prioritise moving away from these.

14.2 The Boids Algorithm

This model will need some adapting to suit our own simulation. The model is intended to simulate the flocking of entities, however our project incorporates two separate types of entities that will flock for different reasons and have distinct behavioural rules separating them from each other.

14.2.1 Boids In Context

Zombies and humans will need to flock in different ways and for different reasons. We have decided that zombies are not inherently intelligent, and would tend to 'flock' through being attracted to movement and noise of other zombies. In a sense, a horde of zombies will flock together wandering reasonably aimlessly until a human entity enters the area. At this point the horde would shift behaviour and attempt to chase the human(s) that are in range.

This type of relation defines that a human, to a zombies is a Super Attractor. Super Attractors are prioritised higher than other behaviours, such as collision avoidance and flock coherency, therefore allowing our horde of zombies to

[3]http://www.red3d.com/cwr/boids/

demonstrate the type of behaviour displayed by zombies in popular culture of getting stuck against obstacles in an attempt to reach the human by the shortest path possible.

In order to simulate a realistic situation of zombies and humans, this would mean that the zombies must be a form of Super Repulsor to a human; meaning that they will attempt to run away from the zombies. Humans, however, are inherently more intelligent than the basic instinct driven zombies that we are implementing. Whilst trying to escape, they will not just pick a direction and run. Whilst the aim for the human entities would be to escape the horde of zombies chasing it, collision avoidance must still be carried out whilst pathfinding an escape route.

15 System Testing

J. Pearse

The behaviour of the simulation was verified and tested using a combination of the diagnostic tools integrated into the visualisation, diagnostic functions exported from Erlang modules and prototyping (see section:13). Entity behaviours arising from interactions were observed and investigated (section:18). The highly concurrent nature of the simulation presents some particular challenges.

15.1 Simulation reporting

The major task of compiling a snap-shot of the system state at any given time is handled by the *report* function of the *environment* module (see section: 16.2) this is the function called by the client to update the visualisation, seen in section 17. The function builds a list of entity data gathered sequentially from each tile. Tiles keep details of the states of entities on them so the compiled list contains a snap-shot of all entity states across all tiles. However some time will elapse between the time the environment calls one tile and another.

1. environment calls Tile A and adds its state to the report, entities on Tile B are still moving
2. environment calls Tile B and concatenates this with the report from tile A, entities on Tile A meanwhile may have moved

Matters are further complicated by the independent states of entities, before an entity moves it updates its state by calling the viewer to examine its surroundings, then it calls the tile with its updated position (see section:18.3, particularly fig:12) - in the meantime other entities will have called the tile with their updated positions. Any given report submitted from the environment consists of many entity states, captured at slightly differing times. Although the report has such discrepancies in the data, the simulation itself does not rely on the report data for any function.

15.2 Visualisation

Every object drawn in the visualisation uses data from the WebSocket report which is very helpful in understanding the state of the simulation. The SVG format uses a coordinate system with the $\{0, 0\}$ origin in the top-left corner, this same system is used in the simulation internal co-ordinates facilitating the direct mapping of simulation positions to visualisation elements. The position values are multiplied with a magnification value for the visualisation, set by the Grid Scale drop-down - here set at 9 as we can see arbitrary attributes (such as y_vel) can be assigned to each element of the SVG. In fig:5 We see the visualisation in its initial state, the system has not started yet, the entity state is largely empty. In the JavaScript console we can examine the SVG element corresponding to the selected entity, we see the X and Y values are equal to the ones in the entity state multiplied by the grid scale value above. There is animated 'tweening' between reports which sometimes causes the appearance of entities cutting corners in the animation. The speed of individual entities is clearly observable in the visualiser allowing us to see the effect of energy-loss slowing them down gradually and then speeding up again when they gain more energy.

15.3 Inspection panel

The inspection panel displays elements from the state of any entities selected in the visualiser. By pausing the simulation and clicking on an entity, the process id of the entity can be determined and used to call any exported functions from the Erlang console. The data-binding methods provided by the Angular.js library such as *ng-repeat* allow an html element to be displayed for each item in a JavaScript array, when an entity is clicked in the visualisation it is added to a *display_list* array, which is simply iterated over and the contents of each element-state is printed in the inspector. As the inspector panel is dynamically updated the length of the lists of visible entities grows and shrinks causing the entire visualisation to jump erratically up and down the page, the inspector panel can be hidden to stabilise the visualisation.

15.4 Line of sight

A line connecting every entity in both the human and zombie lists to the selected entity is drawn into the SVG, (fig:4 the interesting aspect of the LOS visualisation is the way it reveals the concurrency inherent in the system, lines are

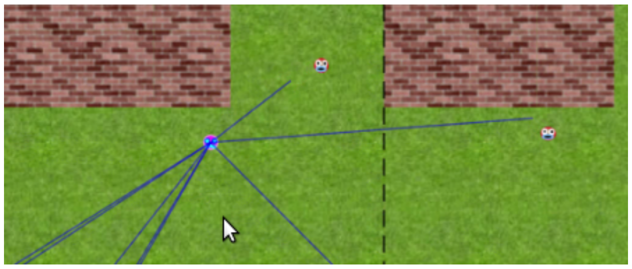
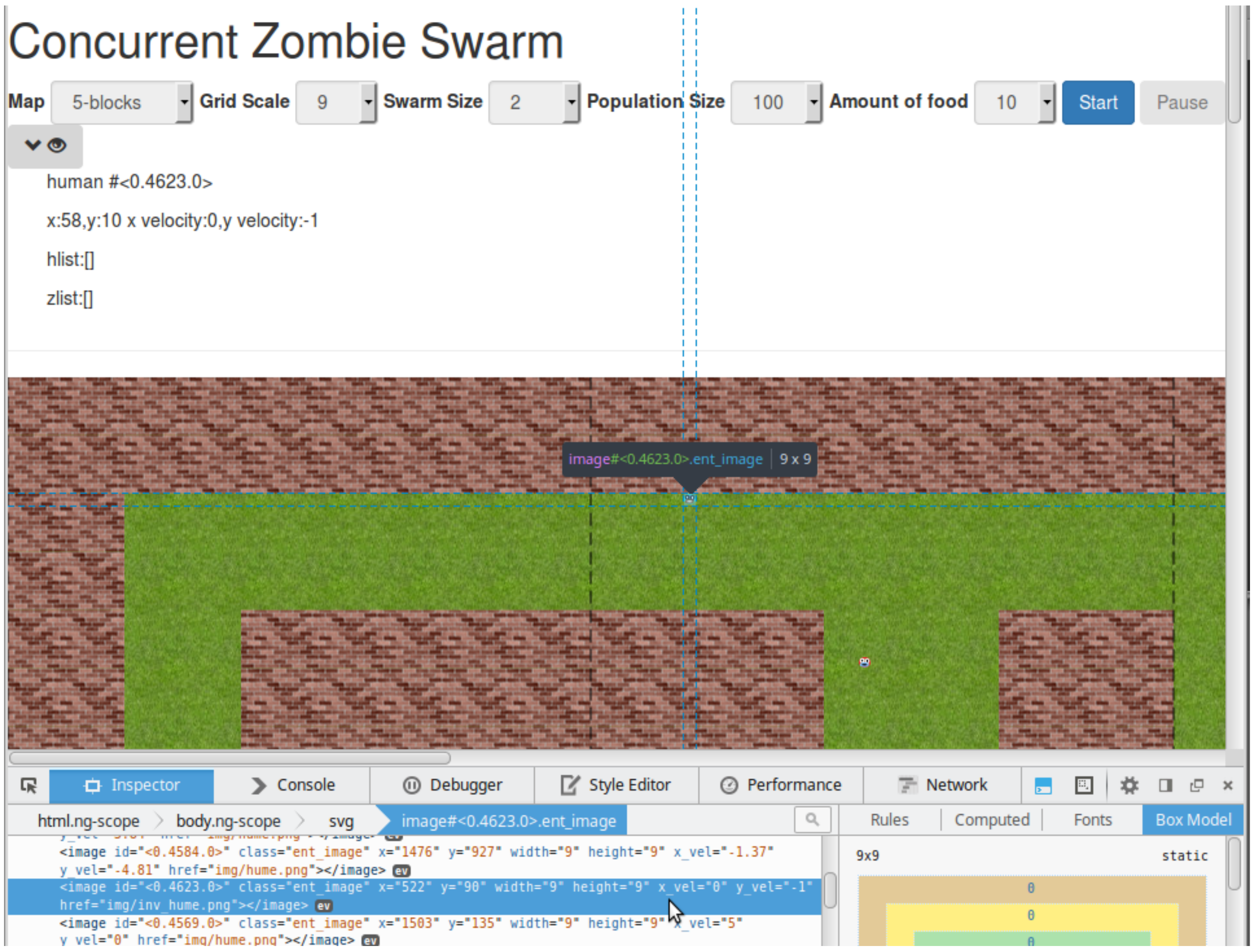


