Rob Dawson

Candidate Number: 77514

Supervisor: Dr. Nicholas Collins

Music Informatics BSc

School of Informatics

University of Sussex

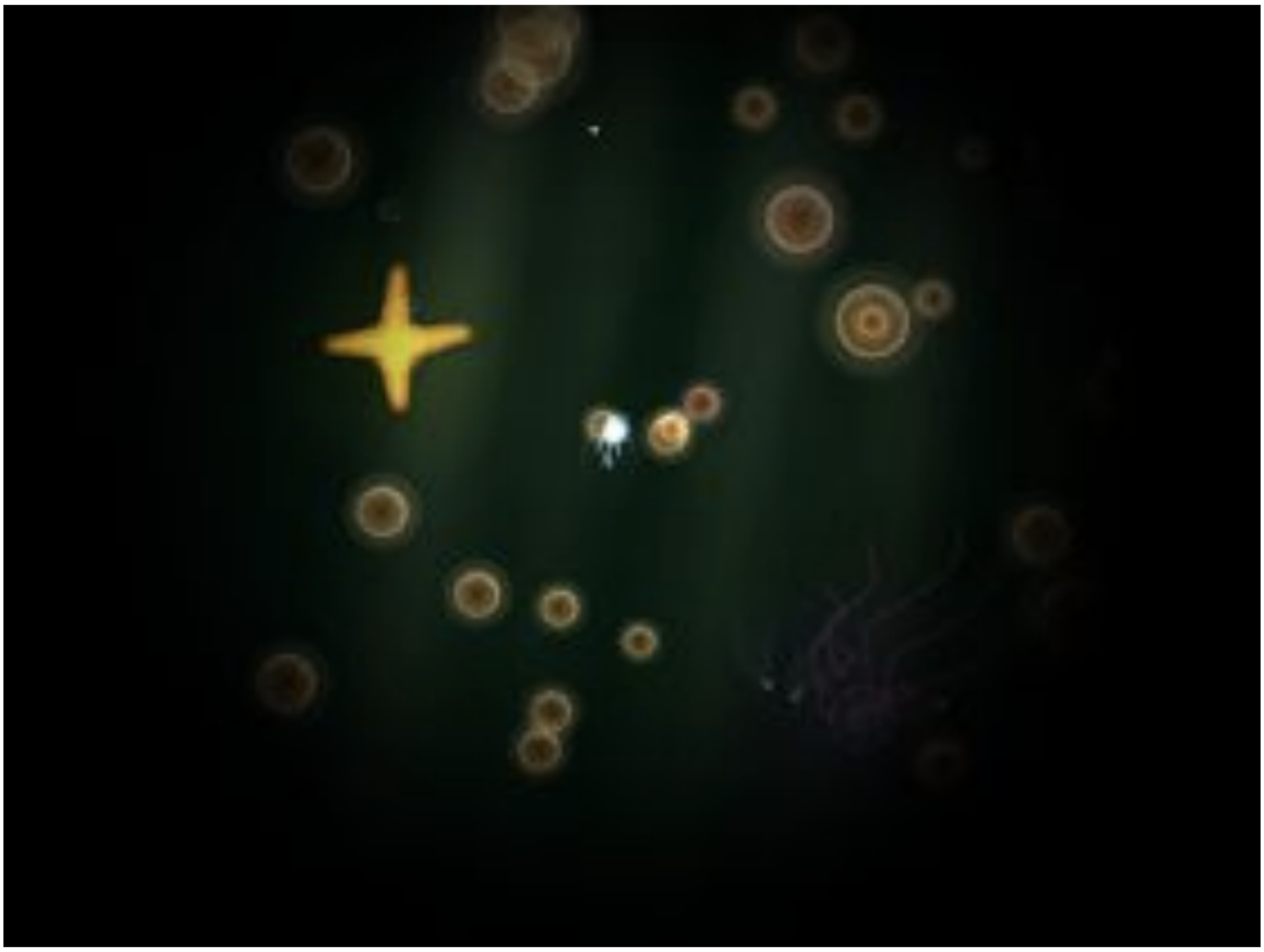
2013

Acknowledgements

Thank you to everyone who helped test *CELL*, who provided the essential user input and made this project what it is today. Thanks to the minds behind all those great open source projects: Cinder, OSC, Processing. Thanks in particular to housemates for putting up with repetitive watery ambience and stochastic sub tones for a year, and to Nick Collins for providing an entertaining and rewarding Music Informatics degree.

Abstract

This report describes the design, implementation and evaluation of a Generative Audiovisual Game '*CELL*', built using the Cinder creative coding framework and the SuperCollider music programming language. The game is a multi-directional explorative scroller in which the user controls a small entity around a procedurally generated world. The user interacts with different entities within the environment, causing musical outputs. Every play-through of the game is intended to be different, allowing the visual and sonic output to be unique for each user. The game's design, implementation and evaluation was performed in conjunction with user evaluation, which guide the game's creation towards a usable and user-friendly experience. Various unit tests ensured that *CELL* was as optimised and efficient as possible.



Contents

1 - Introduction 6

1.1 - A Brief History of Game Music 6

1.1 - The Music of *CELL* 7

1.3 - Intended Users 8

1.4 - Report Structure 8

2 - Professional Considerations 9

3 - Requirement Analysis 10

3.1 - Requirements 10

3.2 - Existing Music-based games 11

4 - Program Implementation 13

4.1 - Cinder Application Structure 13

4.1.1 - Class Overview 14

4.1.2 - Entity Management 15

4.1.3 - Entity Movement: The Braitenberg Vehicle 17

4.1.4 - Navigating the Game World 18

4.1.5 - Swimmer Collision and AI 19

4.1.6 - Entities 21

4.1.6.1 - The Player 21

4.1.6.2 - The Friendly 21

4.1.6.3 - Tails 22

4.1.6.4 - Dust 22

4.1.6.5 - Springs 22

4.1.6.6 - Feelers 23

4.1.6.7 - Eggs 25

4.1.6.8 - Spores and Sparks 26

4.1.6.9 - Plankton 27

4.1.7 - Environment 27

4.1.7.1 - Beams 27

4.1.7.2 - Bubbles 27

4.1.7.3 - Splashes 28

4.1.7.4 - The Mask 28

4.1.7.5 - The Surface 29

4.1.8 - Procedural Content Generation (PCG) 30

4.1.9 - Global Functions 31

4.1.10 - Open Sound Control 32

4.2 - SuperCollider Application Structure **33**

4.2.1 - Chords **34**

4.2.1.1 - Chord Process Example **35**

4.2.2 - Global Elements **36**

4.2.3 - Direct Elements **36**

4.2.4 - Continuous Beat-Locked Elements **37**

4.3 - Version Control and Workflow **38**

4.4 - Compiling to Application **38**

5 - Evaluation **39**

5.1 - User Evaluation **39**

5.1.1 - Interim User Evaluation **40**

5.1.1.1 - Interim User Evaluation: Questionnaire **40**

5.1.1.2 - Interim User Evaluation: Results **41**

5.1.1.3 - Interim User Evaluation: Conclusions **42**

5.1.3 - Beta Testing **43**

5.2 - Unit Tests **44**

5.2.1 - Processing vs. Cinder **44**

5.2.2 - Unit Test 1: CPU% in Cinder **46**

5.2.3 - Unit Test 2: CPU% in Cinder **47**

5.2.4 - Unit Test 3: CPU% of Draw methods **48**

6 - Conclusion **49**

6.2 - Future Developments **50**

7 - References **52**

8 - Appendices **54**

8.1 - Project Log **54**

8.2 - Entity Overview **59**

8.3 - Running *CELL* **61**

1 - Introduction

This project involves an interactive generative experience *CELL* in the form of a video game, whereby a users will provide simple trackpad/mouse-based input that result in the formation and development of sound objects. These inputs and outputs may have direct relationships, where a visual element would directly correspond to a sound object, or more abstract relationships, where visual elements and gameplay styles correspond to broader changes in composition. Gameplay is *CELL* is *synthesthetic,* where "sound is contextualized in terms of its haptic and visual associations" (Collins 2013, p39) utilising a constant feedback loop between user input and musical/visual output. As *CELL* built from scratch, without the use of a pre-existing game engine, this project essentially involves the creation of a game engine *as well as* a game within that engine. In the context of this report, *CELL* is referred to as a 'game', although it is really more of an interactive audio-visual experience, and does not really feature many components and goals that are commonly associated with games.

CELL is based loosely on organisms in nature. The user controls an organism (from here called the "Player" for explanation purposes) who moves naturally around a 2D Environment. There are other organisms who exhibit naturalistic behaviours (Non-Player-Characters). Gameplay is physics based - organisms and game objects should move naturally and obey physical laws of collision. While the game is not a simulation of life itself, the gameplay should *appear* natural as the naturalistic aesthetics are enjoyable and lend themselves to a good User Experience.

Algorithmic Composition itself is a computationally-based approach to music. It involves "composing by a means of formalizable methods" (Nierhaus 2009, p1). Algorithmic Composition, while commonly associated with Computer Music, has it's roots in the musical dice games of Mozart. Today, algorithmic composition has made it's way into video games, where instead of simply playing a predesigned composition as background to the game, the game's soundtrack will evolve and change depending on gameplay. Supercollider takes parameters from the game (from Cinder, via the OSC messaging system), and abstracts these into methods that create music. There are a number of levels of abstraction here - from user input, to the game process, to the algorithmic composition process. This 'abstraction' is the key on obtaining interesting and stimulating audio-visual relationships. The link between Supercollider and Cinder, between audio and visuals is one of **transcoding** -, it "provides a way to express continuity between forms, data, and ideas." (Reas & McWilliams 2010, p79)

1.1 - A Brief History of Video Game Music

The earliest video game music was extremely limited, with data restrictions only allowing single-voiced instruments to play entire soundtracks. These soundtracks were often extremely simple and repetitive, as game designers would 'tack-on' sound after a game's gameplay and visuals were created. *Space Invaders* (1978) introduced the idea of a continuous soundtrack, and even featured a form of reactive music, whereby the soundtrack's tempo would increase as the player progressed (Collins 2008, p2). Sound is an important aspect of video games, "to generate and reinforce the feeling of success" (Kamp 2009, p5), but music in games has a different purpose. Games, like films, convey a narrative, often which characters, settings and events. Music is a narrative tool that can be employed “to control and manipulate the player’s emotions, guiding responses to the game” (Collins 2008, p133). In essence, the soundtrack of a video game can have a direct affect on gameplay. Unlike films, however, games are non-linear, and involve choices which can change the direction, pace, and emotion of a narrative. The audio of games should reflect this non-linearity. As Norbert Herber suggestions in *‘Pac Man To Pop Music’* (Ashgate, 2008):

“Unlike a film, where a narrative dominates the trajectory of a viewer’s experience, games present a player with potentialities. Their interactions with the environment, other characters and ultimately the ‘rules’ of the game can lead to a wide variety of possible outcomes.”

Today, hardware is much more advanced, allowing fully reactive orchestral scores and thousands of samples to be stored inside a game in full quality stereo. This has allowed audio in games to take on far more complex forms, with adaptive and procedural soundtracks that respond to live user input.

1.2 - The Music of CELL

CELL features a dynamic soundtrack which changes directly through user input. It occupies a form of video game music, where music becomes “part of a game’s rules as opposed to (just) it’s narrative” (Kamp 2009, p6). A number of existing games have employed this approach to varying degrees. Games such as *Music 2000* (Codemasters, 1999) are essentially Digital Audio Workstations inside video games, and the gameplay goal is purely to create new music. Rhythm-based games such as the *Guitar Hero* series also create music, but their gameplay goal lies in hitting buttons in the correct rhythm in order to reach the end of each song.

While *CELL* has no obvious gameplay goal - there are no levels to complete or enemies to kill - the generation of sound is what inspires users to explore and continue playing. If it is assumed that the purpose of gameplay in CELL is to create music, then it would be more suited to a music game category, games that “are built around music, rather than the other way around” (Collins 2008, p171). Audio is an extremely important aspect of the game, and is arguably the entire ‘point’ of playing. Unlike video games where music’s purpose is that of films - a storytelling-aid (albeit a dynamic one), *CELL*’s music interactive. in *Playing Sound* (MIT Press, 2013), Karen Collins suggests that “the experience of interacting *with* sound is fundamentally different in terms of the listener/player’s experience from that of listening *to* (noninteractive) sound” (p7). As audio plays such a large role, it is hard to categorise audio elements in terms of normal game audio. *CELL* is, however, more abstract than purely music-based games such as *Guitar Hero* or *Singstar*, and seems to lie somewhere in between Music-Games and games that have an emphasis on music.

There are a number of ways that sound is created through the actions of the user. Some user actions will have direct sonic results, while other collected user actions will result in more broader changes in music. There are a number of stochastic elements which will evolve the music (outside of user control). This multi-layered approach to sound generation hopefully creates an interesting and varied musical output, while retaining a feel that the user has control over the output.

Video games such as flower (thatgamecompany, 2009) and Journey (thatgamecompany, 2012), for example, are essentially ‘ambient’ adventure games with simple objectives - collecting items and interacting with the environment. The music in these evolves as the gameplay changes, but maintains a natural compositional arc. In *CELL*, these ideas have been extended by bringing the music to the foreground - with large numbers of options and larger extensive algorithmic composition libraries, beyond sample triggering. It is a program that offers simpler user input that provides more complex musical background.

1.1 - Intended Users

The intended user base for *CELL* is broad, so that users of all ages and technical abilities would be able enjoy it. This means that gameplay would have to be simple, and the visuals and audio be presented in a clear fashion. In order to avoid gameplay which would be challenging or promote antagonistic gameplay, there will be no end-game, nor enemies or health/damage systems or a specific goal. Instead, the music and visual aesthetics should be enough to invite the user to play and to continue playing. User tests will be conducted at a variety of user bases to ensure that every user can successfully play the game.

1.2 - Report Structure

The following report will investigate the requirements of developing an audio visual game that is usable and enjoyable. It will look at previously released music-based games and investigate the role of music in video games. The report will examine the implementation of such a game, with reference to Visuals, Audio and program structure. Finally, it will document the user evaluations performed throughout development, along with unit tests to assure maximum CPU and GPU efficiency. Finally a conclusion will give an overview of the project in relation to the goals outlined.

2 - Professional Considerations

It has been necessary to take into account is the use of existing libraries, such Cinder. Cinder is licensed under the 'Simplified BSD License', an open source license which allows open and closed source software to be created with it. *CELL* uses no third-party libraries and instead uses only core Cinder code. The exception is the Open Sound Control (OSC) CinderBlock, a packaged-up collection of classes which allow Cinder to use OSC. These classes (the ones inside the 'osc' folder in the XCode project) have been written by Hector Sanchez-Pajares and are free to download from github here: https://github.com/cinder/Cinder/tree/master/blocks/osc

The algorithm for spring mechanics (explained in detail in sections 4.1.6.4 to 4.1.6.7) is adapted from a Processing sketch by Casey Reas and Ben Fry, found in the 'Learning' section of the Processing website: http://processing.org/learning/topics/chain.html. The algorithm has been converted into C++ for use with Cinder, and has been extended in a number of ways to provide more complex behaviour.

The potential for ethical issues to be raised in User Testing, but evaluation was limited to fairly simple Questionnaires and informal interviews. There was no issue of user safety, and all user testing was recorded anonymously.

As music and sound samples (.wav files) are either created by me, or from freely distributed sample packs. All visual artwork is entirely original and created by me, and either drawn using OpenGL calls in Cinder, or through artwork created in Adobe Illustrator.

3 - Requirement Analysis

As previously mentioned, the creation of *CELL* involved creating a game engine from scratch. While it would have been possible to create a similar end-product to *CELL* using an existing 2D engine (with pre-programmed physics, collisions, sprites), I felt that it was an interesting and beneficial challenge to approach the creation of game from the low-level perspective that Cinder provides. Avoiding existing game engines allowed me to hand-craft the specific behaviours that I had designed. In order to fully understand the gameplay requirements of the project, it was important to understand the intended user base.

3.1 - Requirements

As the project has no defined task to complete - in the sense that there is no specific problem that the project is trying to 'solve', the requirements are somewhat open-ended, being subjective to the user's experience. User evaluation will be vital in guiding the design process. However, it was necessary to lay out some requirements for *CELL.* These are based around the "common priorities" ideas put forward by Julian Gold in *Object-Oriented Game Development* (Pearson Education Limited, 2004).

First and most obviously, the program needed to be usable, and it was be necessary to take into account usability paradigms in order to make sure that users can successfully navigate the program and understand what actions will produce relevant output. Most importantly, this was that "the game should never crash, especially if that would result in the loss of a player's efforts" and that the playable character "must respond (perceptibly) immediately to my changing the physical controlling state" (Gold, 2004, p16). These were the simplest requirement to implement, as there are no menus or interface elements to interact with other than in-game objects. Essentially, it the game needed to be usable *but* allow interpretation. Designing usability also involved carefully structuring the game in time, so that users would be introduced to concepts and techniques gradually, and would avoid being overwhelmed by a number of different gameplay elements at once. Another usability issue was that of distributing: packaging the program into an application so that other people could easily play the game. While this compromised the usability, I felt that distribution was not a great concern of mine and I do not consider issues with distribution to be an important part of the project.

Secondly, *CELL* needed to be enjoyable. The User Experience should have been rewarding and challenges should be avoided. The software should have provided an environment for the user to be freely creative. Visual and audio feedback should be relevant to user input, and while surprising results were good (particular in aurally), there should not have been 'shocking' results. The user experience should ultimately have been pleasant and the user should not have needed to be an experienced video-game player in order to enjoy it.

Lastly, it was important to keep in mind is that the program should *not* be a Digital-Audio Workstation. Users should not decide they want things to sound a certain way and be able to create it. Instead, music creation should be a natural, physical process. Inputs should be *abstracted* from the musical generation. While users may see a link between interacting with a Game Object and hearing a particular Sound Object - the 'aim' of the game should lie in a full, balanced experience between audio creation, visual feedback, and user input.

3.1 - Existing Music Based Games

There are a number of existing games which employ a similar emphases on music and generative music creation. These are described below with reference to how they relate to the *CELL*.

Journey

Journey (thatgamecompany, 2012) is an ambient game on Playstation 3 in which the user controls a character in a 3D world. The game features simple controls: only the directional movement of the analog stick, and one other button affect the gameplay. Journey features breathtaking environments through which the user explores, collecting certain items in order to overcome puzzles and obstacles. The game's soundtrack, composed by Austin Wintory, features different composed elements that are triggered by the actions of the user. *Journey* is clever in the sense that it retains a grander compositional arc while allowing reactive audio that seems "totally seamless, so that it feels like the music is unfolding in real time, as if being written by an unseen (and very fast!) composer" (Wintory 2012, interview with thesixaxis, available at http://www.thesixthaxis.com/2012/03/15/interview-journey-composer-austin-wintory/ (accessed 10/4/13)). The game's only button triggers a short melodic burst from the Player-Character - the only form of communication in the occasionally multiplayer game. Journey is a prime example of thatgamecompany's philosophy of emotional-based games - games that are "clearly developed with aesthetics and artistic expression in mind" (Novak 2012, p66). It is this approach (although on a smaller scale) that I hope to apply to *CELL*.

image source: http://thatgamecompany.com/games/journey/

Red Dead Redemption

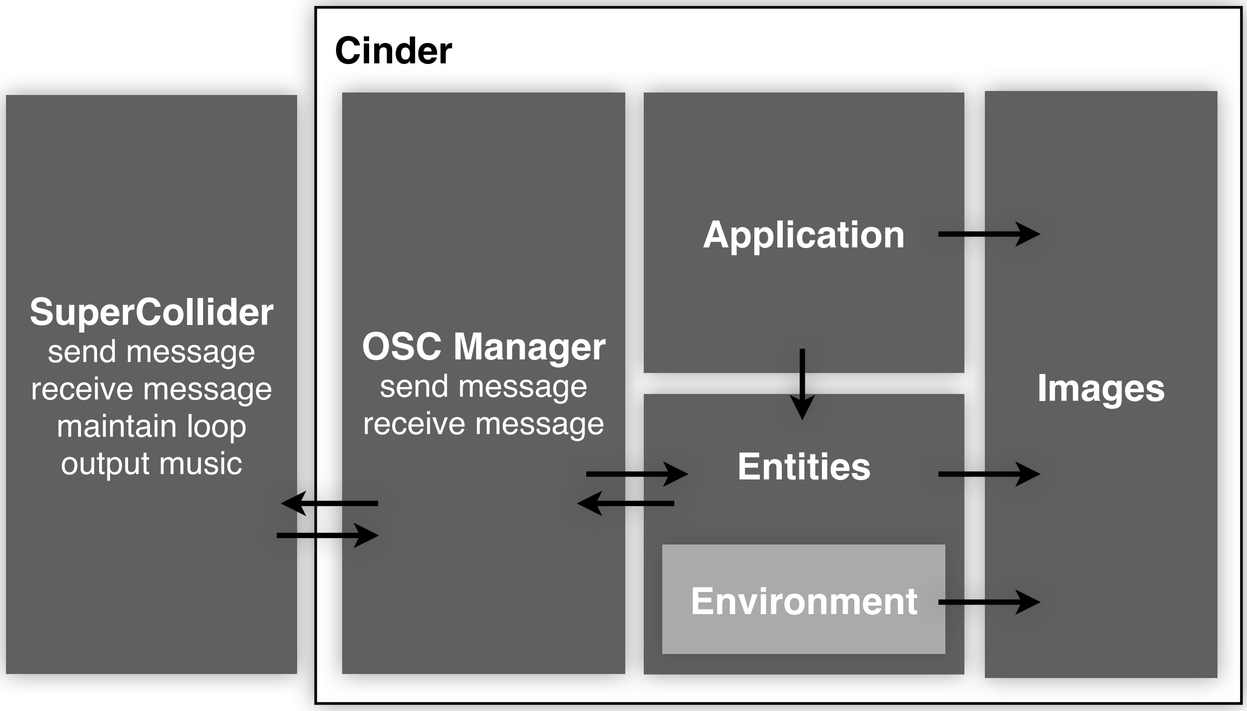
Red Dead Redemption (2010, Rockstar Games) is an open world adventure game set in the frontier of America in the early 1900s. It is notable in that's score was "written in stems versus actual songs with a clear beginning, middle and end" (http://www.rockstargames.com/reddeadredemption/features/soundtrack, accessed 25/3/13). These stems, composed in the same and key and the same tempo, could then be performed at any point during gameplay and still give the appearance of a deliberately composed soundtrack. This form of adaptive audio, and *CELL* features a similar (although more algorithmic and less composed) approach to soundtrack. All music in *CELL* stays in-key, and a repeating loop makes sure that the music remains in-tempo.

image source:http://www.rockstargames.com/reddeadredemption/img/en_us/world/1280/sidewindercreek.jpg

4 - Implementation

CELL utilises an Object-Oriented approach to game development. This played to C++'s strengths and suited the game - essentially an environment of different entities that often share various attributes. The project implementation is split into a number of sections. The majority of the programming took place within Cinder, and as *CELL* is built from scratch without the use of a pre-existing game engine, it was necessary to build a working game engine *and then* build a game within that engine. These two processes naturally crossed over, but keeping the Cinder project well organised, and by following standard programming conventions, meant that it was (and will continue to be) easy to add new features to *CELL.*

CELL Project structure



4.1 - Cinder Application Structure

Cinder applications, by default, are set up with a top-level class (which inherits the AppBasic class) with three main methods: `setup()`, which is called upon the program starting, and `update()` and `draw()`, which are called continuously. The `update()` and `draw()` methods will get called a number of times a second, specified at program initialisation. *CELL* runs at an intended 30 frames per second. This section of the report will detail the implementation of various aspects of the Cinder project. Firstly, there will be a brief overview of the 28 different classes and their relationships. Details of core game mechanics will be detailed, followed by a break down of classes including behaviours, draw methods, and gameplay mechanics. Lastly, there will be an description of the Procedural Content Generation system. For further details on any of the above, the C++ code is fully commented.