Figure 4: Concurrency visualised

not always drawn to the exact position where entities are, because lines are drawn based on the selected entities state they are sometimes drawn to the position of their target in the previous or next report cycle although this gives the appearance that the visualisation is out of sync, it's a natural effect of trying to visualise a highly concurrent system in discrete snapshots. The fact that lines can be drawn to the position target entities will be in, at the next step of the visualisation gives us evidence that the target entity in question has moved since the report was taken from that tile, i.e. the viewer and hence entity state, have updated during the construction of the report. The entity is drawn at the position reported by the tile when it was called. The subject entity however, is already aware of the targets new location, as the neighbouring tile has updated the viewer. Consider the following sequence of events:

1. tileA and tileB are neighbours so tileA reports its contents to viewerA and vice versa.
2. The environment collects a report from tileA, this will be used by the visualisation to draw entity positions and states of entities on this tile.
3. A subject on tileA updates its position, causing the viewer of tileB to be updated.
4. The environment collects report data from tileB where entity states may now reflect the updated contents of tileA.

As we can see, although all states were correct at the time of reporting, there is a discrepancy.



Concurrent Zombie Swarm

Map: 5-blocks · Grid Scale: 9 · Swarm Size: 2 · Population Size: 100 · Amount of food: 10

human #<0.4623.0>

x:58,y:10 x velocity:0,y velocity:-1

hlist:[]

zlist:[]

```
<image id="<0.4584.0>" class="ent_image" x="1476" y="927" width="9" height="9" x_vel="-1.37"
y_vel="-4.81" href="img/hume.png"></image> ev
<image id="<0.4623.0>" class="ent_image" x="522" y="90" width="9" height="9" x_vel="0" y_vel="-1"
href="img/inv_hume.png"></image> ev
<image id="<0.4569.0>" class="ent_image" x="1503" y="135" width="9" height="9" x_vel="5"
y_vel="0" href="img/hume.png"></image> ev
```

Figure 5: Co-ordinate mapping

16 System Overview

J. Mitchard

16.1 Supervisor-Child Architecture

The swarmer application has been implemented in a hierarchical process structure using Erlang Supervisor processes[4] that comes as part of the OTP[5] [6] (Open Telecom Platform). This means that all processes are children of a supervisor process within the applications' supervision tree. Supervisors are responsible for the stopping, starting and restarting of their child processes. The swarmer application has been implemented in a hierarchical process structure using Erlang Supervisor processes[7], a behaviour that comes as part of the OTP[8] [9] (Open Telecom Platform). This means that all processes are children of a supervisor process within the applications' supervision tree. Supervisors are responsible for the stopping, starting and restarting of their child processes. Swarmer has one main supervisor, called the swarm_sup, this is responsible for the tile_sup, viewer_sup, human_sup, zombie_sup, supplies_sup supervisor processes and the environment process.

Apart from swarm_sup, all of the supervisors are created with a simple-one-for-one restart strategy, meaning that each of the children will be identical processes using the same code, and should be restarted if they crash. swarm_sup is using a one-for-one strategy, which can restart its child processes without affecting the others[10].

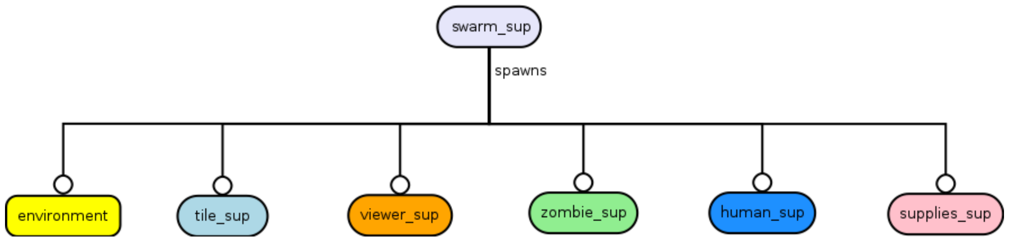


Figure 6: Supervision Tree for Swarmer

16.1.1 OTP Behaviours

Using an Erlang and OTP application with a process supervision architecture provides a standard set of interface functions, behaviours and more advanced error tracing and reporting functionality.

The backbone of swarmer is created using the gen_server[11] (Generic Server) behaviour. gen_server provides a framework for reliable and robust message passing between processes, using either synchronous requests called calls, or asynchronous request called a casts. The environment, tile, viewer and supplies modules have been implemented with gen_server behaviours.

The behaviour of the human and zombie entities in the system are modelled around the gen_fsm[12] (Generic Finite State Machine) behaviour. gen_fsm provides a state machine for the entities to use, and incorporates synchronisation events, such as pause and unpause, for the rest of the system to call. The gen_fsm processes will, once started, run until told to stop. The human_fsm and zombie_fsm modules have been implemented with gen_fsm behaviours.

[4]http://www.erlang.org/doc/man/supervisor.html
[5]http://www.erlang.org/doc/design_principles/des_princ.html
[6]http://learnyousomeerlang.com/what-is-otp
[7]http://www.erlang.org/doc/man/supervisor.html
[8]http://www.erlang.org/doc/design_principles/des_princ.html
[9]http://learnyousomeerlang.com/what-is-otp
[10]http://www.erlang.org/doc/design_principles/sup_princ.html#id68643
[11]http://www.erlang.org/doc/design_principles/gen_server_concepts.html
[12]http://www.erlang.org/doc/design_principles/fsm.html

16.2 System Architecture

As introduced in section 16.1 swarmer is built around a Supervisor-Child architecture. This section will explain the setup of the application in a little more detail.

When the application is started, the initial process is an instance of the swarm_server module. This is responsible for starting the Websocket and spawning the system supervisor module, swarm_sup. The system uses the Cowboy[13] HTTP server to manage Websocket communication to the user client, which is covered in section 17. The web socket is managed by a module called swarm_handler.

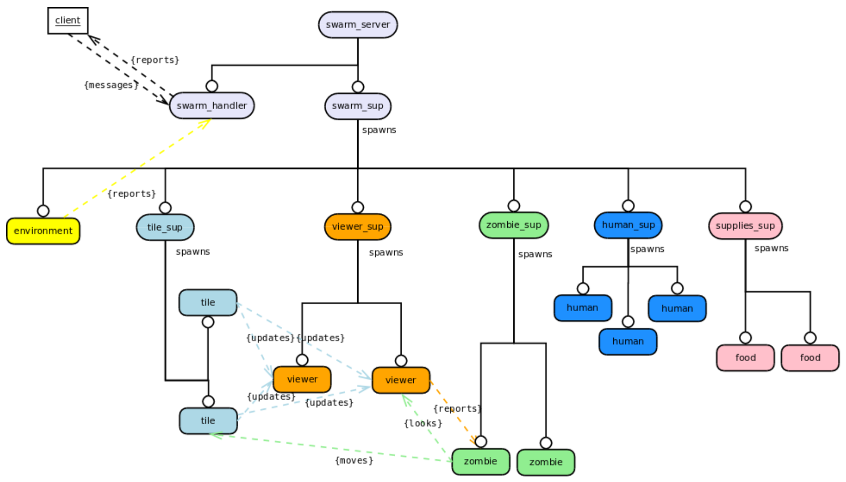


Figure 7: System Architecture

On initialisation, the swarm_sup will spawn the children processes shown in figure 6 and the system will wait for a message from the client to define what to spawn, this will be covered in section 16.3. Once received, it will then continue waiting until swarm_handler receives a message to start the system.