4.1.1 - Class Overview

There are a number of different types of classes in *CELL.* Firstly, there is a global namespace which contains some global variables and some often-used functions. Entities are the classes which are often instantiated in greater numbers, and are the objects which the Player would interact with. Many of these entities would share an Entity Superclass (eventually inheriting from the top-most superclass *GameObject*). Entities sometimes reference other classes which contain specific behaviours but are not entities themselves, these are called Entity Attributes. Environment classes are entities that do not interact directly with the player but are used for aesthetic purposes. Lastly there are singletons. A Singleton is a class “that logically has only a single instance in any application” (Gold 2004, p81). These singletons are generally managers of other objects, and their purpose is to control and manage the behaviour of their ‘children’ objects.

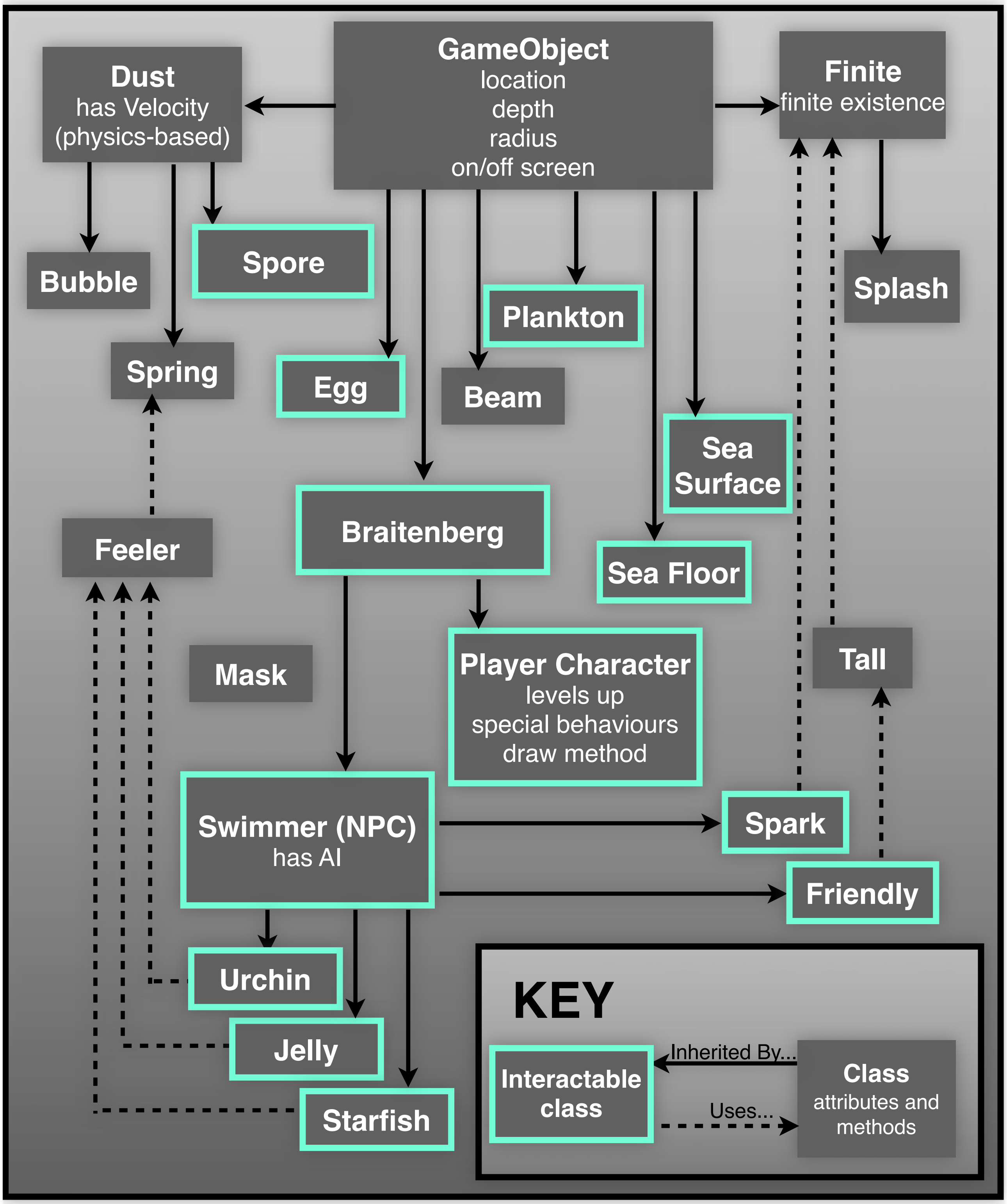
Class	Purpose
Application	Contains the top-level update and draw loops, application settings and splash screens. The core class onto which all Cinder applications are built
Globals	A namespace which is referenced by most other classes. Contains global variables and commonly-used functions
EntityManager	Updates and draws collections of entities, and controls procedural generation of content. Organises interaction between entities
Environment Manager	A subset of the Entity Manager, dedicated to controlling non-interactive elements (environment)
OSCManager	Controls incoming and outgoing messages to SuperCollider
Images	Loads images from the Resource folder, and stores references to images for other classes
GameObject	The highest level of superclass for in-game entities. Has local and global co-ordinates and a depth variable (for parallax scrolling)
Braitenberg	Moves towards a location. Any entity that moves inherits this.
Player	The user-controlled entity
Dust	A GameObject with velocity vector. For any objects with realistic physical movement
Spring	A more complex type of ‘Dust’ that responds to collisions and is pulled towards a target position
Feeler	Collections of ‘Springs’ connected in a line
Swimmer	Any moving non-player controlled entity. Has AI that avoids other entities
Urchin	A deep-sea tentacled entity with Feelers
Spark	Spawn from Spores. Follows the player and interacts with other entities

Class	Purpose
Jelly	Jellyfish with hanging feelers
Spore	Small Dust objects that react to collision and hatch Sparks. Found in groups
Egg	Large membrane that slows movement. Contains unhatched Friendlies
Plankton	Commonly found food source for entities, which come in variety of different shapes
Friendly	Hatch from Eggs. Hunt for plankton and grow in size
Starfish	Non-moving entities that curl up when in proximity to the player
Tail	A selection of Finite objects which are connected by lines. Used by Friendlies and Players.
Finite	A GameObject that has a stored lifetime and finite existence. Used for Splashes and in Tail objects
Splash	A finite with a draw method: an expanding stroked circle which fades in opacity
SeaSurface	Complex system of Path2Ds that move in waves. Marks the upper boundary of the game environment
Bubble	Dust objects which move towards the surface
Beam	Randomly generated beams of light emanating from the sea's surface
Mask	Obscures the game world, acts as the field of view of the player. Tightens in deeper waters as if the player can't see as far.

Key:							
	Namespace	Singleton*	Entity Superclass	Entity	Entity Attribute	Misc	Environment

4.1.2 - Entity Management

The game's entities (that is any particular in-game object) are managed in the EntityManager class. This class is a form of *Manager,* a class that "takes responsibility for creation, deletion and - more to the point - the co-operative (synergistic) behaviour between the classes it manages." (Gold 2004, p95) Within the class, entities can be created, destroyed, and the relevant entities' `update()` and `draw()` functions are called. Inter-entity behaviours also happen within this class. The entity system is structured so that nearly every type of entity inherits from a SuperClass: GameObject. GameObjects have a location (global and local), a depth, and a radius. All moving entities follow Braitenberg system of movement (explained in *Entity Movement and Collision* section), and all moving, non-player entities inherit from the Swimmer class, which accounts for collisions.



All entity collections are updated in similar patterns:

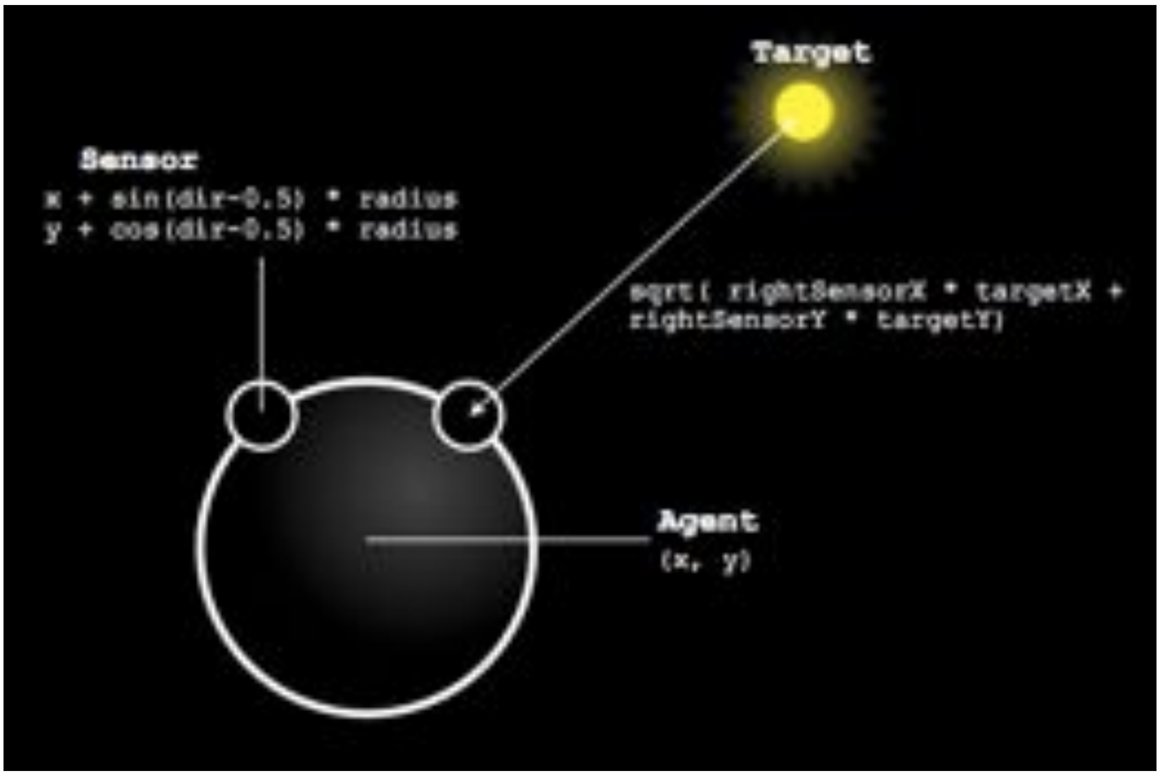
```
-If the entity is far enough away from the Player
      -delete entity
      -remove from any relevant collections
-Otherwise:
      -Update entity, with collisions, destinations, and other behaviours
      -If needed, add environmental elements (splashes or bubbles)
      -If entity is on screen, reset that species' counter
- Increment species 'last seen' counter (for Procedural Content Generator
```

4.1.3 - Entity Movement: The Braitenberg Vehicle

Entities inside the game exhibit some limited behaviour: searching for food, or following certain entities, or simply exploring, and so all NPC entities that move use behaviour based on braitenberg vehicles, “simple robotic designs that are capable of displaying surprisingly life-like behaviours from very simple circuitries of sensors and wheels” (Komosinkski and Adamatzky 2009, p100). In *CELL,* the Braitenberg vehicles have two sensors which receive stimulus from a target location, and the distance between each sensor and the target location is measured. Depending on which sensor is closer, the entity’s opposite motor’s speed is increased, and the adjacent motor’s speed is decreased. The motors shift into the direction the organism faces. Finally the organism can be moved forward in whichever direction it is facing. This causes a very natural movement which essentially mimics how simple organisms would behave towards, for example, a light source. It is an example of *emergent* behaviour which is “behaviour that has not been explicitly programmed into a system or agent” (Pfeifer and Bongard 2007, p85).

In terms of aesthetics, the entity over-shoots the target direction at each turn, causing a sinusoidal ‘wiggle’. This seems to effectively mimic swimming-movement in real animals. Using Braitenberg-style movement fulfils the user requirements of *CELL:* the Player is usable (the entity responds to mouse movements as it should - the entity moves towards it’s target), and having a level of user experience (movement is *aesthetic* and enjoyable to observe). This is the main benefit of using Braitenberg vehicles - they “feature anthropomorphic qualities we assign to moving objects” (Reas and Fry 2007, p473).

The Braitenberg in CELL


Target
Sensor
x + sin(dir-0.5) * radius
y + cos(dir-0.5) * radius
sqrt(rightSensorX * targetX +
rightSensorY * targetY)
Agent
(x, y)

4.1.4 - Navigating the Environment

The game features a free roaming element, meaning that the 'window' through which the user observes the world appears to follow the character around, as if looking at only a small part of an entirely bigger world (otherwise, the game could take place entirely within a static boundary). As the game is exploration based, the user will want to move to places other than the starting window. In order to do this, every Game Object has a set of **Local** Co-ordinates (their location on screen for the user, the actual drawn location of any circles, lines, etc) and **Global** co-ordinates, which describe their actual location in the game world. The difference between the local and global co-ordinates is defined by a set of **offset** co-ordinates. The offset is defined by the user's movements around the game world.

To calculate the offset vector, first the 'd' values are calculated - proportionate to the distance between the character's **local** coordinates and the centre of the screen. This 'd' coordinate set will be added to the offset vector, a global variable which is accessed by all GameObjects. The 'd' part of this equation means that the player can move around *inside* the User Window before the Window begins following the character. The 'd' is modified by `0.1f,` an *easing* value, meaning that the player will gradually move back towards the centre of the screen. Easing "is a technique for moving between two points. By moving a fraction of the total distance each frame, a shape can decelerate (or accelerate) as it approaches a target location" (Reas, Fry 2007, p239). In appearance, this looks as if the camera gently moves to catch up with the player. This fulfils my second user requirement: that the game should be aesthetic and enjoyable to look at. Keeping the character fixed rigidly in the centre of the screen gave an unappealing aesthetic.

```
offset += 0.1f * (hero->local - Vec2f(getWindowWidth()/2, getWindowHeight()/2));
```

Now the offset can be applied to any game object, so that local co-ordinates can be calculated from global co-ordinates. As all GameObjects follow these rules, this process can lie high up the class system. A Vector of (windowWidth/2, windowHeight/2) is applied to all local co-ordinates, so that the point of parallax lies in the centre of the screen, and not at the top left corner. Parallax scrolling is an important part of the way *CELL* looks, and it gives the illusion of depth. It "adds the appearance of depth to 2D games by scrolling across the background images more slowly than the foreground images." (Kelly 2012, p263) In the case of *CELL*, this is achieved by the 'depth' parameter that all GameObjects have. A depth of 1.0 is an unmodified scrolling speed. A lower depth means scrolling will be slower - and the object will be appear to be further away. A higher depth likewise gives the impression that the object is nearer. Most of the interactions in *CELL* will take place where a depth is (or near to) 1.0. Many entities in the game will be created at random depths, however, to give the illusion that the game world is three-dimensional.

The GameObject global -> local process:

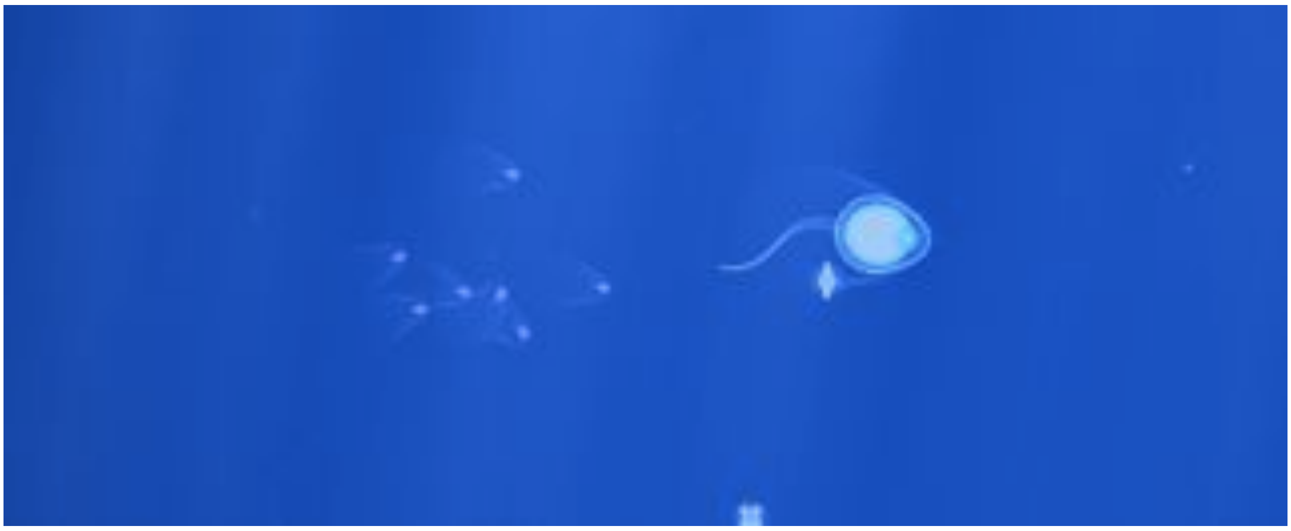
```
local = global - offset;
local *= depth;
local += Vec2f(getWindowWidth()/2, getWindowHeight()/2);
```

4.1.5 - Swimmer Collision and AI

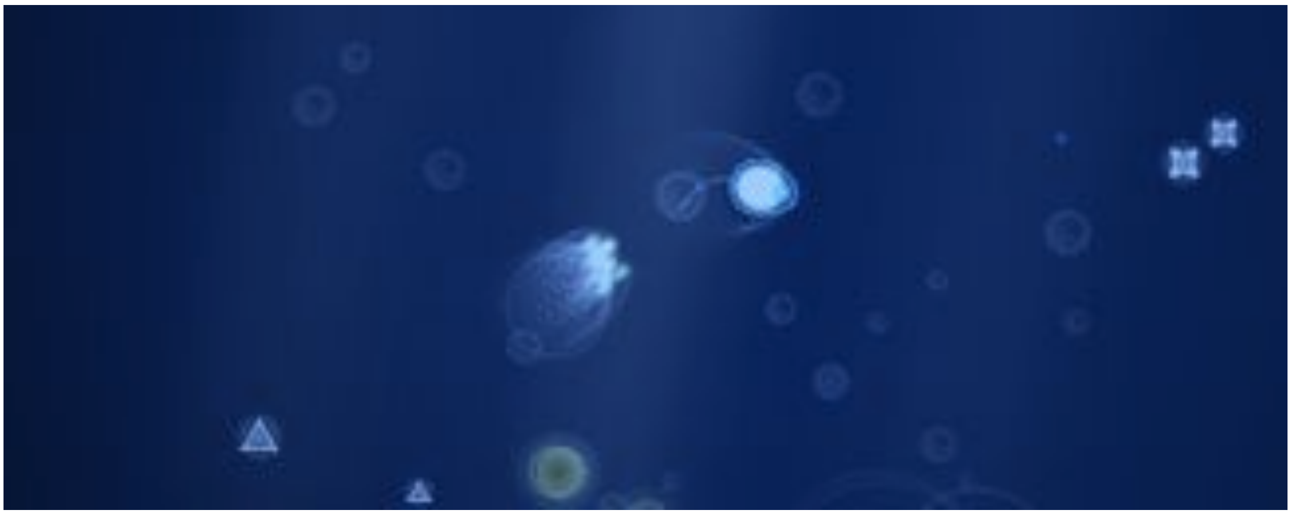
Most entities will steer to avoid pieces of the environment. Each NPC Swimmer (a Braitenberg with AI behaviour) has a target location which it will constantly try and move towards. Every update of the Swimmer will contain a `moveTo(targetLocation)`. This target location *itself* eases between locations, through the Swimmer's `setDestination(Vec2f)` method. The result of this is that the Swimmer (a Braitenberg) will move naturally towards it's destination *and* the destination itself gradually moves between locations. There are no sudden changes in direction, resulting in natural and fluid movement.

Each swimmer has an `updateTarget()` method which checks to see if the swimmer is in space. The method takes a collection (a vector) of GameObjects as it's parameter, and will check if the entity is enough distance away from all of these. A collision occurs if "the distance between there centers is less than or equal to the sum of their radii" (Kelly 2012, p149). If not, the target's location will be adjusted (moved a small amount in a random direction), and the process repeated. This collection of collision-able GameObjects (`vector<GameObject*> colliders`) is a combined collection of all Urchins, Starfish, Sparks, Eggs, Jellyfish, and the Player itself.

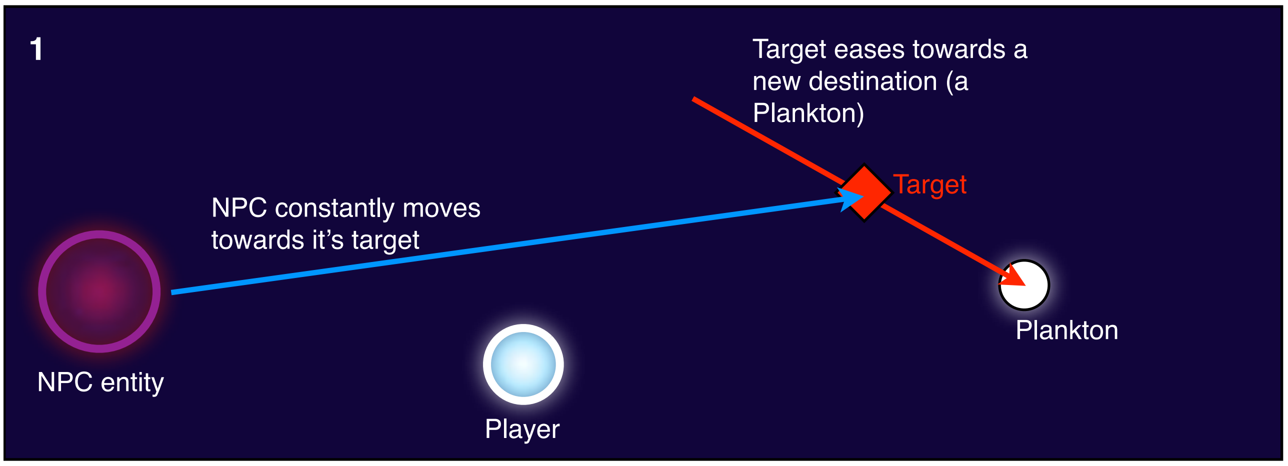
The Sparks following the player are 'avoiding' each other while continuing to follow the player.



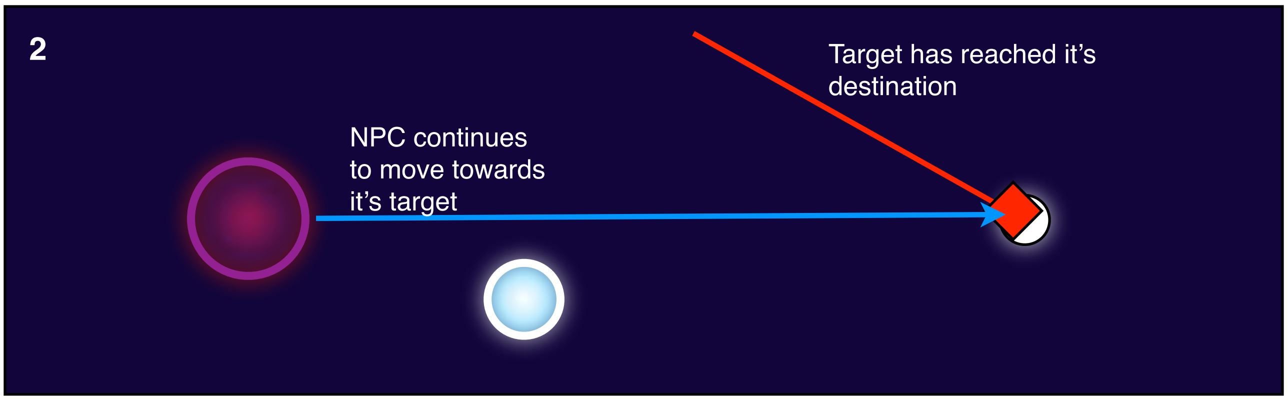
The Sparks behaviour <u>without</u> the collision system - clumped together and occupying the same space



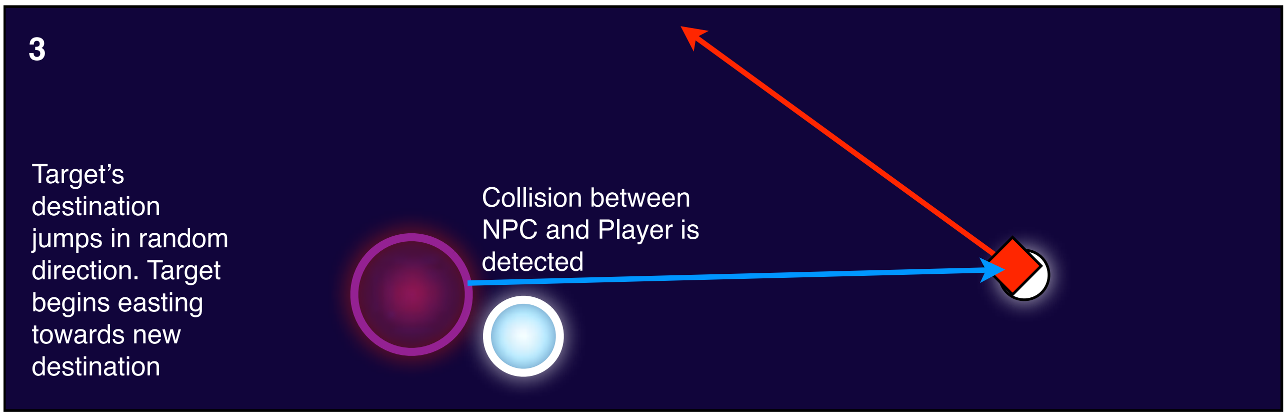
Entity Collision System walkthrough



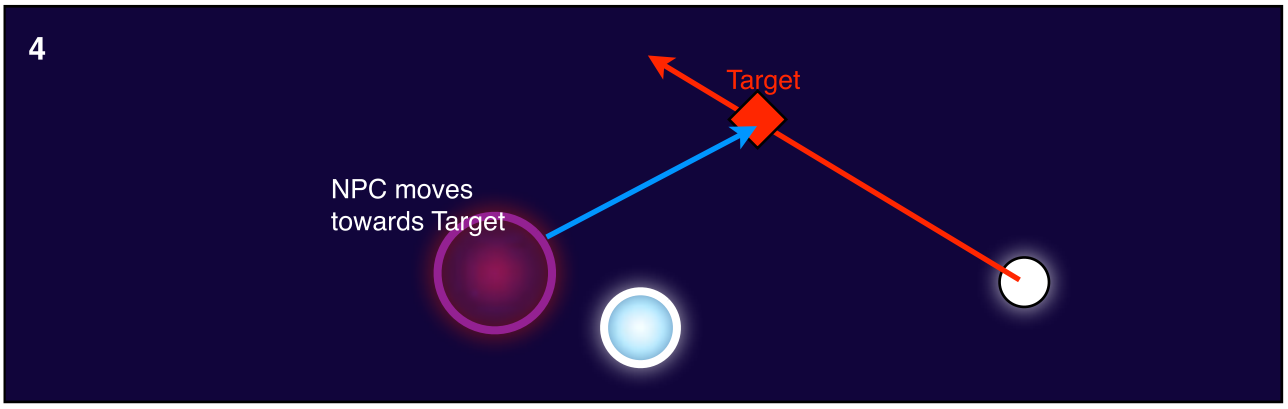
1
Target eases towards a new destination (a Plankton)
Target
NPC constantly moves towards it's target
Plankton
NPC entity
Player



2
Target has reached it's destination
NPC continues to move towards it's target



3
Target's destination jumps in random direction. Target begins easting towards new destination
Collision between NPC and Player is detected



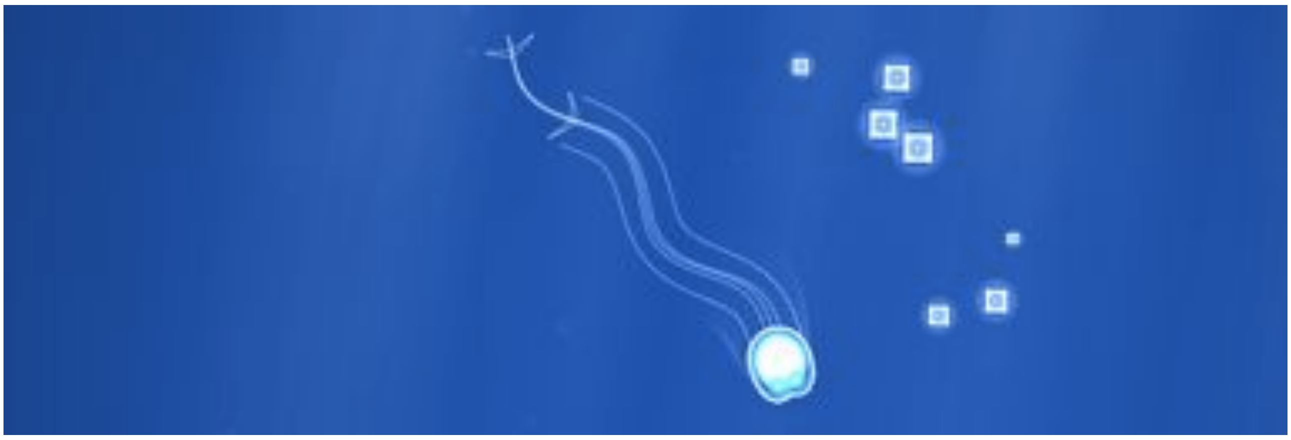
4.1.6 - Entities

This section will detail each of the separate Entities in *CELL.*

4.1.6.1 - The Player

The Player is a type of Braitenberg entity with special attributes and behaviours. Firstly, instead of artificial behaviours, the player is User-controlled. The mouse location is 'globalised' (see section 4.1.9) and sent to the Braitenberg's `moveTo(Vec2f)` method. The player's other unique attribute is it's ability to 'level up'. After eating enough plankton, the player will go through a phase of growth, emitting a white glow and creating splashes around it for a short time. As this happens, the Player's body will evolve and tail will grow. A number of different texture files are stored in *CELL,* and the Player's body shape will gradually change over time.

A highly evolved Player



4.1.6.2 - The Friendly

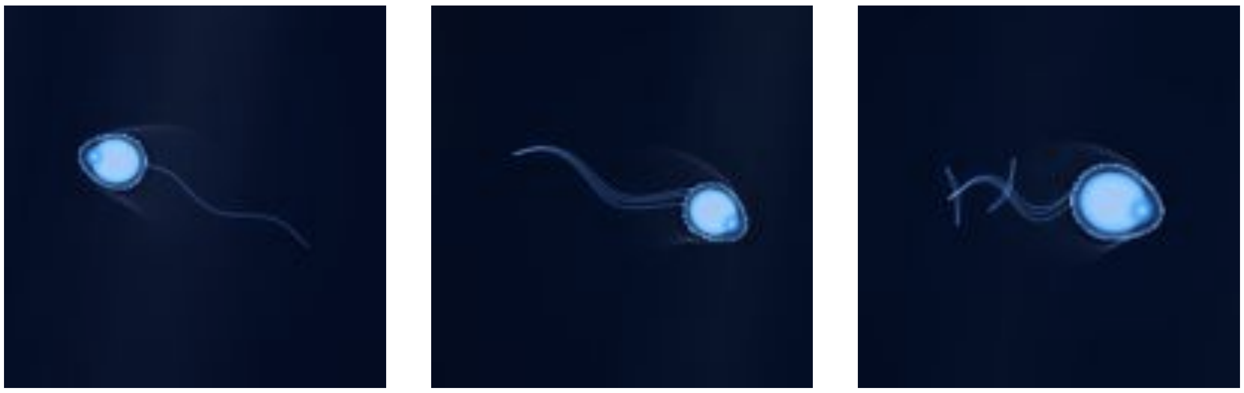
Friendlies are Swimmers that appear similar to the Player, but have their own behaviour. Friendlies hatch from eggs (see section 4.1.6.4), and will, once born, search for nearby plankton. Friendlies will, as with all Swimmers, avoid other swimming entities. Like the Player, Friendlies will grow in size over time, although this is an automatic process that happens regardless of the amount of food eaten. Technically, the Friendlies will always move towards the first plankton in the Plankton collection (the one at position "0").

A pair of green Friendlies

4.1.6.3 - Tails

Tails are objects attached to certain types of entities, with the appearance of sinusoidal appendages. Technically, Tail objects are collections of Finite objects that are connected by a line. At every update cycle, a tailed entity 'leaves behind' a Finite object. The tail object stores a collection of these, updating each of them and destroying them when they have existed for a predefined amount of time, which corresponds to the length of the tail. With this set of Finite objects (which each have a location, a direction, and a lifetime), the Tail can draw out patterns along them. There are a number of types of tails, which entities can grow over time. Tail type and fins are decided upon Tail instantiation, and can be updated later on. The player, for example, will begin life with a single slim tail, but later get a wider tail with several sets of fins.

Tail types



The tails sinusoidal appearance is due to the way in which the Braitenberg system of movement works. Because the entity is wobbling from side to side, it's back point (directly opposite the direction it's facing) moves from side to side, so the Finite objects that the Entity leaves behind are placed in a sinusoidal pattern. This is another benefit of using the Braitenberg system of movement.

4.1.6.4 - Dust

'Dust' objects are GameObjects with a velocity vector. The location of a Dust object is constantly updated by the values stored in the velocity vector. The velocity vector's values are reduced at every tick which mimics the resistance received by a object in water.

4.1.6.5 - Springs

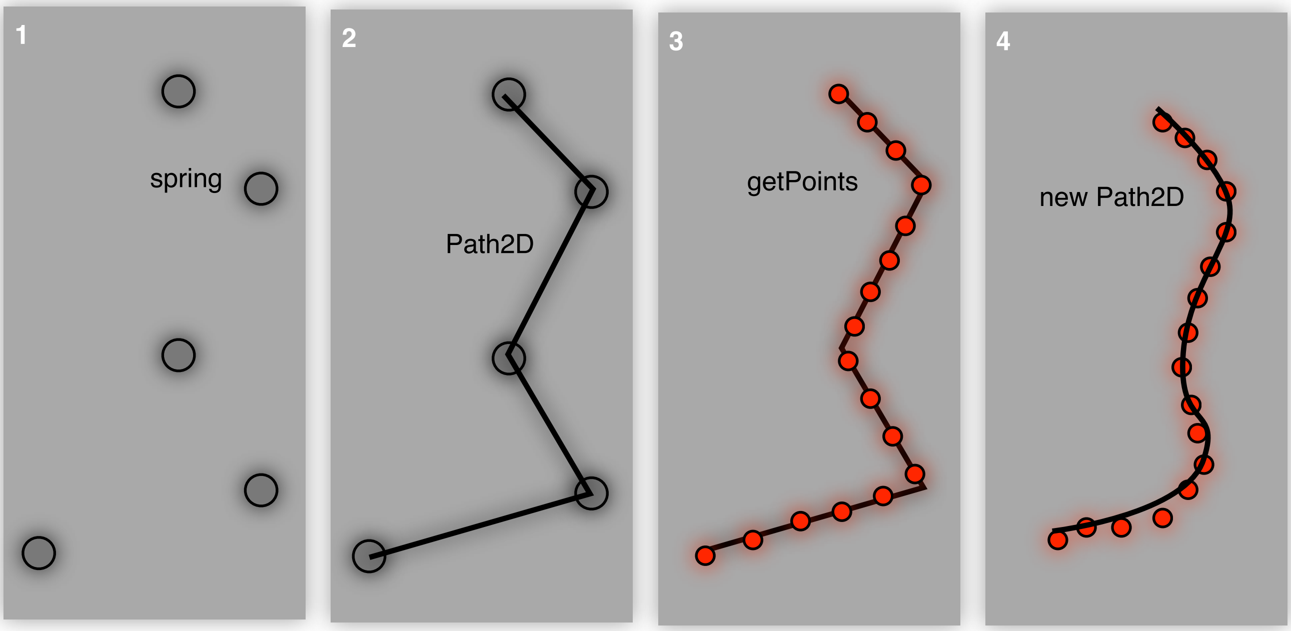
Springs are a subclass of Dust objects and feature a number of extra variables and more complex behaviour. Springs will, when updated, move towards a target location. Due to a Spring's acceleration vector, the Spring will constantly overshoot its target by a gradually diminishing amount, as the velocity vector is gradually reduced at each update cycle. This movement is affected by three tuneable parameters: damping, stiffness and mass. Springs also have the ability to collide with other objects, and can be pushed out of position.

4.1.6.6 - Feelers

A number of different entities use *Feelers*, which are essentially single rows of Spring objects. Each spring is updated against the target location of the preceding Spring, resulting in a string of objects who's velocities are passed along to each other (in one direction). Other objects (such as the Player) can then be collided against the Feeler, which will call the necessary collision methods inside the Spring objects. With the right parameters, these strings of springs have a very pleasant aesthetic which mimics the movement of soft-body materials in the physical world. In *CELL*, Urchins, Grass, Jellyfish, and Starfish use Feelers.

Drawing Feelers

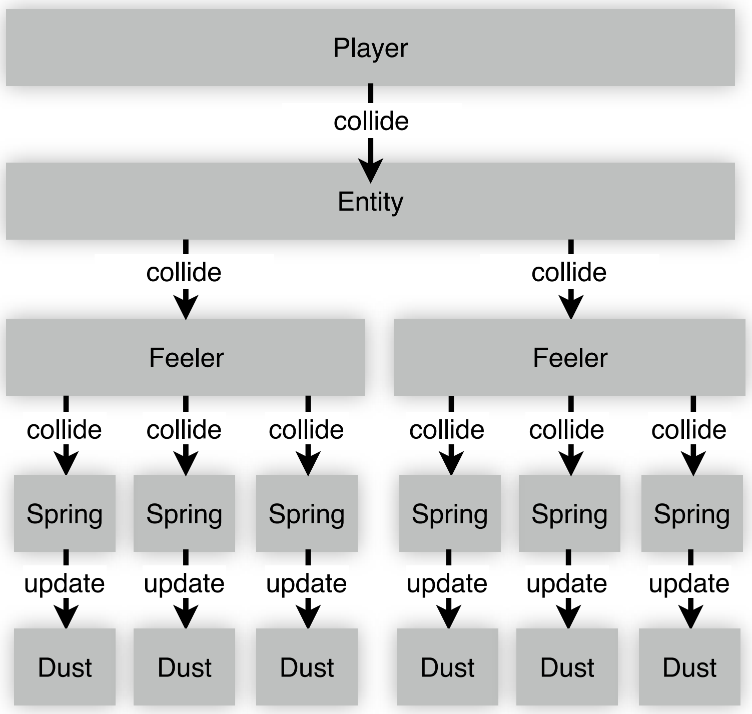
1 - Begin with a set of Springs. A spring is a type of GameObject and so has a global and local location.
2 - Draw a line along each spring's local position, from the first to the last.
3 - Divide this line into a number of equal length sections, the number of which depends on an accuracy constant.
4 - Average each of these points with it's neighbours, twice, to achieve a smoothed set of points
Draw a new Path along these points. This is the path that entities will draw.