In order to visualise what is happening in the system, the client systematically requests a report from swarmer. To report back to the client, the environment module requests the status of all of the supervisors children nodes. This is returned to swarm_handler which is encoded into a JSON object using the jsx dependency and sent to the client.

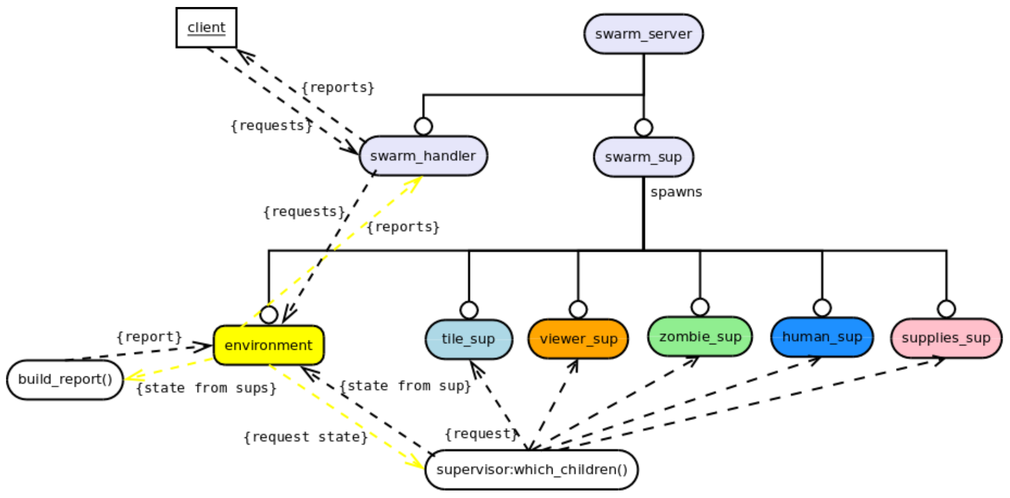


Figure 8: Reporting to the Client

[13]http://ninenines.eu/docs/en/cowboy/1.0/guide

The map is created out of a grid of `tile` instances with assigned `viewer` instances. This is explained in more detail in section 18.1. When the entities are initially spawned they are initialised in a paused state until the system is told to start. After this, the entities begin to run carrying out their type specific behaviours. The way in which entities interact with the environment is explained in detail in section 18.3. Zombies, covered in section 20, in the simulation are created as an instance of the `zombie_fsm` module, and attempt to find human entities in the environment. Over time they will slow down if they have not eaten in a long time, and can turn human processes into zombies if they succeed in killing one, thus spreading the swarm. Humans on the other hand are trying to survive, meaning they must scavenge for food and escape the horde of zombies. Their behaviour is covered in section 21.

16.3 System Setup

The initial set-up of the simulation environment is handled by the `environment` module. This deals with spawning `tile`, `viewer`, `supplies`, `human_fsm` and `zombie_fsm` processes according to the set-up instructions received from `swarm_handler`. On a normal set-up, the system would be told to create an amount of tiles, defined by a 'grid arrity' and an amount of each entity type to spawn. Grid arrity would be an integer between the value of 1 and 10, 1 telling the system to build a 1x1 grid and 10 telling the system to build a 10x10 grid.

Obstructions, which are covered in section 19, play a big part in the way our simulation runs, they block line of sight and prevent movement through them. During the initial setup, this is created through a list of blocked coordinates passed into the tiles on initial set-up.

16.3.1 Making the Grid

When the `environment` process receives a `make_grid` request it will firstly purge the system of currently spawned processes; restarting the system from the ground up. This prevents old instances of processes remaining in the supervision tree.

The process will then proceed to create a grid of given arrity by spawning `tile` instances of a given size. In order to create a grid like structure, a row of tiles is spawned, assigning each an origin coordinate and an end coordinate before creating another row of tiles. All `tile` processes are registered to Erlang as a named process of the form "tileXOYO", O being the origin point for each axis. Each of these then has a viewer process assigned to it.

17 Client and visualisation

J. Pearse

We were aware this would be the user-facing component of the system we wanted to make it attractive, easy to interpret, and use. The software which we tentatively call the client, was designed to fulfil two major functions.

1. Provide a visualisation of the current state of the simulation and a method of control of various parameters.

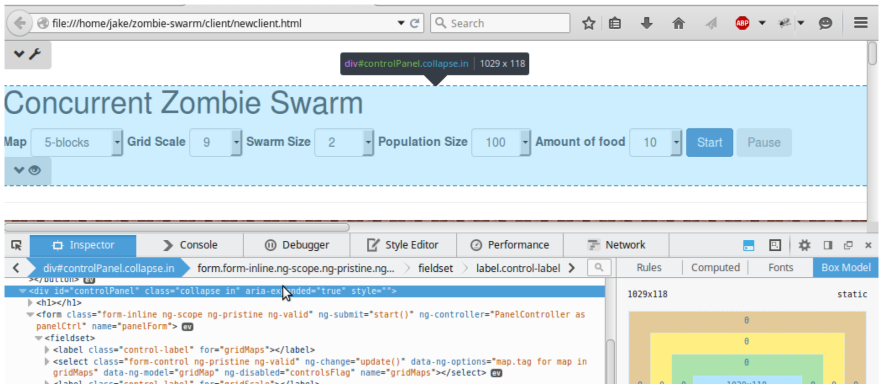
2. The secondary function of the software is to provide a diagnostic tool to be used to examine the state of the system and individual entities.

The client was always envisioned as a prototype but during the course of the systems development the client came to play a substantial role. The development of the client itself became a major focus of the project at points and a lot of effort was required to learn the two JavaScript libraries utilised.

17.1 Core technologies choices

Early on the group selected HTML and JavaScript as the platform, which the client would leverage. The initial stages of development centred around establishing two-way communication via WebSocket. The job of establishing the socket was delegated to the Cowboy framework (see section: 16.2). For the sake of convenience we decided on JSON as the format of WebSocket messages, on the Erlang side we initially identified the MochiJSON library as our method for encoding to JSON however as development progressed we ran into some difficulties with the library, Specifically relating to the encoding of atoms as strings and we switched to the JSX library for our JSON encoder. To aid the development of the HTML we selected several well known JavaScript libraries; JQuery, Angular.js and D3.js. CSS library bootstrap was also included.