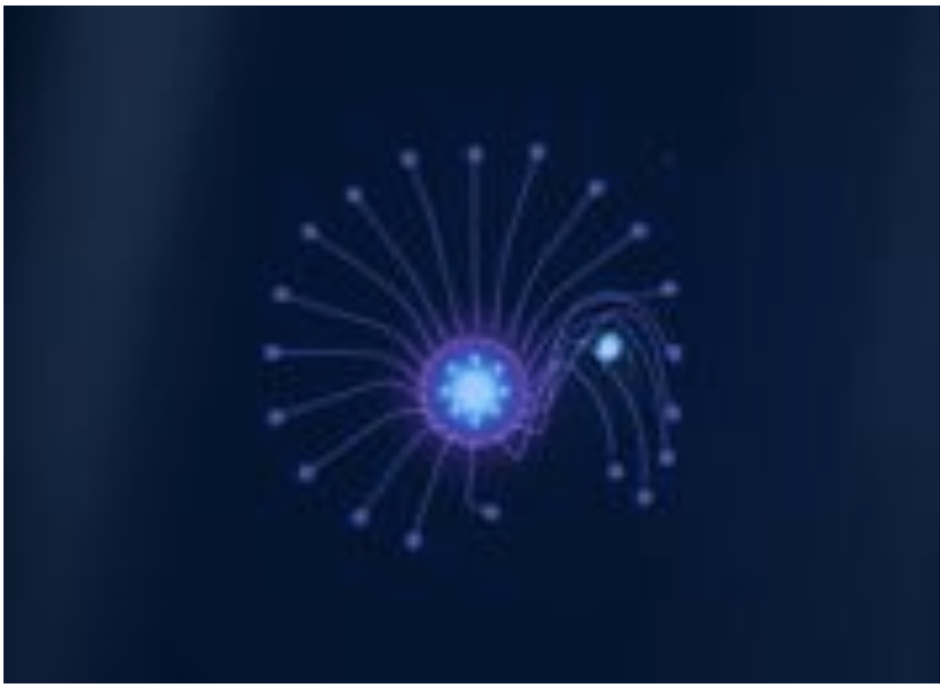
Feeler Collision

In game, Feelers have a collide method. When this is called (by the EntityManager), the feeler calls the collide method of each of it's springs, passing on the location value. If this in-game location is close enough to the spring, the spring's velocity vector is updated so it will move away from that location.

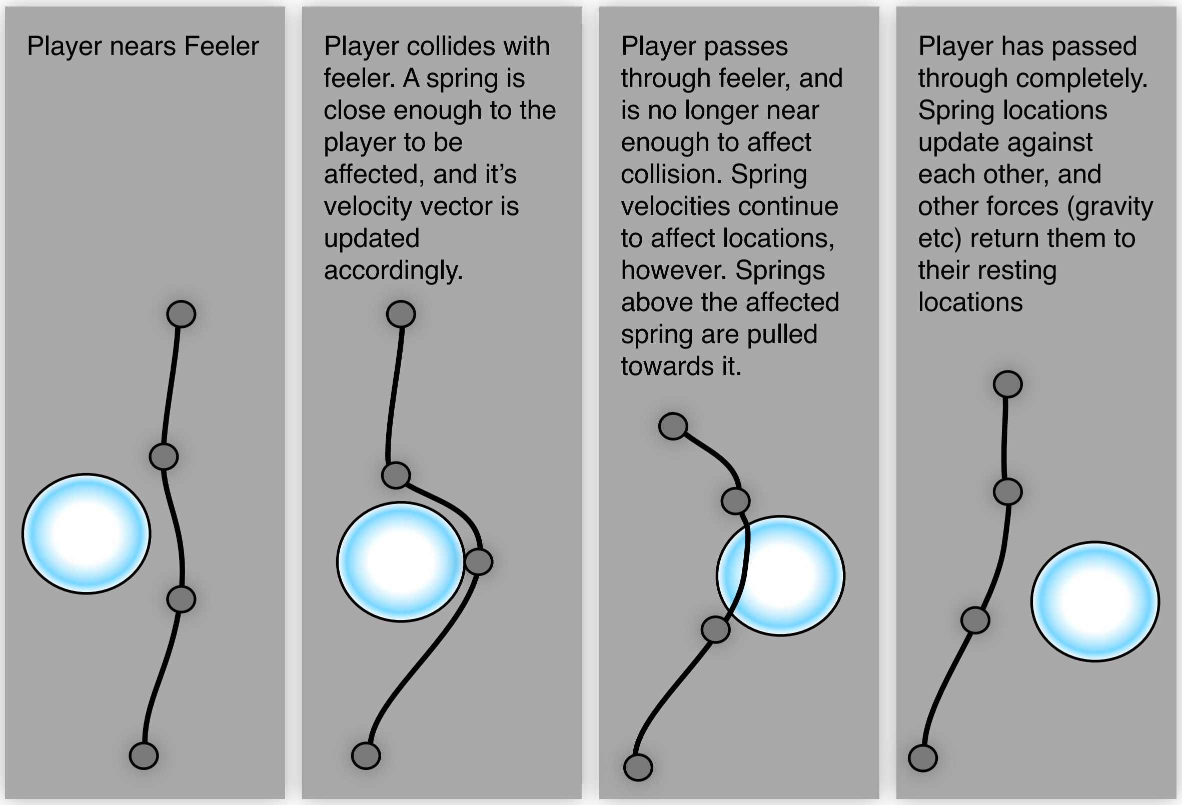
Collisions passing through objects



Feelers also have the ability to be affected by forces other than specific collisions. This is the `addForce()` method, which takes a vector as it's parameter. This method is used to give the feelers a realistic underwater behaviours. Grass on the sea floor for example, are constantly supplied with an upwards-pointing force, to mimic the grass pointing towards the surface. Urchins and Starfish have outwards-pointing forces. Without any constant force, Springs inside the feelers would simply sink back towards their parent spring, and these constant forces give the feelers an appearance of retained shape. Almost all feelers are constantly influenced by a random vector, to mimic the constantly shifting currents underwater.

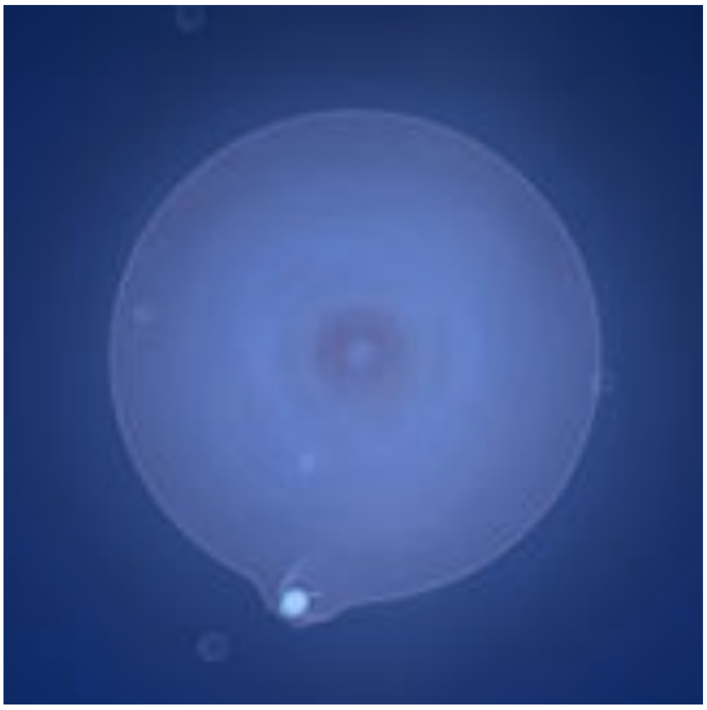


Feeler collision breakdown


Player nears Feeler
Player collides with feeler. A spring is close enough to the player to be affected, and it's velocity vector is updated accordingly.
Player passes through feeler, and is no longer near enough to affect collision. Spring velocities continue to affect locations, however. Springs above the affected spring are pulled towards it.
Player has passed through completely. Spring locations update against each other, and other forces (gravity etc) return them to their resting locations

4.1.6.7 - Eggs

Most entities which use Springs (Urchins, Jellyfish, Grass and Starfish), use them in the form of Feelers - a single row of Springs which affect each other *only in one direction*. Eggs, however, utilise a collection of Springs where each Spring affects the location of it's two neighbours, so that the flow of velocity goes in both directions. the results is a circular membrane that appears to 'wobble' when passed through, with ripples passing along each direction and eventually subside.

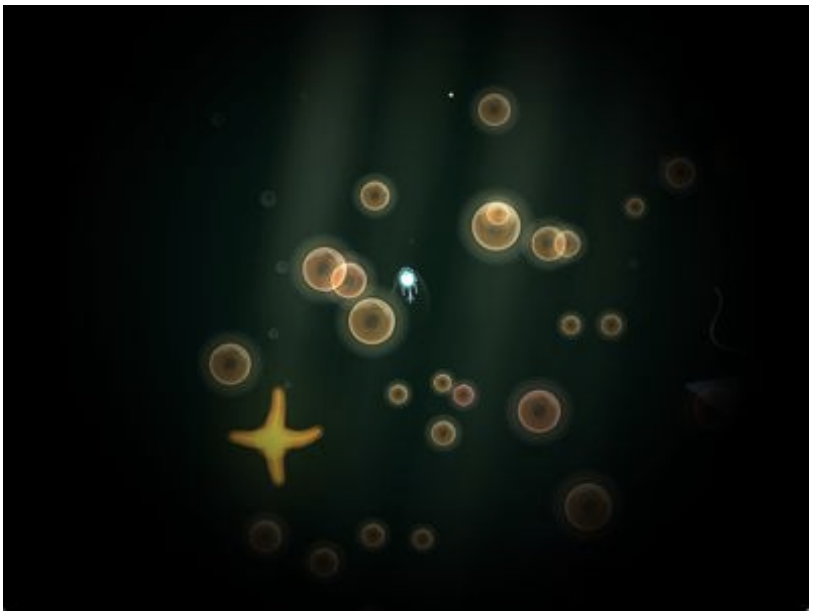


4.1.6.8 - Spores and Sparks

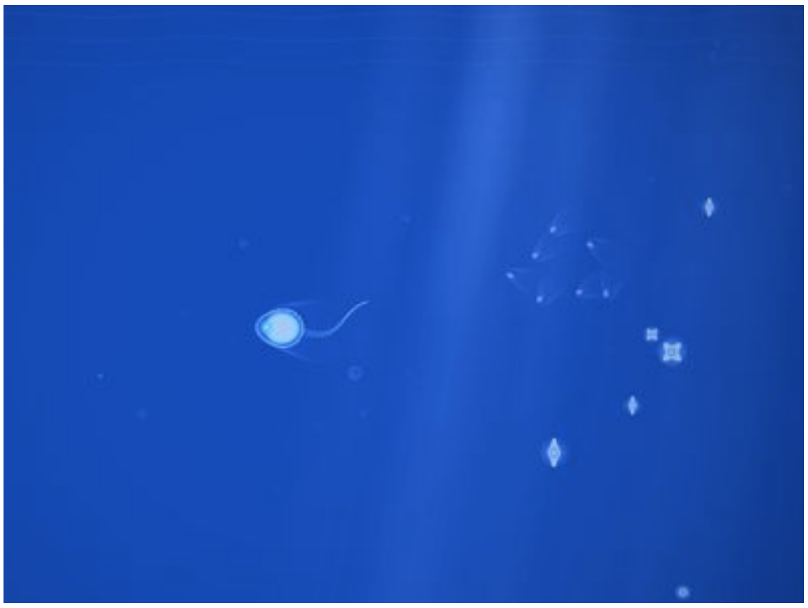
Spores are subclasses of Dust objects, and have the appearance of circular coloured balls. Spores are often found in clusters of 10 or more, and are one of three different variants: orange, green, and purple. Spores react to collision, and if the Player reaches a close enough proximity to the Spore, it's velocity vector will update accordingly, pushing it backwards away from the player. Repeated colliding with the same spore will result in it hatching, revealing a new Spark entity.

Sparks are small coloured Swimmers which continuously follow the player, and behave like other Swimmers (avoids other entities). Sparks have unique behaviour in that they will deliberately interact with entities that the Player is nearby. In terms of gameplay, this means that more interactions (and therefore music) will happen as the Player collects followers. The colour of the Spark is the same as the Spore from which it hatched (see section 4.2.4 for details on how the Sparks sound).

The Player amongst a group of orange Spores



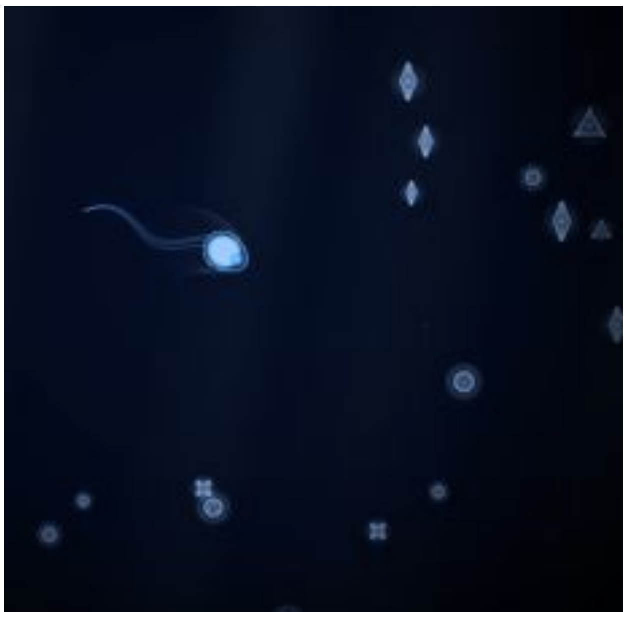
The Player being followed by a group of Sparks



4.1.6.9 - Plankton

Plankton are simple GameObjects that are found throughout the game environment. They are static entities which are eaten by other entities. Eating plankton creates a splash and a number of bubbles, as well as musical accompaniment (see section 4.2 for details). Plankton come in variety of different shapes and sizes, which are randomly generated upon creation. Plankton are the main gameplay focus of *CELL,* as they resemble item collection objectives commonly found in video games.

the player and a variety of Plankton types



4.1.7 - Environment

The Environment is a subset of *CELL's* Entities, and are managed by a separate EnvironmentManager class. The environment is made up of procedurally generated elements (which are generally temporary), and a few constant elements.

4.1.7.1 - Beams, Bubbles and Splashes

The procedural elements are bubbles, splashes, beams and grass. These elements operate in a similar fashion to the other interactive entities explained above, but they are purely environmental and the player does not affect their behaviour (outside of their initial creation).

4.1.7.2 - Bubbles

Bubbles are Dust objects that have their own draw method. Giving them a velocity vector allows them to be initialised with a certain force, which is useful in cases where bubbles need to appear to be 'blown out' of something (eg, when a Plankton is eaten). In this case, Bubbles are initialised with a random velocity vector.

4.1.7.3 - Beams

Beams are GameObjects with a draw method. They are procedurally generated with local positions near to the player, but with varying depths. Beams are constant environmental factors, and therefore help to give the game world a sense of depth by putting emphasis on parallax scrolling. Beams also feature fluctuating opacity values.

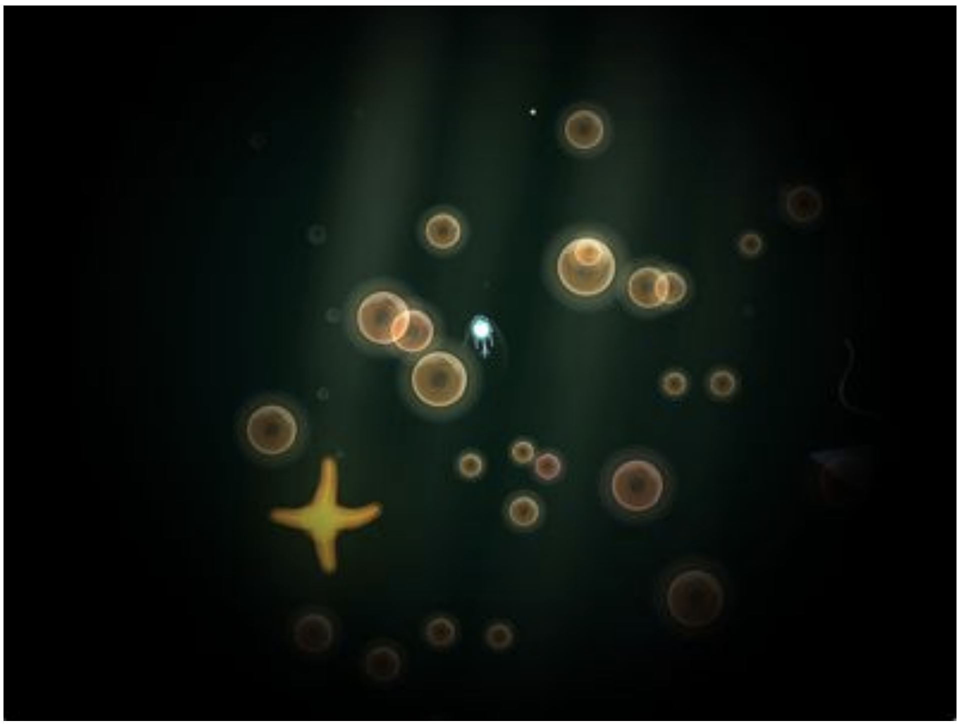
4.1.7.4 - Splashes

Splashes are Finite objects with their own draw method. They are simply stroked circles which gradually reduce in opacity and increase in radius until their existence finishes. Moving entities such as the Player, Swimmers and Sparks constantly generate new splashes at their position. This creates the illusion of movement through water.

4.1.7.5 - The Mask

The Mask is an overlaid image that prevents the user from seeing objects outside of the field of view. This appears as opaque darkness which graduates towards transparency nearer the player. At deeper levels, the mask 'closes in' around the user, limiting it's field of view. To achieve this effect, Cinder's Frame Buffer Object class (FBO) is used. An FBO is an OpenGL object which provides an off-screen destination for drawing calls. The FBO is set up, and a mask image is drawn onto it, with alpha blending that will allot the transparent parts of the mask to show through what it's underneath. The FBO only needs to be set up once, and then the complete FBO can be drawn at any position and size with considerably less CPU usage than drawing a texture itself. The mask helps provide the user with a sense of depth, coupled with the musical effect or reaching these areas.

the mask, obscuring objects outside of the player's field of view. Beams and Bubbles are visible here, too

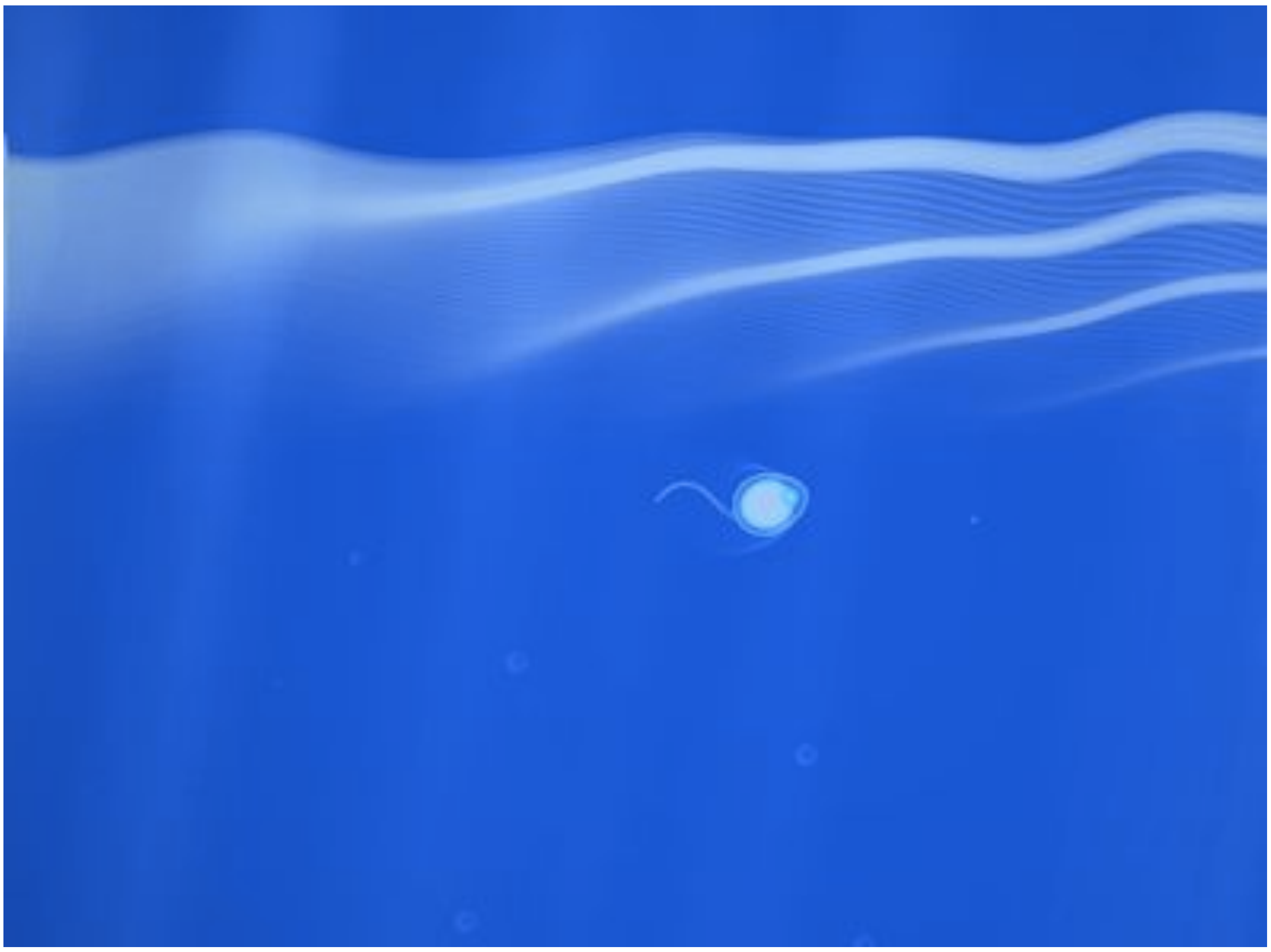


4.1.7.3 - The Surface

The Sea Surface object involves a set of GameObjects, each of which is represented by a wave-like Path2D object. The collection of GameObjects ('rows') are placed at different depths, and give the impression of a 3 dimensional surface when the player is nearby. The rows are drawn as lines from left to right, with a point at every 10 pixels. The vertical location of each point is the row's *local* position, plus a Perlin noise value (the key of which depends on the current horizontal offset), modified by some constant values (which affect the breadth and height of the Perlin values). This point is then affected by an oscillating sine function which causes the waves to move up and down.

In terms of gameplay, the Surface acts the top-most boundary for the game environment. The player can jump out of the surfarce, but it will drop back towards the water as if affected by gravity. If the Player object is above the surface, then the Y variable of it's target location (normally the mouse location) is ignored, and instead will point to a position under the surface.

the Surface above the Player



4.1.8 - Procedural Content Generation

One of the features of *CELL* is that nature's of it's content generation. It utilises procedural generation of the environment and non-player characters. Procedural content generation (PCG) in games is "the creation of game content automatically using algorithms" (Togelius, Kastbjerg, Schedl, Yannakakis, 2011 p1) as opposed to hand-crafting specific game elements. Historically, PCGs were used to save memory space, for example in the *Elite* (Acornsoft, 1984) where entire planetary systems could be stored using a seed value, rather than be stored themselves (Toeglius *et al*, 2011 p3) . Instead of loading pre-defined full levels (which would have caused a vast loading time before levels, as well as requiring more work for the level artist), *CELL* is continuous, and game objects can be loaded at any point. The result is a dynamic and ever changing game where each play through is different. It means that the game environment itself can be without boundaries as content will keep loading in front of the player. Which every direction the user takes the Player-Character, the Procedural Content Generator will keep generating new and unique environments. This process happens in a number of different ways.

For the environment (bubbles, light beams, and plankton), new entities are created after a certain number of frames, to ensure that some instance is regularly on screen. *CELL* also utilises the use of stochastic processes in the case of environmental grass, for example, which has a 0.01 chance of spawning (assuming the player is below a certain depth).

For NPCs, a number of counters keep track of the length of time since that particular type of entity has been on screen. If a Jellyfish, for example, has not been on screen for a certain number of frames, one will spawn in front of the PC (but off-screen), as long as that space is not out-of-bounds (out of the water or under the sea floor). This process assumes that the hero will keep swimming in the same direction and discover the entity. The entities themselves are spawned using random generators. Jellyfish, for example, may be one of three different body types, and have varying numbers of feelers. Most PCGs will utilise some form of constraint like this "but that within these constraints the content can vary according to some pseudo-random process" (Toeglius *et al*, 20011, p2)

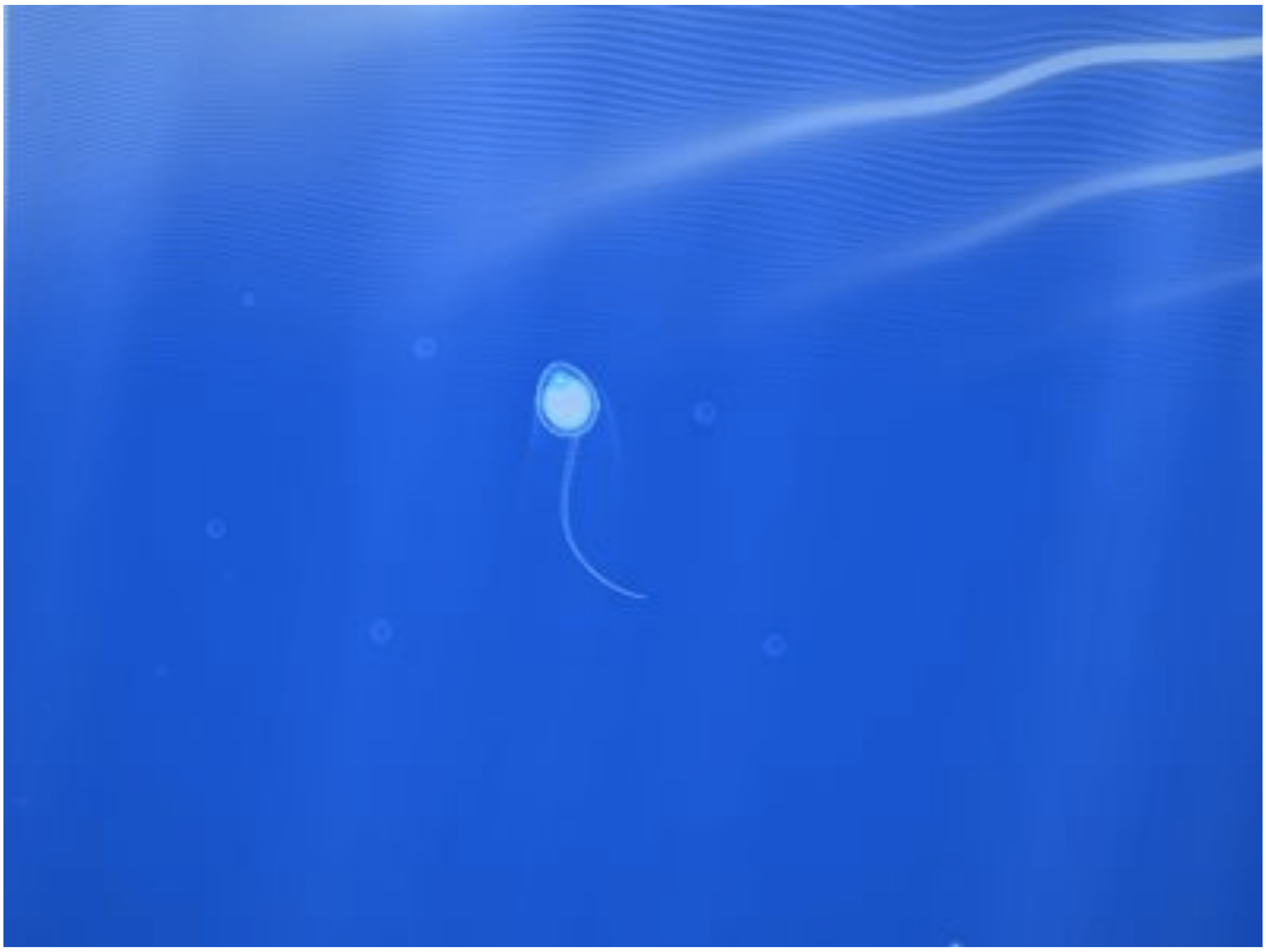
All entities are destroyed and removed from collections under certain conditions. Environment entities are generally removed once they are off-screen by more than 100 pixels, while NPCs are deleted once the character is far enough away from them. This means that the entire collection of entities remains low.

Removing objects (and their references) is performed by looping through the Entity Collection, and checking to see if any "death" condition is met (being too far away from the player, for example). If this condition is met, the object is deleted, and the reference removed from the collection. If the entity is part of the collider collection, the entity is passed to the `removeFromColliders(GameObject* collider)` method, which searches for the corresponding reference in the colliders collection, and removes it from there. As all entities inherit the GameObject class, it is possible to use polymorphism to do this.

4.1.9 - Global Functions

In order to optimally use certain functions and values, *CELL* utilises a custom namespace: ‘globals’. These are:

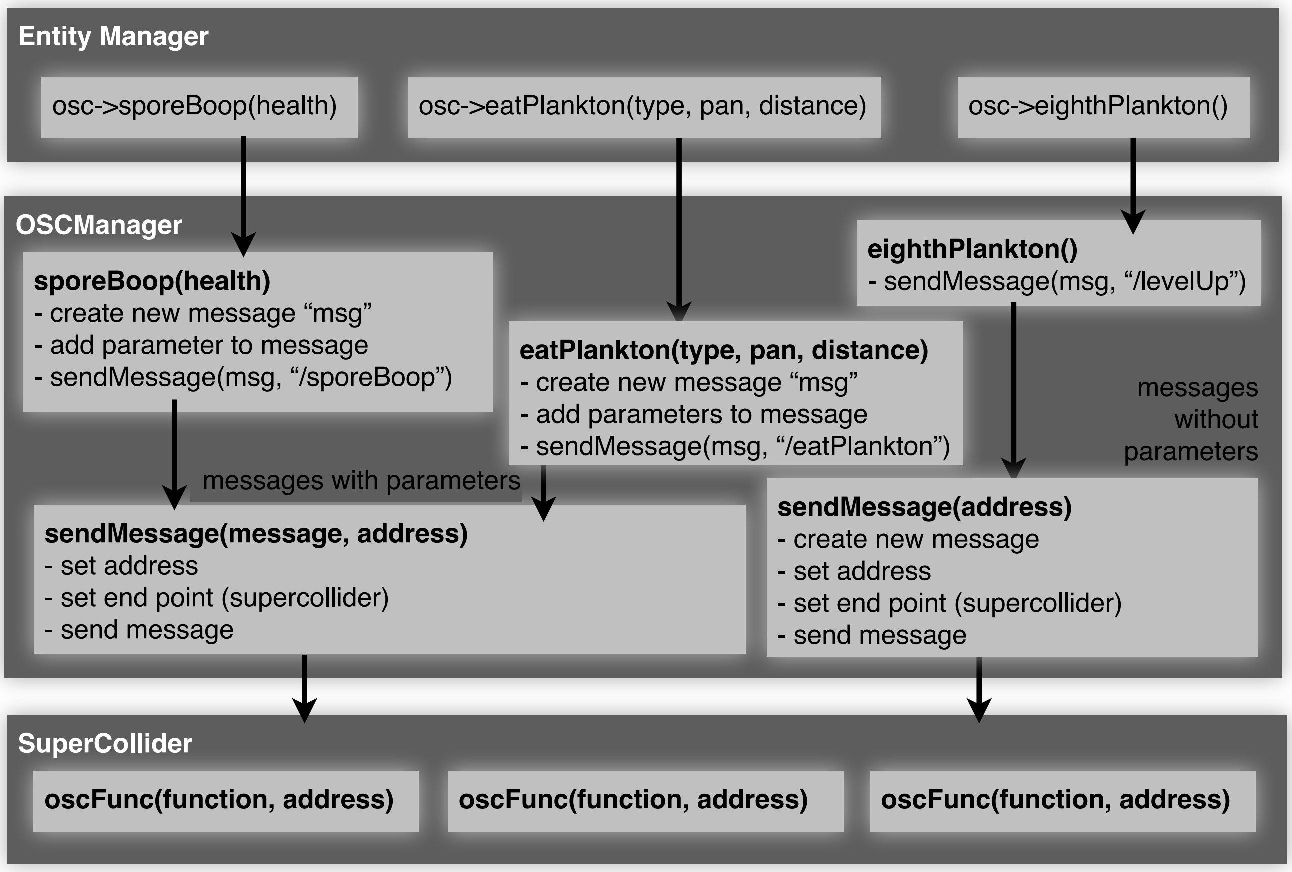
- offset value. This is the value which is constantly updated by the Player moving around. It is the distance between an entity’s global co-ordinates and it’s local co-ordinates
- Random number generators, which provide random float numbers, integers, or a random Vector value
- globalise function: converts a local co-ordinate into a global co-ordinate, depending on a supplied depth value. This is useful for a variety of visual effects. For example, if the game needs to create a set of random-depth bubbles at a specific on-screen location, passing the local location and a randomly generated depth will supply a global location to instantiate each bubble.
- distance function: simplifies the syntax for Cinder’s standard way of obtaining Euclidean distance between two points.



4.1.10 - Open Sound Control

Cinder communicated with SuperCollider using Open Sound Control, a “protocol for communication among computers, sound synthesisers, and other multimedia devices” (opensoundcontrol.org, accessed 2/2/13). When certain events happen in Cinder, the Entity Manager sends a message to special Cinder class which broadcasts a message over a pre-specified port to SuperCollider. SuperCollider will receive these messages and perform some form of musical action (the SuperCollider part of the project is detailed in section 4.2). The Cinder class that takes care of incoming and outgoing messages ‘OSCManager’ features a variety of methods that allow the EntityManager to address it. These methods will call one of two separate ‘`sendMessage`’ methods. One of these will send a given message to a given address, while the other will simply send a blank message to an address. The former is used in the case where parameters will be added to the message (such as the type of a Plankton), while the other is used when SuperCollider needs to be notified of something without any specific parameters. Keeping all OSC procedures outside of the main entity class helps modularise the program.

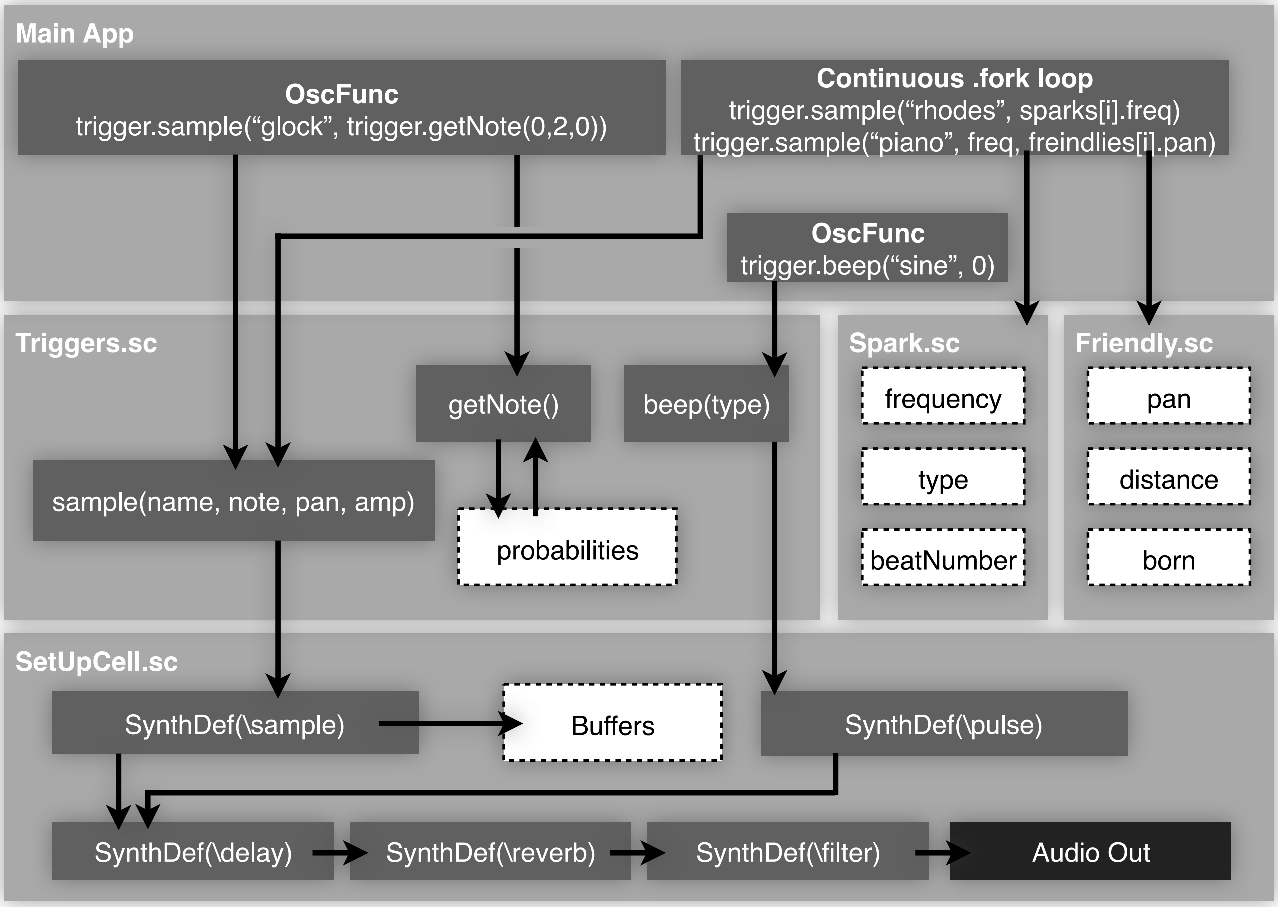
Cinder and SuperCollider with OSC messaging


Entity Manager
osc->sporeBoop(health)
osc->eatPlankton(type, pan, distance)
osc->eighthPlankton()
OSCManager
sporeBoop(health)
- create new message “msg”
- add parameter to message
- sendMessage(msg, “/sporeBoop”)
eatPlankton(type, pan, distance)
- create new message “msg”
- add parameters to message
- sendMessage(msg, “/eatPlankton”)
eighthPlankton()
- sendMessage(msg, “/levelUp”)
messages without parameters
messages with parameters
sendMessage(message, address)
- set address
- set end point (supercollider)
- send message
sendMessage(address)
- create new message
- set address
- set end point (supercollider)
- send message
SuperCollider
oscFunc(function, address)
oscFunc(function, address)
oscFunc(function, address)

4.2 - SuperCollider

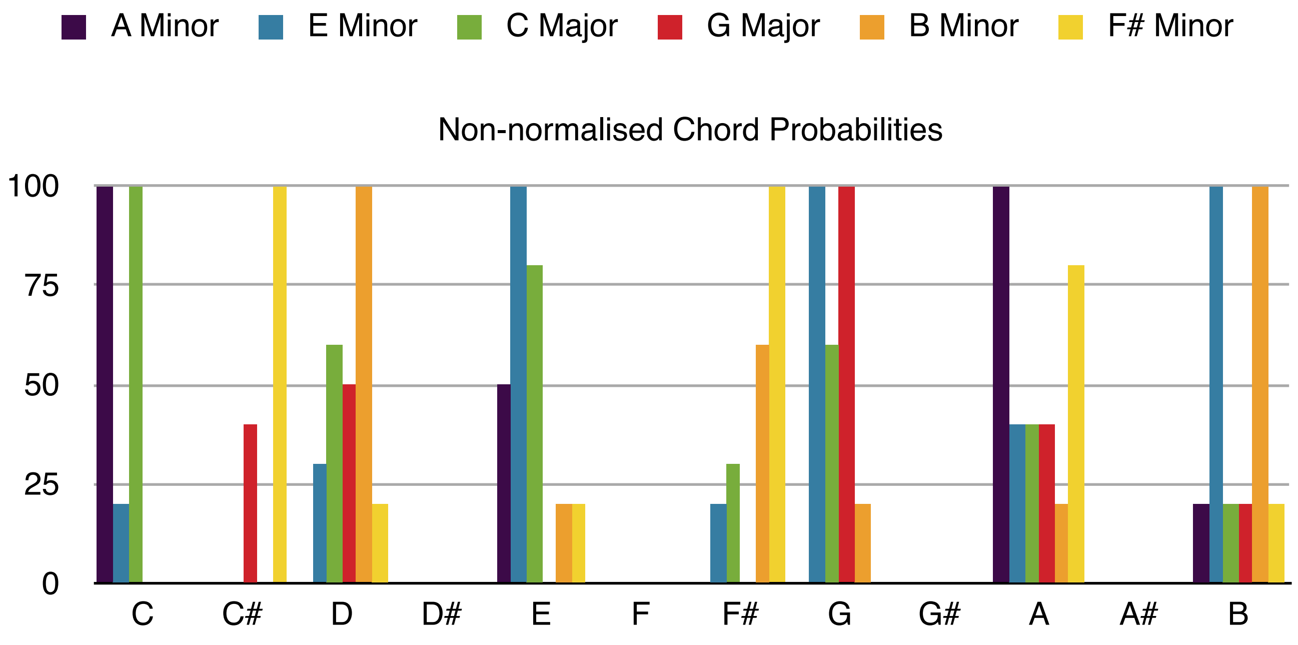
The music in *CELL* has been composed in a deliberate fashion in order to promote certain styles of gameplay. The music is built around a number of looping pads which provide a drone which do not harmonically resolve. This alludes to the music of India, where “the given as a raga (the tonic droned and then played upon for an entire afternoon), creating a meditative space without the repetition and destination of a cycling chord progression” (Booth 2005, p480). This has the effect that it allows a number of different sound objects to be performed and remain in-key, promoting adaptability in the game’s soundtrack: “Keeping a lack of harmonic resolution in the game is one way of making the music more adaptable” (Collins 2008, p151). If there is no rigidly defined structure to the music, then anything that the User does will fit into the soundtrack. Approaching the soundtrack from this perspective pushes the game away from having a deliberately composed piece of music. The ‘composer’, in this sense, is the user. *CELL*’s soundtrack is procedural, meaning that the audio changes as a direct result of the User’s input to the game. “Procedural audio is non-linear, often synthetic sound, created in real time according to a set of programmatic rules and live input” (Farnell 2007, p1).

The SuperCollider part of the project contains a number of distinct sections. A ‘set up’ class initialises synth defs, loads buffers with samples to be used. The trigger class contains calls to triggering synth defs, and information on chords and note values. The OSC receiver class accepts messages from Cinder, and contains the main loop where sound objects can be triggered. There are three types of sonic elements in-game, global elements (which evolve with out direct gameplay interaction), direct elements (which are a direct result of user action) and continuous elements (which are part of the continuous loop).