17.2 Simulation control



file:///home/jake/zombie-swarm/client/newclient.html

div#controlPanel.collapse.in | 1029 x 118

Concurrent Zombie Swarm

Map: 5-blocks | Grid Scale: 9 | Swarm Size: 2 | Population Size: 100 | Amount of food: 10

div#controlPanel.collapse.in > form.form-inline.ng-scope.ng-pristine.ng... > fieldset > label.control-label

```
<div id="controlPanel" class="collapse in" aria-expanded="true" style="">
  <h1></h1>
  <form class="form-inline ng-scope ng-pristine ng-valid" ng-submit="start()" ng-controller="PanelController as
  panelCtrl" name="panelForm"> ev
    <fieldset>
      <label class="control-label" for="gridMaps"></label>
      <select class="form-control ng-pristine ng-valid" ng-change="update()" data-ng-options="map.tag for map in
      gridMaps" data-ng-model="gridMap" ng-disabled="controlsFlag" name="gridMaps"></select> ev
```

Figure 9: System control panel

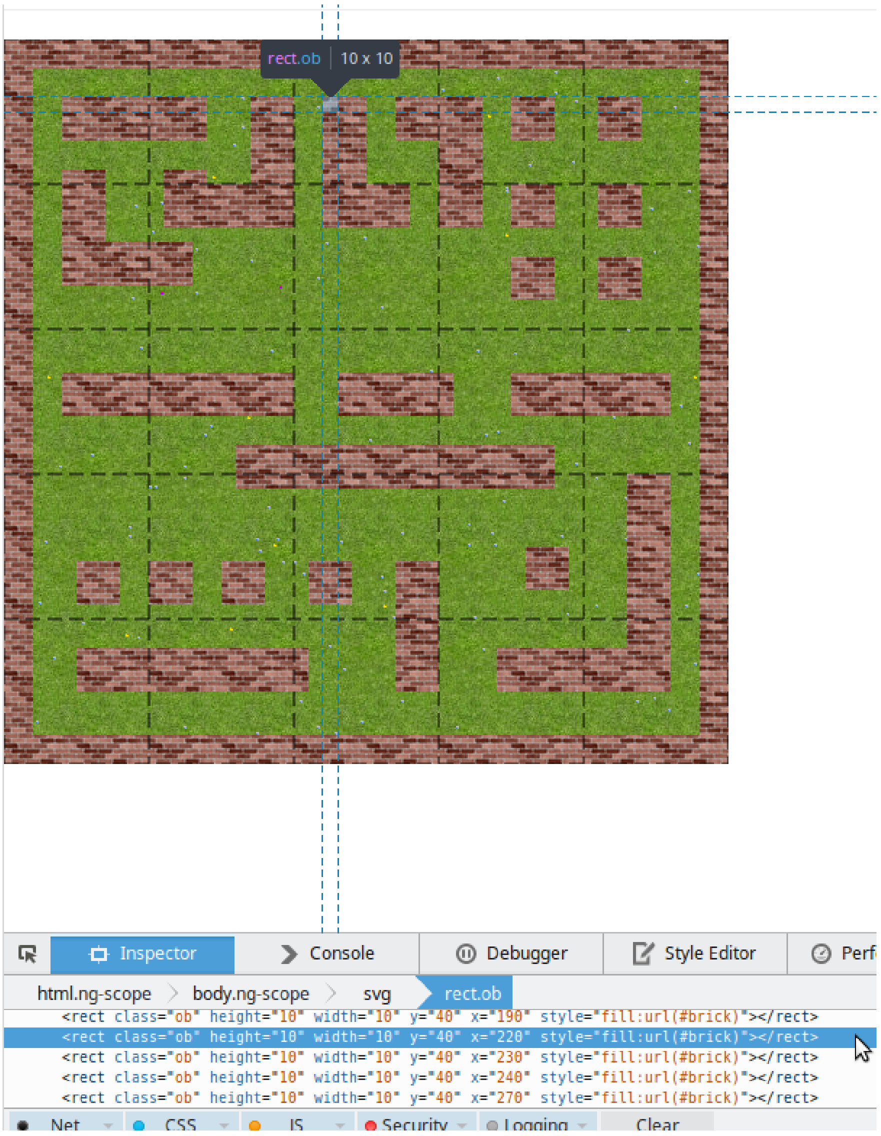
The <body> element is wrapped in an JavaScript ng-controller closure, these type of closures provide scoping of JavaScript variables. At the very top of the document a collapsible form provides access to user definable parameters of the system. Angular provides two-way data binding between HTML elements and JavaScript variables defined within the controller scope, each element of the form is bound to a corresponding variable. A JSON object [14] with the values of these variables is sent over the WebSocket to set up the system. A number of predefined obstruction maps are selectable, the number of tiles in the grid is determined by the map selected. There is also a button to pause the simulation when running and a start button to begin the report loop and start the simulation.

[14] The setup JSON is structured {Type,Arity,SwarmSize,PopSize,Obstructions,FoodAmount} Using this information the *environment* module can set up the simulation framework

17.3 Angular.js

The client software underwent one major revision, the initial version was constructed piece-by-piece in JavaScript in response to each change in the simulation architecture resulting in a lot of messy and difficult to maintain code, once the architecture was largely implemented the client was entirely rewritten using the AngularJS framework. The re-factoring into the AnuglarJS framework granted an opportunity for the convoluted logic to abstracted into a more easily maintainable code-base. The `app.js` source code file contains all the logic which maps variables to HTML elements and communicates with the swarm application essentially providing an control layer between the JSON data coming from the swarm application and mapping the data to the visualisation library.

17.4 D3.js library



```
<rect class="ob" height="10" width="10" y="40" x="190" style="fill:url(#brick)"></rect>
<rect class="ob" height="10" width="10" y="40" x="220" style="fill:url(#brick)"></rect>
<rect class="ob" height="10" width="10" y="40" x="230" style="fill:url(#brick)"></rect>
<rect class="ob" height="10" width="10" y="40" x="240" style="fill:url(#brick)"></rect>
<rect class="ob" height="10" width="10" y="40" x="270" style="fill:url(#brick)"></rect>
```

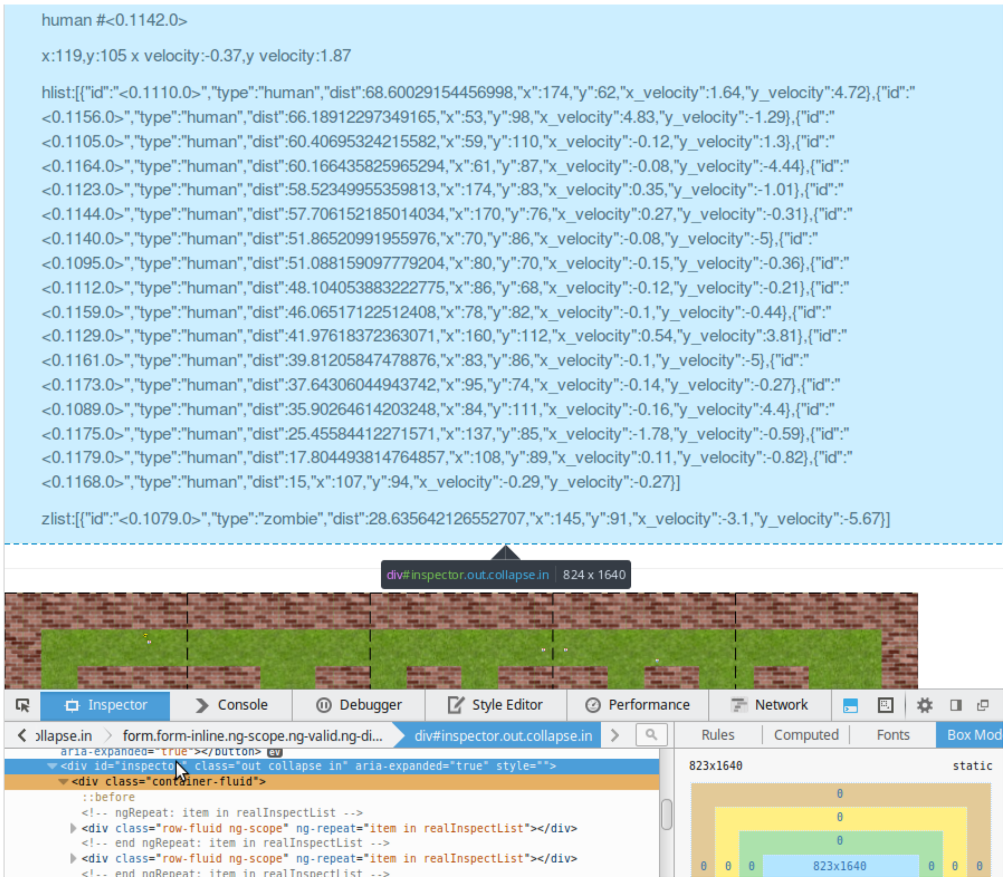
Figure 10: This typical SVG visualisation contains around 3000 unique elements, without sight-lines

In the early stages of our background research the D3 data visualisation library was identified as our visualisation solution. D3 provides a library for mapping data to elements of a scalable vector graphic image. All of the functions related to the visualisation and animation are contained in the `draw.js` source file. As the client provides important diagnostic information, almost everything visualised is from data returned over the WebSocket the only exception being the axis lines marking tile separation. All obstructions, entities and sight-lines are drawn from application data. D3 itself provides a sophisticated model for joining data to SVG elements, updating elements based on changes in the data and removing elements form the visualisation when they are removed from the joined data.

17.5 System pause

We also included a button to pause the currently running simulation. Because every entity must be sent a pause signal individually you can occasionally observe slight a slight delay between one entity entering its paused state and another, it's a staggered pause rather than a hard pause. While the system is paused there is still a continuous stream of reports coming over the WebSocket the data in each report however is unchanging while the system is paused.

17.6 Inspection panel



```
human #<0.1142.0>

x:119,y:105 x velocity:-0.37,y velocity:1.87

hlist:[{"id":"<0.1110.0>","type":"human","dist":68.60029154456998,"x":174,"y":62,"x_velocity":1.64,"y_velocity":4.72},{"id":"
<0.1156.0>","type":"human","dist":66.18912297349165,"x":53,"y":98,"x_velocity":4.83,"y_velocity":-1.29},{"id":"
<0.1105.0>","type":"human","dist":60.40695324215582,"x":59,"y":110,"x_velocity":-0.12,"y_velocity":1.3},{"id":"
<0.1164.0>","type":"human","dist":60.166435825965294,"x":61,"y":87,"x_velocity":-0.08,"y_velocity":-4.44},{"id":"
<0.1123.0>","type":"human","dist":58.52349955359813,"x":174,"y":83,"x_velocity":0.35,"y_velocity":-1.01},{"id":"
<0.1144.0>","type":"human","dist":57.706152185014034,"x":170,"y":76,"x_velocity":0.27,"y_velocity":-0.31},{"id":"
<0.1140.0>","type":"human","dist":51.86520991955976,"x":70,"y":86,"x_velocity":-0.08,"y_velocity":-5},{"id":"
<0.1095.0>","type":"human","dist":51.088159097779204,"x":80,"y":70,"x_velocity":-0.15,"y_velocity":-0.36},{"id":"
<0.1112.0>","type":"human","dist":48.104053883222775,"x":86,"y":68,"x_velocity":-0.12,"y_velocity":-0.21},{"id":"
<0.1159.0>","type":"human","dist":46.06517122512408,"x":78,"y":82,"x_velocity":-0.1,"y_velocity":-0.44},{"id":"
<0.1129.0>","type":"human","dist":41.97618372363071,"x":160,"y":112,"x_velocity":0.54,"y_velocity":3.81},{"id":"
<0.1161.0>","type":"human","dist":39.81205847478876,"x":83,"y":86,"x_velocity":-0.1,"y_velocity":-5},{"id":"
<0.1173.0>","type":"human","dist":37.64306044943742,"x":95,"y":74,"x_velocity":-0.14,"y_velocity":-0.27},{"id":"
<0.1089.0>","type":"human","dist":35.90264614203248,"x":84,"y":111,"x_velocity":-0.16,"y_velocity":4.4},{"id":"
<0.1175.0>","type":"human","dist":25.45584412271571,"x":137,"y":85,"x_velocity":-1.78,"y_velocity":-0.59},{"id":"
<0.1179.0>","type":"human","dist":17.804493814764857,"x":108,"y":89,"x_velocity":0.11,"y_velocity":-0.82},{"id":"
<0.1168.0>","type":"human","dist":15,"x":107,"y":94,"x_velocity":-0.29,"y_velocity":-0.27}]

zlist:[{"id":"<0.1079.0>","type":"zombie","dist":28.635642126552707,"x":145,"y":91,"x_velocity":-3.1,"y_velocity":-5.67}]
```

```
aria-expanded="true"></button> ev
<div id="inspector" class="out collapse in" aria-expanded="true" style="">
  <div class="container-fluid">
    ::before
    <!-- ngRepeat: item in realInspectList -->
    <div class="row-fluid ng-scope" ng-repeat="item in realInspectList"></div>
    <!-- end ngRepeat: item in realInspectList -->
    <div class="row-fluid ng-scope" ng-repeat="item in realInspectList"></div>
    <!-- end ngRepeat: item in realInspectList -->
```

Figure 11: The inspector panel reveals the state of selected entities

The inspector provides continualy updated inspection of the state of selected processes, we took time to ensure that everything in the entity state is reported over the WebSocket so any aspect of an entities sate can simply be added to the inspector display. To add an entity to the inspector simply click on it in the visualisation. For detail on the implementation of the panel see section:15.3

17.7 Sight-lines

Sight lines were a late addition to the visualisation, mapping a line from each selected entity to all other entities recorded in its state, this turned out to be one of the most useful features in understanding entity state and concurrent behaviour in the simulation. More detail on sight-lines is included in Section:19.2

17.8 Report loop

The central function which handles communication over the WebSocket utilises a callback to pass a report request over the WebSocket at a fixed interval [15] when the WebSocket replies with the system data the data is passed over to the update functions in the `draw.js` module. The report loop also handles the processing of the entities selected for inclusion in the inspector.

[15] We experimented with different values for the time out, there is a fine balance between state-machine time out, simulation size and WebSocket timing

18 Entity Interaction

J. Mitchard

As explained in section 16.2, the simulation has been implemented with a tile-viewer architecture. Whilst this allows for the the system to run more efficiently, it meant that a slightly more creative way of entities maintaining awareness had to be created. This section will cover the specifics of how tile and viewer processes interact, and how each entity interacts with them throughout the simulation.

18.1 Tiles and Viewers

When the map of tiles and viewers are created by *environment.fsm*, they are each assigned a unique viewer along with the viewers of all of the neighbouring tile processes. This creates a grid as shown in section 9.2. The purpose of these viewers is to allow entities to ask the viewer for their current tile what is around them, or, more specifically, what other entities are currently located on the same tile and in each of those surrounding it.

Each tile updates each of the viewers making up its 'neighbourhood' of local tiles and viewers with new data regarding which entities are on it and where. Each tile reports to its viewers from four sets of data held in its state. These are:

zombie_map - A map of each zombie currently on the tile with the zombies Process ID as the key.

human_map - A map of each human currently on the tile with the humans Process ID as the key.

item_map - A map of each item currently on the tile with the items Process ID as the key.

obs_list - A list of obstructed coordinates in the tile.

18.2 Keeping the System Current

Each time an entity moves, enters or leaves a tile, the tile in which the action has taken place will update its associated neighbouring viewers with a new set of data relating to the type of entity that has been updated. For example, if a zombie moves from one position within the tile to another, the tile would only update the viewers with its zombie data, rather than that of the items or humans as well.

The viewer holds similar information to the tiles, however instead of mapping each individual process as a key, it is mapped by tile with a list of entities as the values. This allows for a simple method of updating the viewer processes.

Because these data sets change so often and unpredictably, Erlang Maps are perfect for the job as when new data is inserted into the map with a pre existing key, the values are overwritten with the new set. This is considerably quicker than having to iterate through a list and changing its value.

18.3 Entities and Awareness

In order for humans and zombies to have an awareness of their surroundings, it is necessary for them to ask the Viewer process for their current Tile for information; regarding nearby zombies, items, obstacles, and other humans. Because this information is constantly changing, it will have to do this at the start of each cycle of its state machine.

However, because each entity has a defined line of sight that is considerably less than the length and height of three tiles, which is what the viewer would provide information for, these lists will have to be filtered down to have this taken into consideration.