4.2.1 - Chords

Any sound that is triggered has a pitch parameter. For synths, this is a midi value attached to a `.midicps` function (converting it from midi to Hz). For a sample, the parameter signifies the number of midi notes above or below the sample's actual pitch. Whenever a sound is triggered, the pitch may either be specified or stochastically generated. Stochastic generation of pitch values takes place in the `getNote()` method in the Trigger class. The `getNote()` uses a probability table to return a midi value. These probability values are equal to chords. One chord may be active at any time, meaning that *whatever* sample is triggered, the `getNote()` method will *always* return a note that belongs in the current chord. This means that every sample will, combined, play different parts of the chord. Each chord has 12 values corresponding to the 12 notes in the diatonic scale.



The 'current chord' probabilities will be equal to one of these chords at any time, and the current chord can be switched which would result in chord progression.

The use of random number generators and probability tables stems from Greek composer Xenakis, who would apply evolving probability distributions to allow gradual changes in timbre and pitch. These stochastic processes were a "way to grasp and organise the wilderness of unruly sounds which invaded his ear but evaded his grasp". (Matossian 1986, p83). The use of stochastic process in this way allows an algorithmic quality to the music, and further abstracts the output of the music away from the user's actions.

4.2.1.1 - Chord Process example

- The current chord is a C major chord
- The Player hits a plankton (of type ‘0’), sending an “`/eatPlankton`” OSC message to SuperCollider.
- SuperCollider triggers a sine beep, using the `getNote(5,2,0)` function to get a pitch value
- getNote chooses a number from 0 to 11, based upon the current probability weights.

Probability Table for C Major

MidiNote	Pitch	Unweighted Probability	Weighted Probability
0	C	100	0.277
1	C#	0	0
2	D	60	0.166
3	D#	0	0
4	E	80	0.222
5	F	0	0
6	F#	0	0
7	G	60	0.166
8	G#	0	0
9	A	40	0.111
10	A#	0	0
11	B	20	0.055

- The `wchoose` function may land on 4, of which there was a 0.222 chance of happening. This is the lowest E natural note (an E-5, or 10.3 Hz)
- This value is added to 12 times the ‘starting octave parameter’, in this case 12x5 = 60. 60 + 4 = 64. An E0 (329.63Hz)
- This value is added to 12 times a random integer in the range of “`numberOctaves`” parameter. In this case (2), this means that the resulting note will either be an E0 (without any octave added) or an E1 (with one octave added).
- There is no “`startingNote`” parameter, so the resulting note does not need to be offset by any amount. This is used in cases where the raw sample is not tuned to a C note.

4.2.2 - Global Elements

The elements are continuous and may be affected by greater-scale gameplay results. Sounds of these type tend to be abstracted from any specific user action.

Character Depth

- affects the low pass and high pass cutoff filters of the filter SynthDef in Supercollider. Every sound generated passes through this filter, so the depth globally changes the feel of the game's audio. The depth 'filter' is a type of game sound process that involves "shaping the sound", where a generated sound "incorporates game parameters over which [the user] may have some control" (Collins 2013, p6).

Character Level

- 'Levelling up' gradually adds new pads to the soundtrack, resulting a richer and more harmonic chord. The original 3 pads will only play the root, fifth, and octave notes. Subsequently added chords will add notes stochastically generated from the probability tables. Levelling up also increases the feedback parameter of the delay SynthDef.

4.2.3 - Direct Elements

These sounds occupy the concept of Interactive audio, sounds "which can respond to the player directly" (Collins 2008, p185). These sounds are generally 'one-shot' triggers of which the user has direct control, and tend to be the main objective which user's seek and interact with.

Eating Plankton

- results in a number of tones being played in quick succession. The frequency of the sine tones are selected stochastically from the current chord probabilities. The range of potential frequencies depends on the depth at with the character is swimming at the time of the plankton being eaten - being nearer the sea floor results in lower pitched sine tones. The number of tones played depends on the type (shape) of plankton. After every eight plankton, another type of sound is triggered and accompanied with a glowing visual effect.

Interacting with a Urchin Creature

- proximity to the creature triggers a new pad which rises in volume depending on how close to the urchin the player gets. The pad is off-key one of the few sources of dissonance in the soundtrack of *CELL.* Colliding with the Urchin's feelers triggers a bell-like sound.

Interacting with a Starfish Creature

- this causes a change in chord. A message is passed to the trigger class to change the current chord probabilities to those of another chord.

Interacting with a Jellyfish Creature

- causes a wind-chime like sound, where by the jellyfish's tentacles are equivalent to pitched tones, arranged in pitch order from left-to-right (low to high).

Entering/Exiting the 'Egg'

- act as strong low pass filters over the entire soundtrack, affecting the low pass parameter of the filter SynthDef. The main pad's volume is significantly reduced, and a new pad is introduced. Entering and exiting the egg also causes a bubble-like sound to be triggered.

Jumping above the Surface

- Exiting the water releases the cutoff value from the filter. At the point of exit or entry, a 'drip' sound is performed.

4.2.4 - Continuous Beat-Locked Elements

A 16 beat loop begins upon the game starting up, and different elements can be added to the loop depending on interactions. These elements are the most soundtrack-like, as they involve repeated patterns which give the music a sense of timing and repetition. Continuous game-time results in a richer and busier loop.

Spark Creature

Sparks are elements in the 16-beat sequence, and are sonically represented as pitched tones. The temporal position of the tone depends on at which beat the sequence was at when the spark first spawned. 'Collecting' sparks results in a looping melody being played continuously. These notes will respond to chord changes, and will be passed through the global filter.

Friendly Creature

Friendlies play randomly generated beat-locked notes. The volume and pan of the notes relates to the distance and position of the friendly in comparison to the player. The notes the friendly plays will remain inside the current chord.

4.3 - Version Control and Workflow

In order to aid development, *CELL* utilises the version control system Git. This allows the entire project to be kept Open Source, in keeping with the policy of Cinder and SuperCollider. Utilising GitHub was a secure way of backing up during the development of the project. The project's source code can be found at: http://amputek.github.com/Cellv0/

4.4 - Compiling to Application

In order to help achieve the original Project Requirements of a usable game, I felt it was necessary to attempt compiling *CELL* into a single executable file. Compiling Cinder applications into an executable is relatively simple (although the compiled file would only operate on OS X). Similarly, the process of creating a SuperCollider standalone is well documented. However, archiving the Cinder project *with* the SuperCollider files into a single executable .app file proved more complicated. It was possible to run *CELL* as an application if the Cinder .app was opened a few seconds after the SuperCollider .app. To go a step further, these could be executed in that fashion by an encapsulating shell script. The *Platypus* software by Sveinbjorn Thordarson "creates native Mac OS X applications from interpreted scripts" (sveinbjorn.org/platypus, accessed 12/1/13). The shell script which executes the files is as follows:

```
#!/bin/sh
open Cell-Audio.app &
sleep 5
open Cell.app &
```

The result is a single .app which runs both the Cinder-made and SuperCollider-made standalone applications. The shell app itself quits immediately, leaving just the Cinder and SuperCollider apps running. Cinder's `app::shutdown()`method is overridden, so that prior to quitting, an OSC message can be sent to SuperCollider, which has it's own "`0.exit`" function for quitting internally. This makes sure that all applications are quitted at the correct time. The SuperCollider-made standalone (called "Cinder-Audio") will contain code that has not been written by me, and is based on the SuperCollider Standalone template available here: http://sourceforge.net/projects/supercollider/files/Mac%20OS%20Standalone%20Template/. For more details on running *CELL* from Standalone, turn to section 8.3.

The Cell icon in the Dock

5 - Evaluation

In order to determine whether or not the project has achieved its goals, it was necessary to perform a number of forms of evaluations. In a full game development process, there are several stages of Testing and Evaluation. Many of these would be in-house (such as Quality Assurance, and various forms of Usability Testings). Others are performed by members of the public (Beta Testing). As the game is intended for users, it was important to perform user-based testing and evaluations at various points throughout the design and implementation process in order to steer the project in the right direction - not only from a usability perspective, but from a user experience perspective. Understanding the user's experience helps improve the "immersion, flow and playability" (Bernhaupt 2010, p4) of the game.

While there are requirements that were set up at the beginning of the project planning, requirements for a heavily user-based project such as this tend to evolve depending on user feedback. User evaluations were performed during and after the project implementation. Testing "involves playing a game before its release to determine where or not it is *playable* - bug-free, consistent, and entertaining" (Novak 2011, p339).

Similarly, regular unit tests were run to access CPU usage of the program, to compare the usage balance between Cinder and Supercollider, and to locate particular processes that were stalling the CPU. This ensures that the program, which contains real-time visual and sonic updating, runs smoothly for users.

5.1 - User Evaluation

The first part of the Evaluation process is concerned with usability and user experience, based on the opinions of the public - potential users of the game.

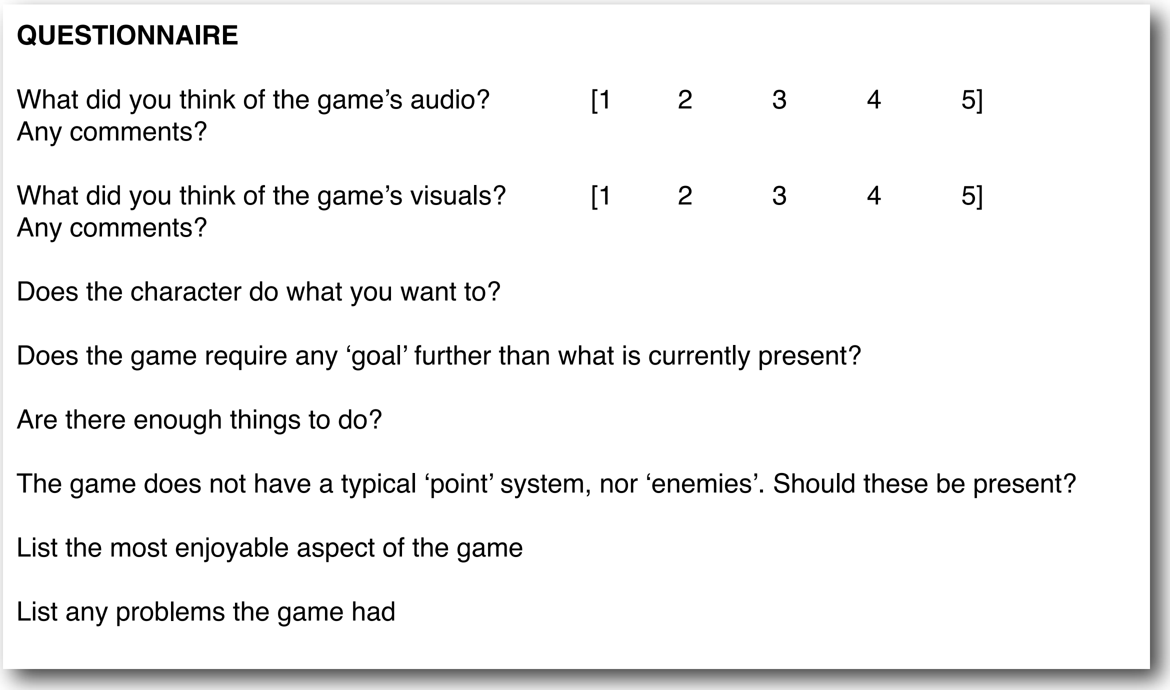
5.1.1 - Interim User Evaluation

At an early point in the design, it was necessary to steer the design of my project into a more focused route. This form of user evaluation occupies the “Production / Prototype” phase of the game development cycle - it “contains a sample of gameplay” with the core “shining gameplay mechanic/story hook” (Levy, Novak 2010, p 52). This Interim evaluation can be seen as part of an iterative evaluation process which will continue in various forms until the project is completed.

While a number of core gameplay mechanics were in place, there were a number of potential forms that the gameplay could take. I set up some informal user evaluation sessions to give users a chance to playtest *CELL* and voice their opinions. I gave each tester a short questionnaire which enabled me to:

- obtain some quantitative data
- find out any issues that may be apparent in the game
- address some concerns I am having in relation to the direction the game is taking. Currently, the chief gameplay is simply moving around and interacting with the different entities. There is no point system, and there are no enemies to chase and fight.

5.1.1.1 - Interim User Evaluation: Questionnaire



QUESTIONNAIRE

What did you think of the game’s audio? [1 2 3 4 5]
Any comments?

What did you think of the game’s visuals? [1 2 3 4 5]
Any comments?

Does the character do what you want to?

Does the game require any ‘goal’ further than what is currently present?

Are there enough things to do?

The game does not have a typical ‘point’ system, nor ‘enemies’. Should these be present?

List the most enjoyable aspect of the game

List any problems the game had

5.1.1.2 - Interim User Evaluation: Results

Audio: 5/5. Good, dream-like

Visuals: 5/5. Very smooth and flowing

Usable: Yes

Goals: No further goal required

Enough to do: Maybe some more organisms to interact with, but not many

Points / Enemies required: No, I don't think of it as a "game", more of a "moving interactive picture"

Most enjoyable: Look, movement, sound

Problems: None

Audio: 4/5. Some volume issues (some things are too quiet)

Visuals: 4/5. Movements look good, but the main character could be more exciting.

Usable: Yes - although I didn't realise you could move the cursor outside the window

Goals: There doesn't need to be any other goal than that which is currently present. The 'goal' of making music is enough as it is.

Enough to do: There should be more plankton, as there are certain points where there is very little on screen. The plankton clusters should be more common. There could be more creatures in general, especially 'fish' like things (like a manta ray)

Points / Enemies required: No

Most enjoyable: The starfish, and levelling up.

Problems: There is a bug with the 'twirl' creature, which sometimes doesn't make sound.

Audio: 4/5. Music suits the setting. Some minor volume issues

Visuals: 4/5. More detailed seafloor. Ocean surface would be interesting

Usable: Yes

Goals: No further goal needed

Enough to do: Yes

Most enjoyable: Going upwards towards 'surface' and seeing lightbeams etc

Problems: Main character could grow quicker, and have faster movement

Audio: 3/5.

Visuals: 5/5.

Usable: Yes

Goals: No further goal needed

Enough to do: Would be interesting to have more complex interactions, like pushing objects to specific locations, or to find fish that would follow you. Or some sort of beat-making area with pushable objects

Most enjoyable: The feelers

Problems: None

5.1.1.3 - Interim User Evaluation: Analysis and Conclusions

Reactions to the game at this stage were generally positive. There were few fundamental gameplay faults, and no visual/audio bugs. Most problems encountered were things that could be tweaked fairly easily. I have taken the thoughts of the user's on board and made the following changes:

- Created a floor and surface class to give the game boundaries along the Y axis. The game is still infinite along the X axis however.
- Increased the rate at which plankton spawn. 'Clusters' of plankton are more likely
- Increased the rate at which the character grows
- Repeatedly touching the 'eggs' causes them to hatch into small spark-type creatures which create sounds on the sequencer loop.

There may be an issue with how the Procedural Content Generator works. Currently, if a certain entity has not been seen for a number of frames (eg, 1500 frames), a new one will spawn in the direction the character is facing. While this means that the user constantly has something to interact with, it compromises the user's will to explore, as the entities are brought to user, rather than the user finding them. Although no users brought this up during interviews, I observed that a large proportion of the user base tended to stay at the depth they were first spawned up, and rarely attempted to reach the surface of their own accord. To cope with this, I have made sure that entities will only spawn at certain depths. Urchin creatures for example, will only spawn at the lowest depths, while other creatures will spawn nearer the surface. This results in a varied experience (visually and sonically) throughout the environment, and will hopefully inspire users to search the environment themselves.

I also observed that users tended to keep the mouse cursor close to the player character, resulting in slow movement. Moving the cursor outside of the Cinder window results in much faster movement, but this is not obvious (nor desirable). I will limit the cursor location to inside the window, but increase the average movement speed accordingly.

It was interesting to observe that people were hesitant about approaching the Jellyfish - something they associated with danger. However, the musical output of the jellyfish encourages interaction. This is a form of "adaptive audio" (Collins 2008, p4). Instead of introducing dissonance as a sign of danger, or fear, the jellyfish creates pleasant in-key sounds.

5.1.2 - Beta Testing

Prior to the completion of the project, I distributed a working application to a number of people to allow a short of Beta Testing. The beta phase is a milestone in game development, as “all assets are permanent and all high bugs have been addressed” (Levy, Novak 2010 p53). Beta Testing involves volunteer users “playing a game thoroughly from start to finish to determine whether the game is usable and playable” (Novak, 2012, p341). I asked the testers to play the game for a few days and note down any bugs they found while playing *CELL*. I also asked them them to note down any changes that they think would improve the game. The Beta stage complies to my original established requirements: that *CELL* is <u>usable</u> and <u>enjoyable.</u>

Problem	Result
Popping sound when exiting the Eggs	Fixed - using the ‘rate’ parameter of synth defs, and using triggers when using `filter.set()`. This means the cutoff value sweeps to a new value over a period of 1 second.
Spark creatures don’t always follow you, they get ‘stuck’ on the Jellyfish	This was actually a deliberate gameplay mechanic, and the Sparks would have eventually returned to the player. The issue was that the volume of the jellyfish did not diminish as the Player moved away. This has now been fixed.
CELL wont open on Mac OSX 10.6	Unsure of why this happens - will need an opportunity to sit down with a OS X 10.6 Mac to work out what the issue is. Not considered a major bug.
The delay effect is too strong, it shouldn’t be on every sound.	Fixed, the delay sound starts off very quiet and slowly grows in volume (through the feedback mix parameter on the Delay SynthDef)
It is not necessarily clear that the mouse controls the character. A hint at the beginning would be useful.	I have implemented a written instruction that pops up after the main splash screen. It says “Cell will follow your cursor. Explore”
All audio is too quiet.	Slightly increased the volume of all samples in SuperCollider. Increased the volume of the FX SynthDefs.

5.2 - Unit Tests

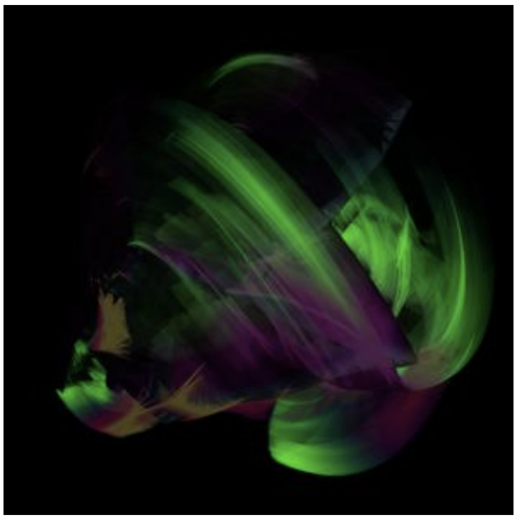
5.2.1 - Processing vs. Cinder

Prior to starting the project, I made a number of experiments in the Processing environment. Processing is a graphical framework built upon Java. I experimented with visuals, user input, building Graphical User Interfaces. The original plan was to use Processing as the main part of the project (with Supercollider providing the music). It was soon evident, however, that Processing (and Java in general) were extremely CPU intensive. This is because Java builds a Virtual Machine on top of the operating system every time the program runs. While this makes creating programs easier (the compile process is abstracted from the programmer), it means that users are required to be up-to-date with Java versions, as well as significantly increasing CPU usages. Running everything through the Java Virtual Machine was costly, particularly when it came to drawing and rendering. Cinder offered an alternative to Processing while following many of the same rules. Cinder is "a community-developed, free and open source library for creative coding in C++". (libcinder.org, accessed 10/11/12)

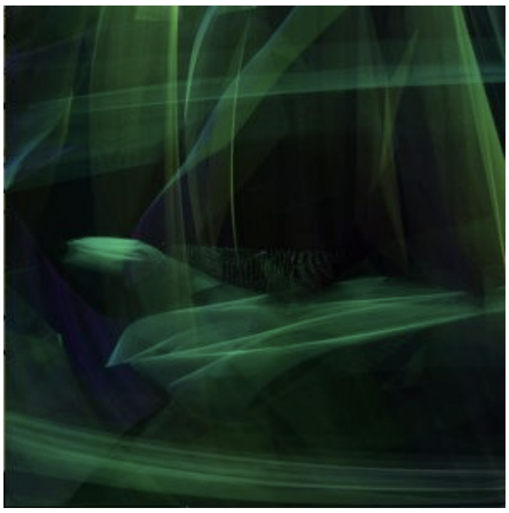
	Pros	Cons
Processing (Java)	- simple, easy to use - apps will on any machine that has Java installed - automatic garbage collection	- limited IDE - no debugging
Cinder (C++)	- C++ is faster - XCode is a more advanced IDE - allows debugging - XCode allows easy Profiling	- XCode is more complex - Will only compile for OSX - No garbage collection - C++ is a lower level language