Firstly, each list is filtered to remove entities further than a defined distance away, the line of sight varies between humans and zombies. Secondly, the list uses the *findline* function from the *los.erl* module, which is introduced in section 19, to filter out entities in which line of sight is obstructed by obstacles. This will provide each entity with the information needed to make informed decisions of what to do next.

18.4 Example of Process Communication

This diagam shows an example snapshot of communication between a human, tile and viewer processes in which the human requests information from the viewer, and then requests to move to a new position. The example assumes that the requested position was not obstructed.

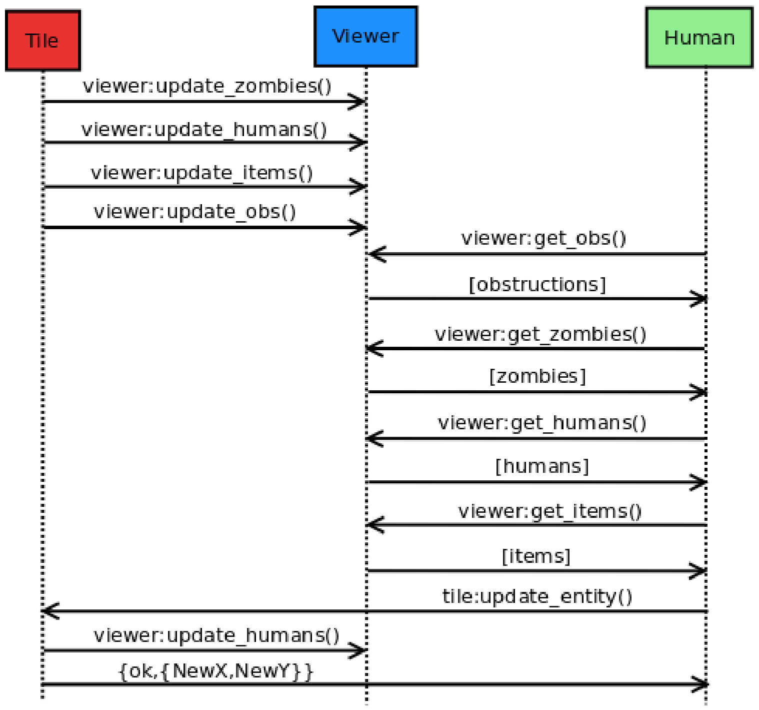


Figure 12: Human, Tile and Viewer Communication

19 Obstructions and line-of-sight

J. Pearse

19.1 Physical obstructions in the simulation space

One of the primary motivations for including obstructions in the simulation was to allow the implementation of path finding and the potential for more complex or unexpected behaviours to arise. There was concern that the resolution of the simulation could incur to high a computational cost with multiple agents all executing heuristic searches concurrently over a large space. As a result it was decided to implement obstructions at a lower resolution than the one used naturally by entities in the simulation. Although the resolution of the 'obstruction grid' was parametrized, it has only been tested at a single resolution $\frac{1}{10}$ of the the resolution of the natural simulation space. As each tile in the simulation has a resolution of 50^2 the resolution of the obstruction grid was set as a matrix of 5^2 cells, so each tile contains a grid of 10^2 obstruction cells. We wanted a simple method of drawing maps of obstructions so we selected a paint program with the capability to export ascii art pictures. Each ascii art map was stripped of line breaks and included in the main JavaScript application as a string. The strings were then indexed to locate the integer positions of obstructed cells and using modulo and integer division, translated into co-ordinates within the obstruction grid space. The implementation of the obstruction grid within the simulation although simplistic with hindsight was a fairly complicated process involving a lot of experimentation, testing and a great deal of drawing everything out on graph paper to make sure it was functioning correctly. However once the correct co-ordinate translations had been established the obstruction grid worked smoothly. The process of instantiating the obstruction map is;

1. Iterate over the map string (in JavaScript) to create an array, of indices, of obstructed cells
2. Pass the array over the web socket into the swarm application when the simulation is being set-up
3. As part of the `make_grid` method of the `environment` module, the obstructed cells are assigned to the state of their respective tile when the tile is created.

Each tile maintains a list of obstructed cells inside its geometry as co-ordinate pair on the obstructed grid scale. When entities call a function which requires consideration of obstructions the native co-ordinates are simply divided by 5 to determine if they fall inside an obstructed cell.

19.2 Line-of-sight

The main system dynamic which makes use of obstructions is line-of-sight, when we first implemented the obstruction system we observed the behaviour of entities become very unrealistic, there was nothing in place to prevent entities from 'seeing' other entities through obstructed cells. This caused large swarms to become trapped up against the opposite side of obstructions as they continually tried to close the distance between themselves and the target. It was obvious that to maintain any sense of realism we would need to prevent entities from seeing through obstructions. It was clear that data obtained from the viewer (already filtered by distance) would need an additional filter to restrict it to only those entities, to which an uninterrupted line segment could be traced i.e a line segment which did not intersect an obstructed cell. An extra module was written to provide the linear algebra functions required to trace these line segments (`los.erl`), again a seemingly simple task in hindsight, this required a great deal of testing and drawing out of graphs with pencils and paper. Once the list of visible entities has been established it is passed over the the WebSocket as part of the entity state. The lists of zombie-type entities and human-type entities are separated in the state but each entity in either list is stored with its corresponding position. SVG line elements are simply defined by four points $(X1, Y1, X2, Y2)$ these obviously correspond to the entity (X, Y) and target (X, Y). In the final version of the client, the animation has been removed from the sight-lines visualisation. As the entity is still being animated this sometimes causes the origin of the lines to jump ahead of the entity as they move forward, however the animation of a very large number of lines proved too computationally expensive on some of our test systems.

19.3 Conclusion

As a result an extra functionality was added to the client (see section: 17.7) to visualise every uninterrupted line which could be drawn from a given entity. This new functionality when added to the client proved to be an unexpected source of information about the system state and inadvertently revealed one of the more interesting aspects of trying to visualise a highly concurrent system. Unfortunately, due to time constraints real path finding was never fully integrated into the system although we did make some progress in that direction. We consider the obstruction system as a key component of the simulation which contributes to the emergence of many interesting entity interactions.

20 Zombie Intelligence

R. Hales

The zombie entities are designed to be relatively unintelligent, only interested in finding,chasing and killing humans,avoiding obstacles and following a group of other zombies to form a swarm.To achieve this zombies have had multiple behaviours implemented to enable them to act realistically and follow the rules of our system.

20.1 Movement

Before going into specific behaviours it is important to understand how zombies move in the simulation. Each zombie contains two pairs of variables that are directly used in determining it's next position; An X co-ordinate and a Y co-ordinate, which together represent the current position of the zombie in the simulation, and an X velocity and a Y Velocity, which represent how fast the zombie is currently moving and in which direction. As well as these variables, whenever a zombie makes a decision two numbers are returned.These represent the change in X velocity and the change in Y velocity resulting from the decision.These numbers are naturally added to the current X velocity and current Y velocity respectively to create the new velocity of the zombie. New X Velocity = Current X Velocity + Change in X Velocity New Y Velocity = Current Y Velocity + Change in Y Velocity These new velocities are then rounded down and added to their respective current co-ordinates to give the new X and new Y co-ordinate and thus the new position of the zombie. New X = Current X + Rounded New X Velocity New Y = Current Y + Rounded New Y Velocity These new co-ordinates then have checks performed on them to ensure the move is legal and if not to change it to the nearest legal move.After this the zombie is moved and is then ready to perform his next move.

20.2 Velocity

The reason that when a zombie makes a decision it changes velocity rather than directly changing its position is that it allows the system to keep track of the zombies current speed and heading.because of this if a zombie keeps choosing to head in the same direction it will gradually speed up from moving small distances each action to moving larger distances, simulating the zombie building up momentum. This momentum also means that if a zombie suddenly decides to change direction it will not instantly move in the direction he wants, instead it will start turning gradually from its current heading towards the heading it wants to be going. The zombies therefore move in a much more natural and realistic way than without the use of velocities.

20.3 Decision Making

The way zombies decide how to change their velocity is to investigate their local area to find all the human and zombies entities it can see and then uses this information to perform the highest priority action that is possible.Priority is determined by the types of entities visible and how close those entities are.In particular entities will be considered either in reach of the zombie or within sight of the zombie based on distance.

20.3.1 Priorities

The zombies priorities are very simple.Zombies prioritise killing humans above all else, if a human is within reach of the zombie it will try and kill and eat the human.The next priority is not being too close to another zombie, if another zombie is within reach then the zombies will attempt to move away from each other to prevent clustering together, this is to ensure that zombies form swarms rather than clustering together too closely and being unable to move away from each other due to getting surrounded by other entities.Next if a human is within sight of a zombie then the zombie will attempt to chase after the human, this allows zombies to break from swarming and attempt to kill humans.The next priority is forming a swarm with other zombies, this allows the zombies to form groups and combined with the previous priority will cause whole swarms of zombies to chase after humans even if not all of them can see the human. Finally if there is no other entities to interact with the zombies will move in an attempt to locate something to interact with.

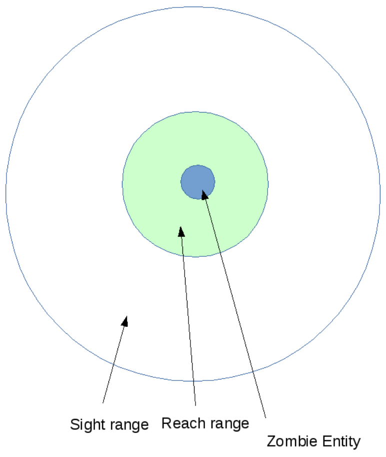


Figure 13: Illustration of a zombies sight and reach ranges

20.3.2 Visibility

To find all the humans and zombies within sight the zombie queries their viewer to receive lists of the human and zombie entities that the viewer is aware of, the zombie then iterates through these lists, calculating the distance of each entity from the zombie and removing any that are out of sight range.The zombie then cuts the list again to remove all entities that would be blocked from line of sight by obstructions.Finally the zombie sorts the list in order of distance so that the closest entities are first. The zombie can then use these lists to determine it's action.

20.3.3 Human entity within reach

The highest priority is if a human entity is within reach of a zombie.if this happens the zombie will attempt to kill the human.There is a random chance the zombie will fail to kill the human, in which case it will just continue at the same velocity and nothing else happens this action. If the zombie succeeds it will still carry on at the same velocity but it will also pass a message to the human. This message will cause the human process to perform a function which creates a zombie entity in the humans location and kills the human entity.

20.3.4 Zombie entity within reach

The second highest priority is if there is another zombie within reach.If this is the case the acting zombie will attempt to avoid collision with the second zombie.This is based off the collision avoidance used in boids algorithms and is intended to stop a swarm of zombies from all clustering into a single point.The acting zombie will change its velocity to attempt to move in the opposite direction of the second zombie's location compared to the acting zombies. ie. if a second zombie is to the upper left of the acting zombie, the acting zombie will change velocity down and right.

20.3.5 Human Entity within sight

The third highest priority is if there is at least one human entity not within reach but within sight.The zombie will choose the closest human entity to it and change velocity in its direction. ie if there is a human five spaces above the zombie and another ten to the right, the zombie will change velocity up.

20.3.6 Zombie entity within sight

The fourth highest priority action is if there are zombie entities within sight but not reach, and there are no human entities within sight.The zombie will have two factors effecting its change in velocity in this case. Firstly the zombie will try to move towards the centre point of all zombies it can see, it does this by iterating through the list of zombie entities and averaging out their X and Y positions to find a target X and Y co-ordinate.It then calculates the direction and distance away the target is from its current position and changes its velocity by a fraction of those X and Y values. Secondly the zombie will attempt to match the velocities of the other zombies it can see, this is calculated in the same

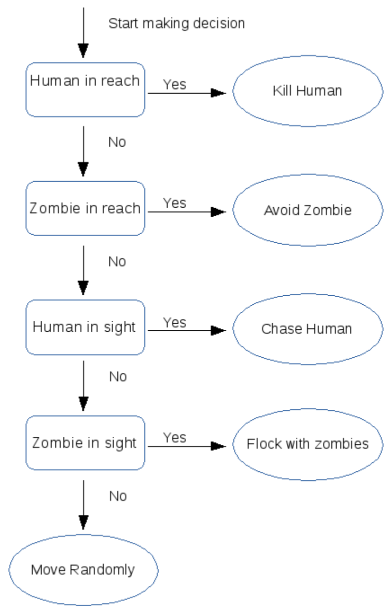


Figure 14: Zombie priorities diagram

way as moving towards the centre point, but uses the X and Y velocity of the other zombies instead. the two resulting changes in velocity are added together to get the total change in X and Y velocity. This action is also based off the boids algorithm, in particular the idea of flock cohesion and alignment.

20.3.7 No entities in sight

The lowest priority action is if there are no humans or zombies in sight.The zombie will start to change it's velocity randomly, this is to ensure that if a zombie has no targets it will still move and appear to search for other entities to interact with.

20.4 Behaviour After Decision Making

After making a decision the change in velocity resulting is added to the zombies current velocity then several limits will be checked to prevent the zombie moving against the rules of the simulation.

20.4.1 Limit speed

Zombies have a maximum distance they can move in one action,if the velocity of the zombie exceeds this value then a new velocity is calculated that keeps the same heading but with a distance at the maximum allowed, this prevents zombies moving unrealistically fast if they repeatedly change velocity to go the same direction.

20.4.2 Energy

Energy represents how recently a zombie has eaten. Energy decreases every time a zombie makes an action. If the energy of a zombie falls too low then the maximum distance a zombie can move per action is decreased to represent the zombie slowing down from hunger.Energy is refilled by killing humans and eating them.

20.4.3 Obstructions

If the spot the zombie intends to move to is obstructed either by another entity or an obstacle then the zombie will bounce off it to avoid collisions.If the obstacle is a wall, the zombie is placed next to the wall and its velocity is changed to -1 of whatever direction it hit the wall in but kept the same in the other axis to enable the zombie to keep moving without repeatedly hitting the wall again. If the zombie hit another entity it is placed next to the entity it hit but its velocity is not reset as there is little risk of repeatedly running into the same entity as all other entities will be moving at the same time.

21 Human Intelligence

J. Mitchard

From early on, we had decided that human entities are to be more intelligent than their undead counterparts. In order to establish a sense of realism in our simulation, the human entities have to make complicated decisions based on their current situation and what is going on around them. The way in which an moves is covered in section 20.1, as this is generic to both types of entity within the simulation. This section will explain which behaviours have been implemented, which decisions are made in order to achieve them and what is taken into consideration to make them believable actions.

21.1 Human Behaviours

The human behaviours in the simulation are modelled through a heavily extended version of the Boids algorithm, as introduced in section 14.1. In order to survive, the human entities have to avoid the zombies, however it is also necessary for humans to eat food found around the world in order to maintain energy. Over time the humans become hungry which, in turn, will cause the humans to slow down as they lose energy, making them easier for the zombies to catch.

21.1.1 What Behaviours are Present?

Human entities in our simulation are currently able to replicate a number of different behaviours, these are:

Collision Avoidance - The human avoids hitting both other entities and obstacles.

Zombie Repulsion - In order to survive, a human entity must try and avoid contact with zombies.

Flock Attraction - Humans are social creatures, this behaviour will allow the humans to attempt to group together with other humans and move as a group.

Food Attraction - When hungry, and there is nearby food a human will be attracted to the item of food.