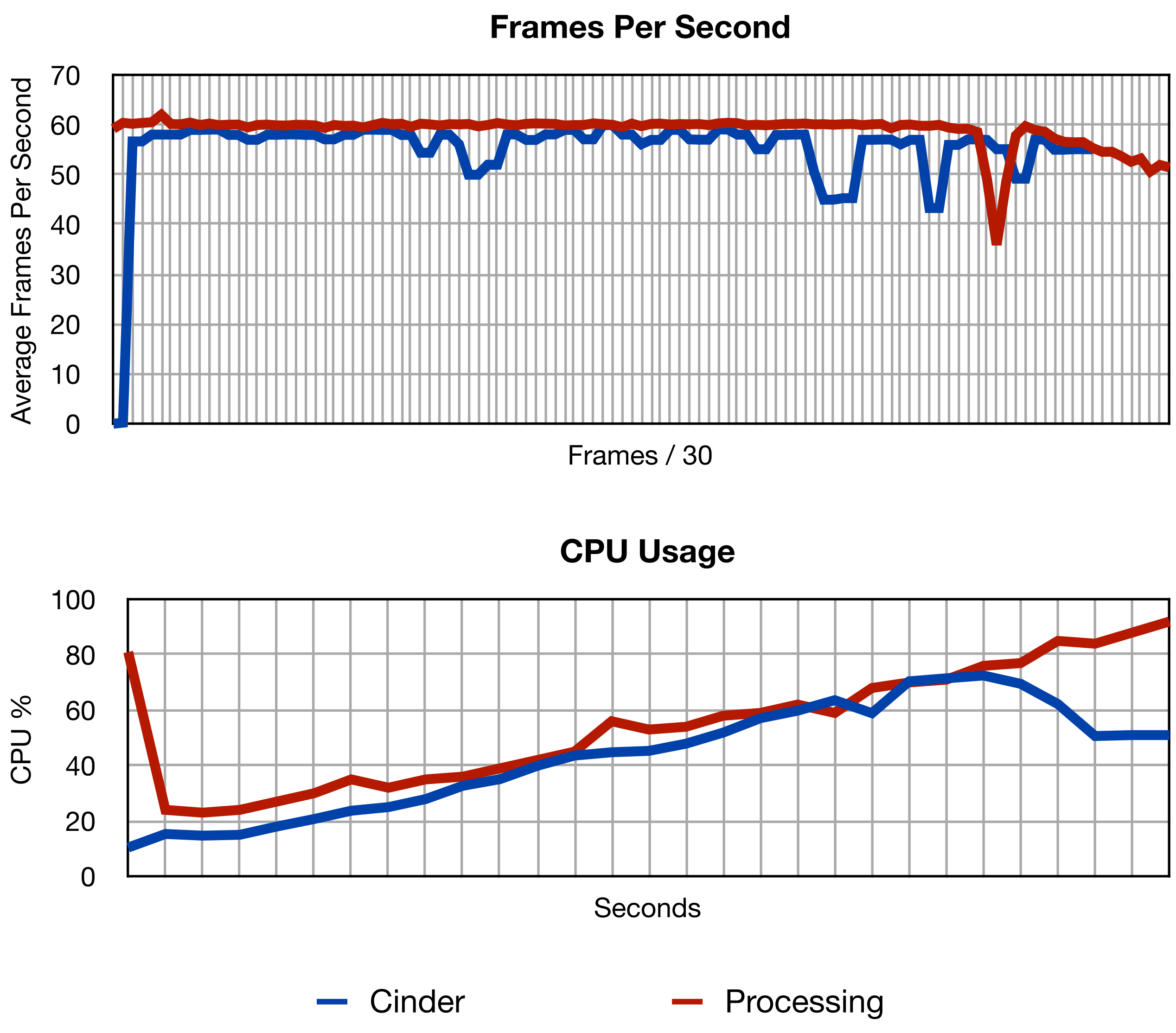
In order to help decide between the two frameworks, a sample program was written in both and CPU measured. The test program is a visual generative piece, involving propagating lines that move within a sphere - nearby vertexes are joined by new lines to form complex webs. This is useful as the program contains large collections of data as well as continuous drawing. A drop-off in frame rate is to be expected because of this. The program has been written in both Cinder and Processing, with as many similarities kept as possible.

Cinder WEBS output



Processing WEBS output





The graphs show that Processing and Cinder both perform in a similar fashion, with Cinder performing slightly better in most cases. Cinder has other benefits in that - through XCode - it produces a stand alone application that could potentially be submitted to the Mac App store. Java applets have their own issues with compatibility.

5.2.2 - Unit Test #1: CPU% in Cinder

These unit tests (or ‘profiling’) involves keeping track of CPU and memory usage while the program is running. These were performed regularly to ensure that *CELL* was as optimised as possible - using the minimum of CPU and running and a high and regular framerate. Regular framerate was one of my key established requirements.

Through the use of Apple’s *Instruments* application, I was able to perform unit tests throughout the development of the project. A chief concern of mine was that of large frame-rate drops at certain points in the game, and while a high frame rate was not necessarily important, Cinder tended to have inconsistent frame rates, with sudden drops happening at certain points. From a user experience perspective this needed improving. I ran a unit test which observed the percentage of CPU usage of each particular method in the code. The program’s code is compartmentalised into update functions, and draw functions, meaning it would be easy to localise the points at which the CPU was struggling. I assume that frame-rate drops are generally due to Cinder struggling to draw a large number of shapes to the screen.

Unit Test #1 (5/2/2013): Event Profiler

Process	CPU%	Sub Process	CPU%	Sub Process	CPU%
draw()	68.2%	Starfish::draw()	16.0%	drawSolidCircle()	15.9%
				other processes	*neg*
		SeaSurface::draw()	14.4%	draw(Path2d)	14.2%
				other processes	*neg*
		Plankton::draw()	7.9%	drawStrokedCircle()	6.5%
				drawSolidCircle()	1.2%
				other processes	*neg*
		Spore::draw()	7.3%		
update()	15.0%	updateEnvironment()	10.3%	SeaSurface::update()	10.2%
				others	*neg*
		updateEntities()	4.1%		
		entityGenerator()	0.0%		
non-Cinder processes					

As I assumed, the `draw()` loop inside Cinder is hogging a disproportionate amount of the CPU, which is causing the frame rate to drop. I have localised the problem to that of the ‘Starfish’ entity, which features a large number of drawn circles. In response to this I have changed the number of circles drawn by that entity, resulting in a minor change in visual output but a drop in CPU usage by that entity. The surface is also hogging the CPU, so I have changed the code so it will only visually update and draw when it is in view (the 'GameObject' part of the surface will always update, so the program can keep track of it's relative location, but it does not need to go through the process of applying perlin noise and sine waves to the shapes of the Paths). While CPU use would still be high when it *is* in frame, it wont happen all the time. Many aspects of the game involve this sort of trade-off between CPU usage and aesthetics.

Based on this unit test, I feel I need to restructure my entity manager. The procedural content generator creates new entities, but the entity manager never removes them. In the update loops, entities are deleted if they are far enough away from the user. The PCG will spawn new entities in front of the user, so hopefully users will not notice the difference but CPU will be saved.

5.2.3 - Unit Test #2: CPU% in Cinder

Unit Test #2 (25/2/2013): Event Profiler					
Process	**CPU%**	**Sub Process**	**CPU%**	**Sub Process**	**CPU%**
draw()	68.2%	Feeler::draw()	26.6%	drawSolidCircle()	22.9%
				other processes	*neg*
		Plankton::draw()	7.2%	drawStrokedCircle()	5.9%
				drawSolidCircle()	1.2%
				other processes	*neg*
		SeaSurface::draw()	5.7%	draw(Path2d)	5.5%
				other processes	*neg*
update()	19.7%	updateEnvironment()		SeaSurface::update()	10.2%
				others	*neg*
		updateEntities()			
		entityGenerator()			
non-Cinder processes					

An improvement, and it is it be expected that Feelers hog CPU, as a number of different entities use them.

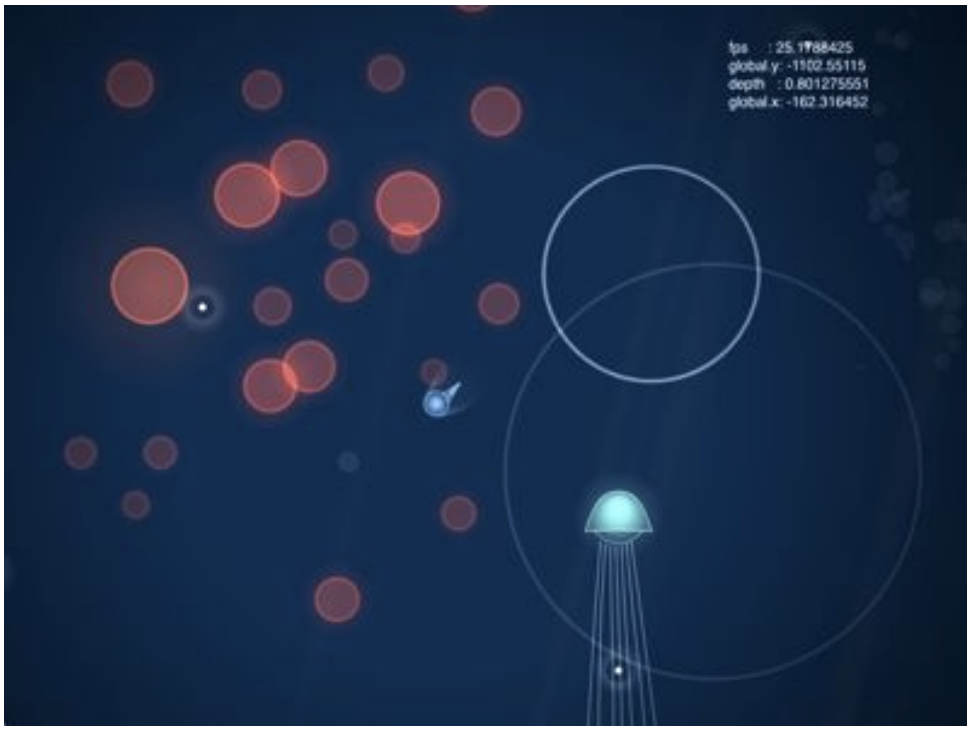
Unit Test #3: Draw methods

To further optimise CPU usage of the program, it was necessary to examine draw methods, where alot of the CPU was being used. In a number of cases, large numbers of circles (`gl::drawSolidCircle()`) were being drawn, a comparatively expensive process for Cinder to be continuously carrying out. A single plankton entity, for example, involved 14 separate circles being drawn. As all plankton look the same (apart from size), CPU usage could be improved by using a png image as a texture instead. Unit Tests were carried out with both options to see which would provide less CPU usage

Unit Test #3 (28/2/13): Plankton Draw Comparisons			
	Samples	**overall CPU %**	**CPU% of draw()**
draw Texture	56	3.6%	8.5%
draw Circles	390	18.8%	37.3%

Based on these results it was clear that, where possible, it would be better to draw textures rather than draw circles. The draw methods for Beams, Bubbles, Plankton, Urchins, Jellyfish, Eggs, Spores were updated to this new method. As well as causing an improvement in CPU usage, it resulted in a more aesthetic look - something which I consider an important part of my initial requirements.

CELL prior to the change to png textures



```
fps      : 25.1?88425
global.y: -1102.55115
depth    : 0.801275551
global.x: -162.316452
```

6 - Conclusion

The purpose of this project was to develop an interactive music-based video game that users would find enjoyable. My original requirements stated that *CELL* should be usable, have a pleasant user experience, and have an abstraction between gameplay and music. I believe that these goals have been accomplished.

Usability

Usability testing fell into two categories: usability within the game itself (or 'playability'), and usability of running the game in the first place. User testing showed that there very few usability issues in-game, as there was literally only one method of input (the trackpad/mouse). Any issues of usability were discovered through repeated user testing, and immediately solved. For example, the suggestion that using the mouse might not be obvious was fixed by drawing a written instruction upon the game's start up (see section 5.1.2).

However, there were some issues in getting the game to run. This was generally due to the complexity of compiling an XCode C++ project (which included Cinder's 'CinderBlock' OSC library) with SuperCollider. This was a relatively un-documented process (outside of pure game development) in the way that these programs would compile together, and most game development literature is concerned with the creation of games using a game engine which already has it's own sound creation modules. Running *CELL* out of XCode and SuperCollider environments offered few issues, but automating the processes into a single standalone proved to be more complex. I do, however, feel that these issues are not central to the project itself, and are more to do with distribution. The game itself, in terms of usability and playability, works fine.

The usability of *CELL* is largely based on user feedback, and users rated the game very highly in terms of usability. I put forth the usability-based question: "Does the character do what you want to?" (section 5.1.1.1) and the user response was positive in all cases.

User Experience

In comparison to Usability, User Experience (UX) was a much more subjective concern. My original requirements stated that *CELL* should be enjoyable, rewarding, and non-challenging, but what is considered enjoyable, or rewarding, depends very much on the User. The various stages of user evaluation proved that *CELL* was indeed enjoyable. Most users rated both the audio and visuals very highly, and most felt that there was a sufficient amount of things to do. There were some minor criticisms, mostly based around the absence of larger-scale objectives. I feel however that these would be an eventual addition through future development of the project, and were not a concern of my original requirements. Users responded with varying opinions on their favourite aspect of the game, but all responses were positive. From my own perspective, I feel that *CELL* has been very successful, and when I would observe people playing it, I was pleasantly surprised at the length of time that people would spend in-game, with the music and visuals combining to give an entrancing and calming experience.

Abstraction

This was one area that could have been more successful. Most of the visual-audio relationships are direct, and deliberately so. Having audio 'come from nowhere' would be confusing in terms of gameplay, regardless of the audio output. My original plan was to have more global changes in music in response to broader changes of gameplay, for example if the player were to be particularly careful at catching plankton, or if the player explored a considerable amount. While there are few of these implemented (the pad growing in richness, and the delay feedback increasing over time), it would have been good to implement more. These sort of things are, however, more difficult to implement, as quantifying these complex behaviours is not obvious. Nevertheless, *CELL* is not a Digital-Audio Workstation, where sounds are added ‘deliberately’ by a composer to form a song. Instead, the layers of sound build up gradually *because* of gameplay. There is without a doubt a level of abstraction between the player and the music that results.

I feel that *CELL* has been successful and has achieved the initial requirements that I set out. It is, by all accounts, an enjoyable experience, regardless of whether or not it is considered a ‘game’ or just an ‘interactive experience’.

6.2 - Future Developments

The *Cell* project is at one stage of completion, but as with all game development cycles, there is plenty of room for added features and development. As this project involved the creation both of a game engine, and of a game within that engine, it would easily be possible to add more entities to the environment, as the framework for generating, drawing and updating entities is already in place. Classes are modular and well-encapsulated so that entities can use certain classes easily - more Feeler-based entities, or other entities which use Tail objects. Further developments could include more complex gameplays, perhaps eventually with a specific progression and an end-game. More abstraction between visual and audio would be a good thing to implement - with gameplay behaviours being recorded, quantified, and translated into musical outputs. Currently, the music of *CELL* is very ambient, and it would be interesting to allow the music to take different directions depending on gameplay. The *CELL* presented here is a single step in an ongoing development process.

7 - References

ADAMS, Ernest. *Fundamentals of Game Design.* Pearson Education, CA, USA (2010)

BERNHAUPT, Regina. *Evaluating User Experience in Games: Concepts and Methods.* Springer, London, UK (2010)

COLLINS, Karen (editor). *From Pac-Man to Pop Music: Interactive Audio in Games and New Media.* Ashgate Publishing Limited, Hampshire, UK (2008)

COLLINS, Karen. *Game Sound: An Introduction to the History, Theory, and Practice of Video Game Music and Sound Design.* The MIT Press, Cambridge, MA, USA (2008)

COLLINS, Karen. *Playing With Sound: A Theory of Interacting With Sound and Music in Video Games.* MIT Press, Cambridge, MA, USA (2013)

COPE, David. *Computer Models of Musical Creativity.* MIT Press, Cambridge, MA, USA (2005)

FARNELL, Andy. *An Introduction to Procedural Audio and it's Application in Computer Games.* (2007)

GOLD, Julian. *Object-Oriented Game Development.* Pearson Education Limited, Harlow, UK (2004)

GREENBERG, Ira. *Processing: Creative Coding and Computational Art.* Apress, Berkely CA, USA (2007)

KAMP, Michiel. *Ludic Music in Video Games.* Masters Thesis, Utrecht University, NE (2009)

KELLY, Charles. *Programming 2D Games.* CRC Press, USA (2012)

KOMOSINSKI, Macief and ADAMATZKY, Andrew. *Artificial Life Models in Software.* Springer, London, UK. (2009)

MATOSSIAN, Nouritza. *Xenakis.* Taplinger, New York, USA (1986)

NOVAK, Jeannie. *Game Development Essentials: An Introduction.* Delmar, Cengage Learning. New York, USA (2012)

NOVAK, Jeannie and LEVY, Luis. *Game Development Essentials: Game QA & Testing.* Delmar, Cengage Learning. New York, USA (2010)

PFEIFER Rolf and BONGARD Josh. *How the Body Shapes the Way We Think: A New View of Intelligence.* MIT Press, Cambridge, MA, USA (2007)

NIERHAUS, Gerhard. *Algorithmic Composition: Paradigms of Automated Music Generation*, Springer-Verlag/ Wein, Germany (2009)

REAS, Casey and FRY, Ben. *Processing: A Programming Handbook for Visual Designers and Artists.* MIT Press, Cambridge, MA, USA (2007)

REAS, Casey and McWILLIAMS, Chandler. *Form + Code in Design, Art and Architecture.* Princeton Architectural Press, New York, USA (2010)

STEVENS, Richard. *The Game Audio Tutorial.* Focal Press, Buildington, MA, USA (2011)

TOGELIUS, Julian, Kastbjerg E, Schefl D, Yannakakis, G.N. *What is Procedural Content Generation? Mario on the borderline* ,IT University of Copenhagen, Denmark (2011)

WINTORY, Austin 2012 interview with *thesixaxis,* available at http://www.thesixthaxis.com/2012/03/15/interview-journey-composer-austin-wintory/ (accessed 10/4/13)

Video Games

***Title*, Developer, Publisher (year)**

Space Invaders. Taito Corporation, Taito/Miday (1978)
Elite. David Braben and Ian Bell, Acornsoft (1984)
Music 2000. Codemasters (1999)
Rez. Q Entertainment, Sega/Sony/Microsoft (2001)
Singstar, London Studio, Sony Computer Entertainment Europe (2004)
electroplankton, indieszero, Nintendo (2005)
fl0w. thatgamecompany, Sony Computer Entertainment (2006)
Flower. thatgamecompany, Sony Computer Entertainment (2009)
Child Of Eden. Q Entertainment, Ubisoft (2009)
Red Dead Redemption. Rockstar San Diego, Rockstar Games (2010)
Journey. thatgamecompany, Sony Computer Entertainment (2012)
Guitar Hero. Harmonix Music Systems, Red Octane (2005)

8 - Appendices

Appendix: 8.1 - Project Log

Pre-term:

I have been exploring Processing in detail, and have loaned a number of books which detail the creation of generative art using Processing.

Early experiments in processing, with parallax scrolling, world navigation, and item collecting. The player-character here 'evolves' as it collects items. Some 'follower' nodes will cause some simple output in SuperCollider.

Autumn Week 2:

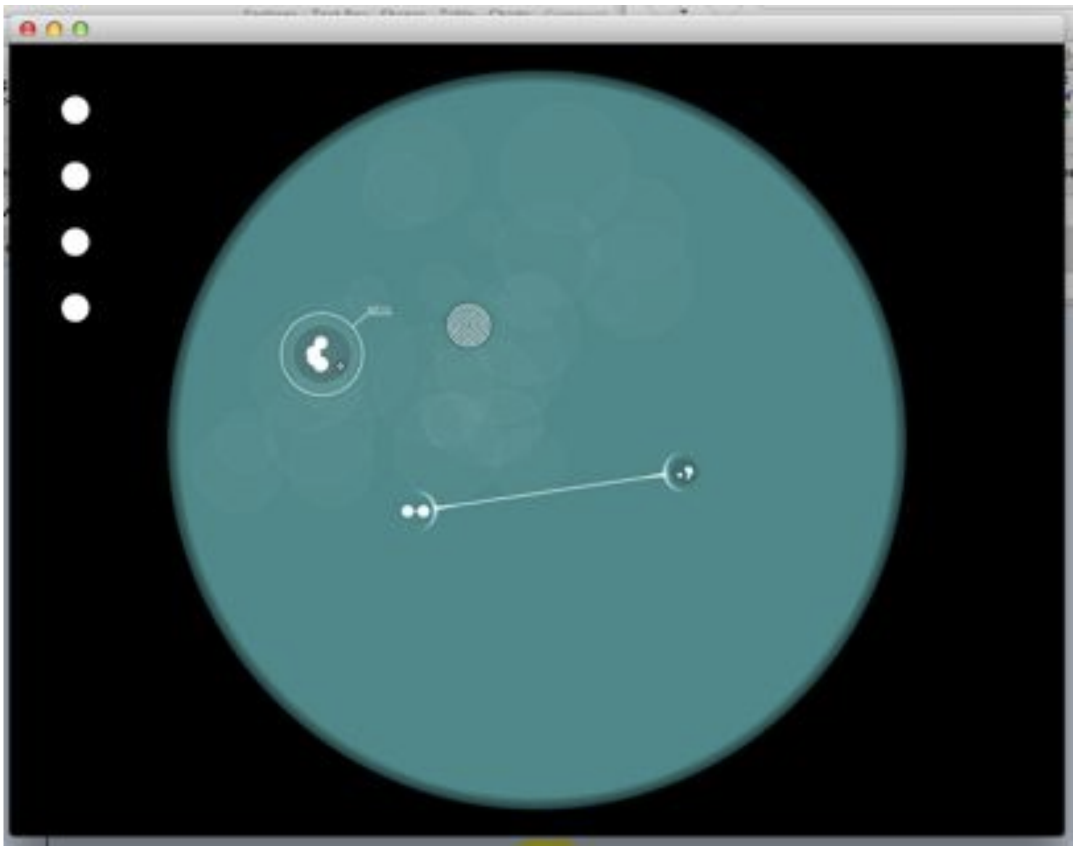
I have had the first meeting with my supervisor and have focused some of my ideas. I have begun to explore Cinder as an alternative to Processing. It is essentially a C++ version of Processing (which is an extension of Java). This will hopefully reduce some overhead and allow some more specific control. I will explore this over the next two weeks as well as start designing some various 'entities' that the user will interact with. The best way to approach this will be to associate a specific interface with a specific musical outcome, which will contribute to a final piece of music. This form will lend itself to processing/ofx/cinder as it allows me to test each 'mini-game' individually.