Maintaining Memory - Food can be scarce, so it is important for a human to remember the location of food that it doesn't need at present.

Pathfinding to Food - When hungry, a human will try and move towards known locations of food.

Search - When there is nothing else to do, a human can search the area around him, looking for food or other humans.

21.2 Human Intelligence Implementation

21.2.1 The Decision Making Process

Humans decide what to do next based on a set of weighted priorities, taking into consideration their current surroundings and circumstances. Each time a human runs through its state machine cycle, introduced in section 16.1.1, it will calculate new levels of hunger and energy and request new, updated lists of nearby entities from its current tiles' Viewer, this process is explained in section 18.3. These include a list of other humans, a list of zombies, and a list of items; allowing the human to know what is happening around him.

Based on the information received from the Viewer the human can now make an informed decision. The priorities are ordered as such:

1. Move to avoid crashing into other entities.
2. If there are nearby zombies, move away to avoid them.
3. If hungry and currently next to the food, attempt to pick it up.
4. If hungry, there are no zombies and there is a path to an item set, follow the next position from the path.
5. If hungry, there are no zombies and there is nearby food, move towards it.

6. If there are no zombies, group together with other human entities.
7. If nothing else matches, search for food.

The order of priority has been heavily considered and tweaked to ensure the humans act as realistically as possible. For example, we believe it unlikely that a human would run aimlessly towards a source of food knowing full well that there is a number of zombies between itself and the food. In order to represent this decision we have put finding food lower in the chain of priorities than avoiding zombies.

21.2.2 Implementing the Choices

Implementing these behaviours in Erlang presented some unique challenges, but also some very interesting advantages.

Due to the nature of a concurrent system, with lots of processes running simultaneously, timing can be a complicated issue. An example of where this sort of issue could become a problem is that multiple humans could attempt to pick up an item of food at the same time. The process for the item of food would receive both requests in a queue, however would only respond an 'accept' message to the first one, any others it would send a 'reject' message. This allows us to pattern match against what is received from the food process and deal with it accordingly, whilst avoiding causing a deadlock in the system.

This snippet of code from *human_fsm.erl* shows how the human process matches against what is returned:

```
% There are no zombies, I have no path, move towards food and attempt to eat it
make_choice(_,[],{ItemId,{ItemX,ItemY,food,_}}, very_hungry, _Path, #state{x=X,
    y=Y}) ->
   case swarm_libs:pyth(X,Y,ItemX,ItemY) of
      Value when Value =< 2 ->
         Item = supplies:picked_up(ItemId),
         case Item of
            ok ->
               {X,Y,eaten};
            _ ->
               {X,Y}
         end;
      _ ->
         boids_functions:super_attractor(X,Y, ItemX, ItemY, ?SUPER_EFFECT)
   end;
```

On the other hand, functional programming has presented us with a number of interesting ways of solving problems. Deciding what behaviour to next carry out lends itself perfectly to pattern matching, whereby the function will fall through multiple function headers until one meets the current state. Using pattern matching in Erlang, we have created a 'one-size-fits-all' function that decides on what the entity should do next; The behaviours are each implemented in a different branch of the function, each meeting a different specification until one is accepted by the function and the behaviour is carried out.

In the above example, the human entity knows that there are no nearby zombies in sight, and that it is near an item of food. The human will attempt to pick it up if it is close enough, and if another human hasn't picked it up before them it will return to the main state machine that the human has eaten. Otherwise, the human will make an attempt to move towards the food.

22 Survey Results

R. Hales

As the poster fair involved showing a number of people a stable release of our project, we decided it would be a good chance to have users test our project and to see how well they felt it met our aims. To do this testing we created a short survey for the users to fill in after using the simulation. In total we had twelve people answer our survey.

22.1 How realistic is our simulation?

Since our goal was to create a realistic simulation of zombies, the first question we asked was how realistic is our simulation? This was to be answered with a number between 1 and 5, 1 being not very realistic and 5 being very realistic. We had an average result of 4.3 with no users answering below 4. Therefore every user we surveyed thought our simulation was realistic.

22.2 How easy to understand is our system?

We also wanted to know how much users could understand about the behaviour of the system without having to have it explained in great detail. Again we asked for answers between 1 and 5, with 1 being very difficult to understand and 5 being very easy to understand. We had an average result of 4.3, with a range of 3 to 5 given as answers. Thus showing our system was largely understood at least fairly easily.

22.3 How easy to understand is the user interface?

As well as the actual system, we also wanted to know specifically how easy to understand our user interface was. Again answers were given between 1 and 5, with 1 being very difficult to understand and 5 being very easy to understand. We had an average result of 4.2, with a range of 2 to 5 given as answers. This showed the user interface was largely understood but there is room for improvement as some people had trouble with it.Based on feedback changing the sprites to more clearly respresent zombies and humans and making it more obvious that entities are clickable are the most obvious improvements.

22.4 Is our simulation speed too fast or slow?

One subject we were concerned with was the speed of our simulation, if it is too fast it can become hard to follow and understand, whereas if it is too slow it can become uninteresting and hard to see behaviour clearly. For the survey we asked the users to answer whether they thought the simulation was much too slow, slightly too slow, the correct speed, slightly too fast or much too fast to help us decide on a speed for the simulation. Two thirds of people said the speed was fine as it was, everyone who did not said the speed was too slow although only one person thought it was much too slow. As such we decided to keep the speed the same.

22.5 Are there any changes you would make to our simulations behaviour?

We asked our users if they'd make any changes to the simulations behaviour, in case there were suggestions we would not have considered otherwise. We got the same suggestion multiple times, humans should be able to fight back against zombies and have a win case. This was a feature we had hoped to include since early in development but did not have time for ultimately.

22.6 Are there any changes you would make to the user interface?

We also asked users if they would make changes to the user interface for the same reasons we asked about changing simulation behaviour. The most common answer was to make the interface more visually appealing, with some people suggesting the visuals should match with the concept more. We also got several suggestions to make the animation of the simulation smoother. As well as these one person asked to make the entities easier to select while the simulation is running and another asked to make it easier to tell selected entities apart.

22.7 Do you have any other comments you would like to make about our system?

Although we asked this question, none of our users had any extra suggestions to make.

23 Unimplemented Features and Fixes

`R. Hales`

Unfortunately we did not have time to implement everything we would have liked to in the time given for our project. The following are all features we intended to include at some point in our project and would like to implement if given more time.

23.1 Features

23.1.1 Humans fighting back

One feature we had in mind since early development was for humans to be able to fight back against zombies. To do this we intended to have a more complex combat system where, based on the number of entities of each type in the immediate vicinity and the states of those entities, the results of combat are calculated and either humans or zombies are killed.

23.1.2 Weapons

Extending on humans fighting back, we were also planning on implementing weapons, such as guns. These could be used to improve a humans chance of winning against zombies. The weapons would be items similar to the food implemented in the current system, which would be sought out and picked up by humans as part of their behaviour.

23.1.3 Different families of humans

We would like to create more variety in human behaviour by implementing different varieties or families of humans. This would allow different humans to have different priorities while path-finding. This would create more interesting behaviour and also allow a way to see what sort of behaviour improves a humans survivability.

23.1.4 More visually appealing GUI

We would have liked to create a more interesting looking user interface,as our current one is very plain. One idea was to make an interface that visually more closely fits the theme of a zombie apocalypse.

23.1.5 Humans turning into corpses before becoming zombies

We would also have liked to create a form of entity that humans become between dying and becoming zombies. This would allow a slower transformation, making it less easy for a single zombie to quickly wipe out lots of humans, and would allow the zombies more chance to eat from the new form.

23.2 Fixes

23.2.1 Ghosts

One bug we have identified but have not fixed is entities that seem to move without regards for obstacles or other entities and are ignored by other entities in return. We believe this bug to be caused by the entities somehow losing track of their viewer, as without a viewer an entity would be unable to notice its surroundings and would also have no way of being seen by other entities.