Autumn Week 3:

I have refined my idea and avoided approaching the 'Adventure' aspect at this point - this would result in a lot of time and effort being put into drawing and rendering levels and environments. I have instead focused on a clearer immobile environment that a user has more explicit control over, at least for now. The abstraction still exists - but in the form of more complex algorithmic composition, rather than complex gameplay types.

This will enable me to focus on the interaction between supercollider and Cinder before approaching more high level gameplay aspects. I have a number of Processing sketches which deal with parallax scrolling, character movement and procedural content generation, and will port these over to C++ as and when I need to. As long as I keep the program modular in this way, it should be easy to combine the various functionalities. I have strived to keep the program as Object-Oriented as possible and keep code as concise. It is tempting to go over-board with complex drawing styles at this stage but it it's important to keep in mind how different game objects would share functionality.

Week 3 and 4: Cinder project with different nodes. Also implemented is the ability to link nodes. the 'highlighting' method of all GameObjects is displayed here - with customisable tooltips above and to the right of game object.



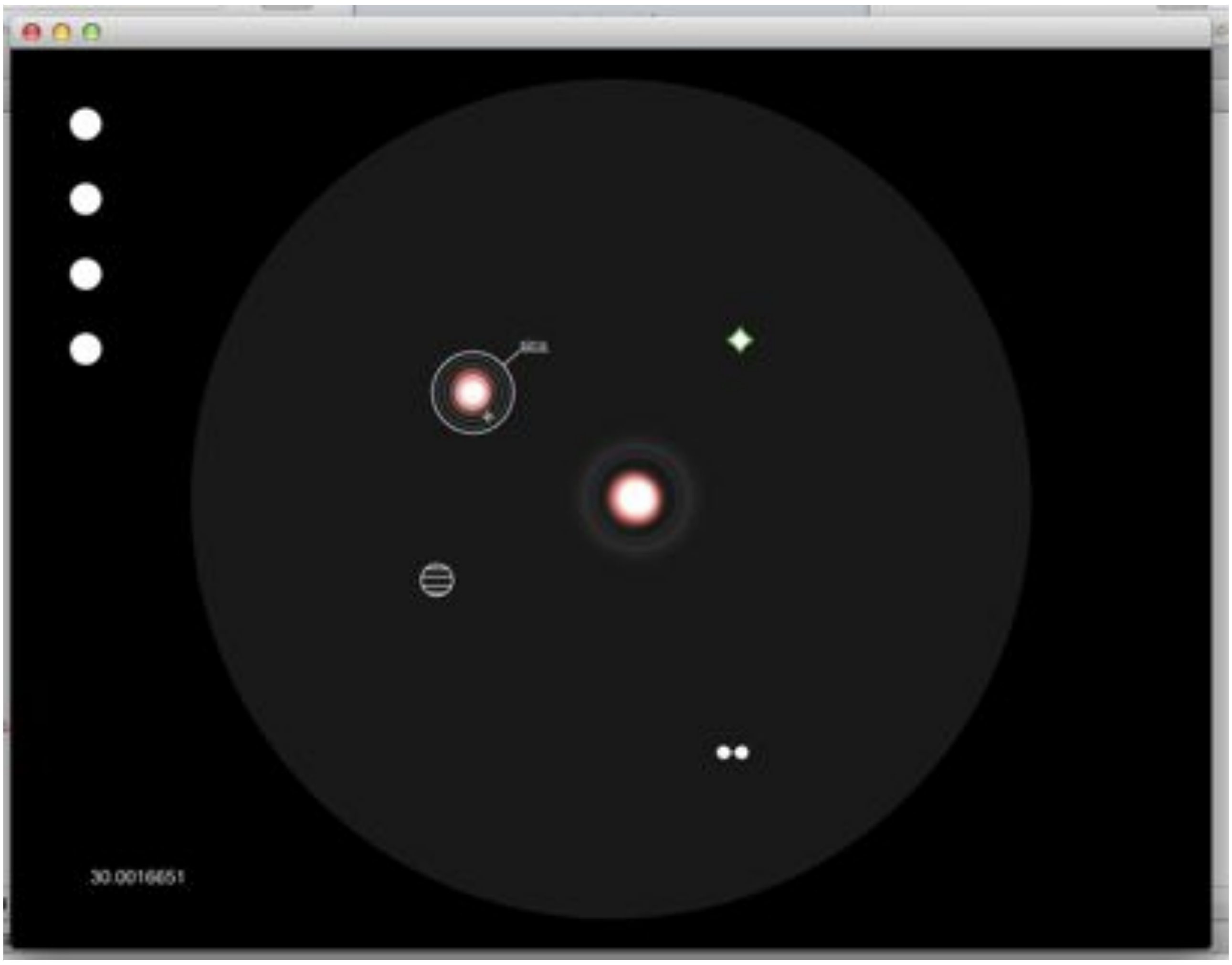
Autumn Week 4 and 5:

I have begun working on the Supercollider aspect of the project and successfully managed to get OSC messages passed between Cinder and Supercollider. Nodes created in Cinder have a mirrored 'Node' class inside Supercollider, which stores a relevant SynthDef and triggering information.

Autumn Week 6:

I have made some new tests regarding drawing in Cinder. Initial results seem to show that Cinder copes far better with repeated drawing of circles. For example for a typical 'particle' effect, where translucent ellipses are drawn upon each other in varying sizes to create a blurred effect, Cinder copes considerably better - with little or no frame rate drop compared to simply drawing opaque circles. This means I can experiment with more complex drawing techniques and styles. Cinder has an additive blending function which causes interesting blurred drawing styles.

Moving to Cinder. experiments with Cinder's additive blending function. This experiment is based around "Nodes" which can be fully and explicitly manipulated by the user. Alot of this will carry over to the final project - each node has a different musical connotation. The glowing red nodes are "Sine bubbles" which produce random sine blips which are kept in time and key by Supercollider. The pan and density of the sine bubbles depend on the node's location in the environment. The other nodes are "FX Nodes" which change how the "Sine bubbles" node play back. Attaching a delay Node to the Sine bubble node will cause every note that the Sine Bubble node plays to be put through a delayline in supercollider.



Autumn Week 7:
I have started the project proper, converting my concepts above into an exploratory adventure game and worked on creating a class system for an entity which follows the mouse around using Braitenberg system of movement. Trying out different draw styles in OpenGL. I have planned the structure of the project to make heavy use of inheritance - all GameObjects inherit from the same class.

Autumn Week 8:

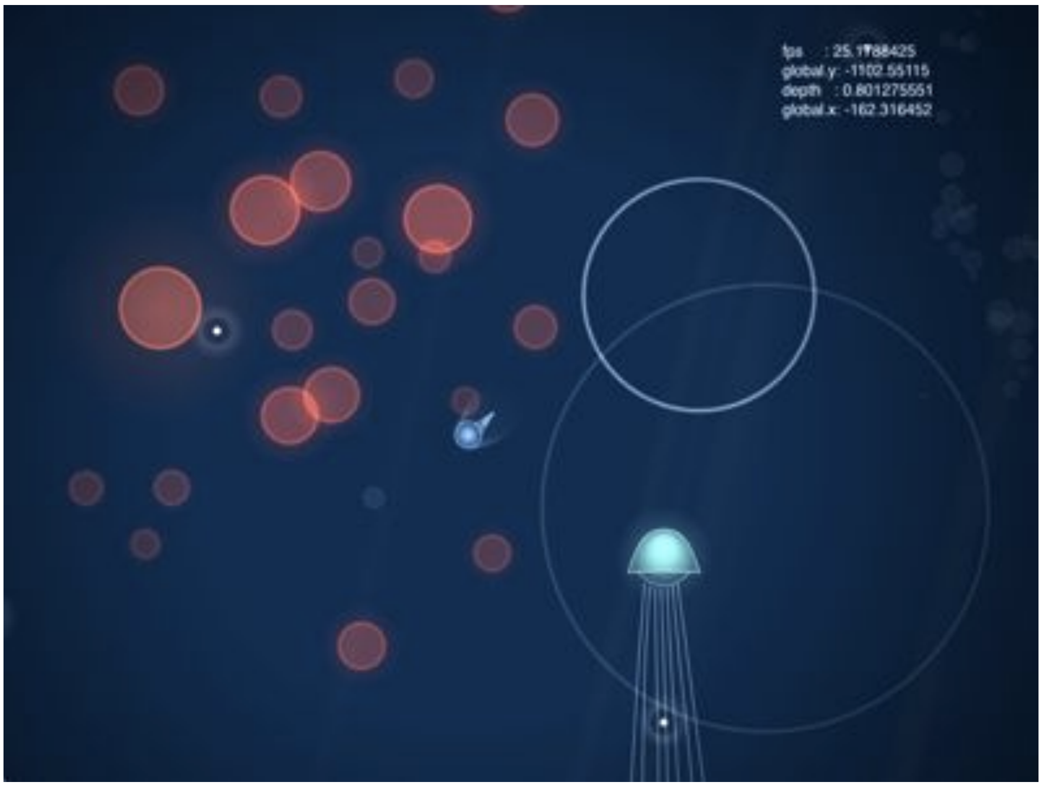
Added a basic tail to the player, with an encapsulating class system. Splashes and Tails inherit attributes from the same class - a “Finite” - a gameobject that has limited existence. I have worked on creating a proper scrolling system so that player can move around the game world. This is done by using a system of local and global co-ordinates with a depth parameter. The depth parameter causes parallax scrolling.

Autumn Week 9:

I have worked on a limited AI system - entities can ‘follow’ a particular position but avoid each other, so that their radius’s don’t intersect. I am basing the system on an AI system I built for a multi-directional shooter I have previously created. It gives a pleasant visual affect when entities are moving around together as they don’t all occupy the same space.

Autumn Week 10 and 11:

Adding some different entities, including plankton and jellyfish. Jellyfish use a complex system of Vectors and Path2D objects, along with solid circles drawn to make them appear to glow. I am pleased with how they look. I have created ‘feeler’ objects which respond to collision and move around in the environment. They are fun to play with and I think will be a big part of the game’s mechanics.



```
fps      : 25.1788425
global.y: -1102.55115
depth    : 0.801275551
global.x: -162.316452
```

Winter Break:

Over the winter break I have conducted my first user evaluation sessions, with family and friends. This featured a short questionnaire and an informal interview to guide the project. At this point I am having concerns about the gameplay and whether or not it is enjoyable and interesting enough. However the results have been positive and I don’t feel I need to radically change too much. I have also conducted some Unit

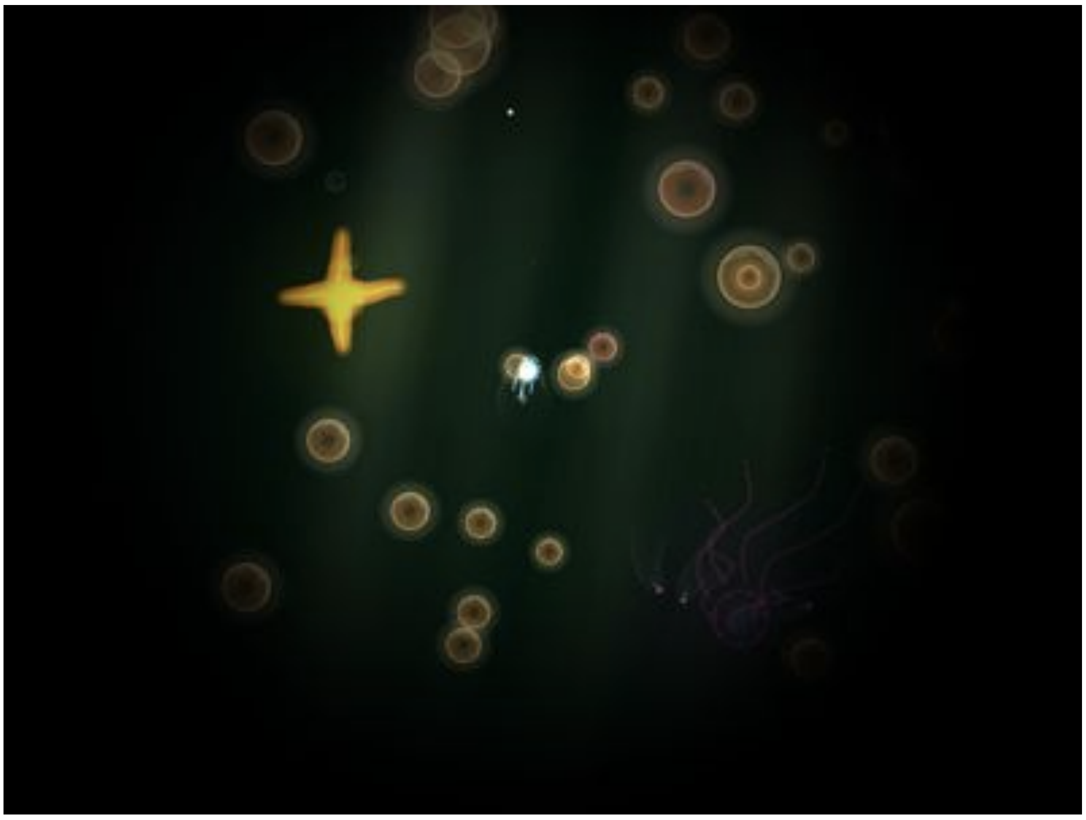
Tests using Apple's 'Instruments' software to see where CPU is being used the most. Optimising the program continues to be a problem, and although I think the code is clean and, for the most part, modular, frame rate still has a tendency to radically drop at certain points which ruins the 'flow' of the game

Spring Week 1:
I have successfully added a 'mask' to the game, a black rectangle which radially graduates towards full transparency in the center. This was done using an FBO in Cinder (a Frame Buffer Object), which can then be moved around and scaled accordingly. This is an extremely efficient way of doing this and uses very little CPU but has a pleasing visual effect.

Spring Week 2:
Using Adobe Illustrator, I have drawn some images that could be used instead of the OpenGL calls in Cinder. This has dramatically decreased CPU usage for certain entities. The plankton were drawn as a series of circles (upwards of 10 circles). Now, a gl::Texture is drawn instead. This means I can make the entity look as complex as I'd like without worrying about CPU usage too much. Because of this, I have increased the number of types of plankton and update SuperCollider part of the project to produce different sounds depending on the type eaten.

Spring Week 3 and 4:
I have continued to use Adobe Illustrator to make more texture for entities. Now eggs, jellyfish and spores use textures instead of vector objects.

figure: employing the use of .png as gl::Texture fields instead of vector objects inside Cinder.



Spring Week 5 and 6:

I have restructured and polished the SuperCollider part of the project. The SynthDefs are encapsulated inside a class, and sample Triggering is kept inside another class. The main SC file now just receives and sends OSC messages, and maintains the algorithmic loop.

Spring Week 7:

I have located a bizarre issue which involves Cinder's framerate dropping considerably while the Supercollider server is runner. I eventually managed to track it down to my Audio Interface (which I require for any sound). For some reason, sending the audio out through USB causes a frame rate drop (unrelated to any particular spike in CPU use). Using Core Audio does not affect frame rate.

Spring Week 8:

I have successfully packaged the application into a single executable app. This was done using the 'Platypus' software which makes executable shell scripts. The script runs each of the applications in turn. I ran into some issues in that it that it would result in three applications being open: the executable script, the Cinder standalone, and the SuperCollider standalone. These would remain open after quitting the Cinder app. I've managed to fix it, however, as SuperCollider has a "0.exit;" function which quits the application from the inside.

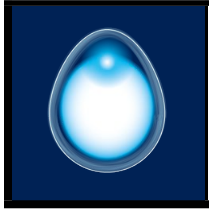
Spring Week 9:

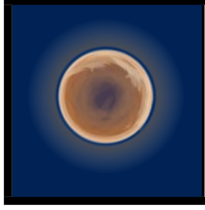
I have sent the project around for Beta Testing and worked on commenting and cleaning up. More documentation. I created a video for YouTube to allow others to see the project.

Spring Week 10:

Very little coding work - just some minor commenting and cleaning. Added some extra jellyfish types (which is easy to do due to the class system). Jellyfish now give out a 'distance' parameter to SuperCollider. More documentation work.

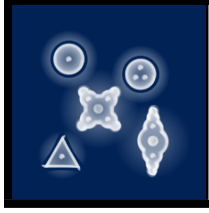
Appendix: 8.2 - Entity Overview

PLAYER		
Appearance	Behaviour	Audio
	User controlled, eats plankton and interacts with other entities. Evolves and grows over time	None

SPORE		
Appearance	Behaviour	Audio
	Collides with the player, causing the spore to be knocked backwards. Enough hits from the player results in the spore bursting, creating a 'Spark' entity	Colliding with the spore triggers a number of synth notes, the density of which depends on the number of times the spore has been hit. Bursting a spore causes a high-pitched noise

JELLYFISH		
Appearance	Behaviour	Audio
	Floats around at medium depths. It's tentacles collide with the player	Act like horizontal chimes, with the pitches from left (low) to right (high)

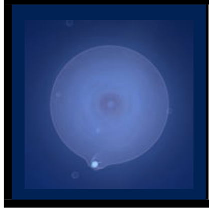
URCHIN		
Appearance	Behaviour	Audio
	Chases plankton and avoids other sea creatures. Found at deeper parts of the environment.	Dissonant pad at proximity. Colliding with feelers causes a bell-like sound

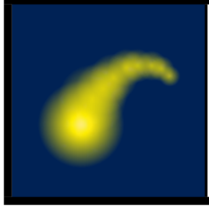
PLANKTON		
Appearance	Behaviour	Audio
	None, often found in clusters. Get eaten by other sea creatures	Short samples, with timbre relating to plankton shape

FRIENDLY		
Appearance	Behaviour	Audio
	Begins life inside an egg, once 'hatched' by the player, hunts for a specific type of Plankton.	Repeated notes depending on type/ colour

STARFISH		
Appearance	Behaviour	Audio
	Legs move away from player. Changes colour when player is nearby.	Causes a change in chord

GRASS		
Appearance	Behaviour	Audio
	Collides with the player	Collisions causes envelope synth noise depending on length

EGG		
Appearance	Behaviour	Audio
	Player can enter and exit the egg, causing the egg to 'wobble'. Slows the player down when inside	Low pass filter over the entire soundtrack. New pad is introduced and a bubble sound plays upon exit/entry

SPARK		
Appearance	Behaviour	Audio
	Follows the player. Interacts with other entities it is near. Glows when it is triggered	Equivalent to the repeating notes in the 16-beat loop in SuperCollider. Depending on the type of spore it came from, may be a glockenspiel, a piano, or an electronic piano note.

Appendix: 8.3 - Running Cell

A video of CELL in action can be found at: http://www.youtube.com/watch?v=RUm413HGNHA

It is possible to run CELL from a Standalone application. Please note that due to the way that the Standalone has been created, there will be code inside the Standalone that has not been written by me. The SuperCollider standalone "Cell-Audio" is based on a "Standalone Template" which is essentially a modified clone of the entire SuperCollider application - containing numerous classes. Details here: http://danielnouri.org/docs/SuperColliderHelp/Extending%20and%20Customizing%20SC/Creating-Standalone-Applications.html. The code that has been written by are the files that are in the "SuperCollider files" folder

- Download the Standalone, available at: `http://www.sendspace.com/file/ljmoyc` and unzip

Option 1:
- Run the 'Cell.app' standalone - this will only run on Mac OSX 10.8. It will open two applications, the Cinder-created standalone, and the SuperCollider standalone. They will both close when the Cinder app is quitted.

Option 2:
- Right-click on Cell.app, and click "Show Package Contents"
- Run the 'Cell.app/Contents/Resources/Cell-Audio' standalone
- Wait a few seconds
- Run the 'Cell.app/Contents/Resources/Cell-Visual' standalone

Option 3:
- Run the 'Cell.app/Contents/Resources/Cell-Audio' standalone
- Open the Cell Xcode project
- Compile and Run the Program from Xcode

Option 4:
- Copy the following classes into the SuperCollider classes folder (for Mac, the directory will be /Users/<username>/Library/Application Support/SuperCollider/Extensions/): SetUpCell.sc, Triggers.sc, Friendly.sc, Spark.sc. These classes are found in 'SuperCollider files/Classes'
- Recompile the SuperCollider class library
- Open CellSC.scd in SuperCollider 3.6+
- Run the code in that project (select the code and press cmd-Enter)
- Open the Cellv0 Xcode project, compile and run the program

Note that to run Cell in this way, you will need to move the directory of sounds (Cell.app/Contents/Resources/sounds) to another folder, and change the SetUpCell.sc class to reference this folder instead
Also required is the OSC CinderBlock