Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2014

Project Title:	**Ecosystems Visualisation and Game for Scientific Outreach**
Student:	**Jonathan Zheng**
CID:	**00730712**
Course:	**EIE4**
Project Supervisor:	**Dr Dan Goodman**
Second Marker:	**Dr Deniz Gunduz**

Abstract

This is EcoBuilder - a game in which you build an ecosystem. Our objective was to make use of game mechanics to help visualize the dynamics of ecosystems, which are networks of interactions between living organisms. To ensure that our model was accurate, it was a collaboration with the Life Sciences department.

We presented the project at the Imperial Festival, and the statistics and feedback gathered there are what we used for evaluation; it was generally well received and the public displayed clear interest in the idea, but it's also evident that there's a lot of room within the potential scope for future work.

Acknowledgements

I'd like to first thank those who were most closely involved. My supervisor Dan Goodman, my partner in Life Sciences Orestes Gutierrez Al-Khudhairy, and his supervisor Samraat Pawar.

I'd also like to thank everyone who helped out at our stall during the festival, including: Dimitrios Kontopoulos, Calum Pennington, Catalina Estrada, Tom Smith, Chris Wilson, Guy Woodward, Sarah Mayor.

Contents

1 Introduction

This is EcoBuilder - a game in which you build an ecosystem. The player is given the ability to introduce species into their world, and the goal is to add them in the right order to support as much life as possible.

Our objective was to make use of game mechanics to help visualise the dynamics of ecosystems, which are networks of interactions between living organisms. The models used to represent these are made up of sets of differential equations, and are challenging to analyse because they can be high dimensional and non-linear.

The motive for creating a visualisation is to give a platform for players to use their intuitions to interact with the model This is interesting on two fronts. For the player, the game will teach them about how ecosystems work. For the researcher, recording how players use their intuitions to create healthy ecosystems can potentially reveal previously unseen insights into these complex models.

To facilitate this, the game attempts to show the user the workings of the model, but in a digestible manner such that they aren't overwhelmed by the underlying complexity. The progression of their ecosystem is shown in real time, and the player is asked to alter their system in order to maximise a score.

The big goal of the project was to present it at this year's Imperial Festival, which was on the 7/8th of May. It gave us quite a unique opportunity to test and gather feedback, which made it clear that there's a lot of potential for future work, and these possibilities will be reviewed in the concluding evaluation.

The project was a joint effort between me and Orestes Gutierrez, a student in the Life Sciences department at Imperial, over at the Silwood Park campus. He provided guidance through the mathematics of the model that I implemented in the game, with help from his supervisor, Samraat Pawar.

The implementation was designed and built in the Unity Game Engine, which is a popular game development tool. As it's capable of compiling into a HTML5 WebGL player, the game can be played within a browser without having to download any extra plugins; you can play this version right now at www.ecobuildergame.org.

2 Background

The idea for the game was born when my supervisor Dan watched a BBC documentary on the wolves of Yellowstone Park, which told the story of how a wolf reintroduction initiative helped to heal its ecosystem. In 1914, the US Army started hunting wolves in the park as part of a predator control program put in place "under political pressure from the western cattle and livestock industries" [1]. By 1926 they had hunted the wolves to local extinction, and the ecosystem's health declined. Removing the wolves had a sort of butterfly effect. For example, one of the big changes was that their loss relieved a huge amount of predatory pressure from elk, which let their population rise out of control, leading to overconsumption by elk of important areas of vegetation, which in turn degraded the entire system due to the loss of these plants that, as primary producers, acted as a foundation for a lot of other animals to survive [1].

The wolves were reintroduced starting from 1995, and the health of the ecosystem has greatly improved since then [2]. On the surface, it seems counterintuitive that introducing a predator would help other species succeed, but looking a little deeper can reveal all kinds of interesting outcomes. This kind of effect, where a change to a species can affect others that aren't directly interacting with it, is called a trophic cascade, and it's this hidden depth and consequence within ecosystems that became a big part of what we wanted to capture. Dan thought, and I agreed, that being able to play with your own ecosystem in a similar manner would make a great game, and this was the concept that acted as the direction for the entire project.

This inclusion of ecology was also where the Life Sciences department came in. How ecosystems are modelled is quite far out of scope from the EEE syllabus, so the project was set up as a collaboration between a student from each department, me and Orestes Gutierrez. I worked on everything to do with implementing the model and visualisation as a game, and he worked by simulating in Python to research the model in more depth.

At the beginning of 2016, we then applied to be a part of the Imperial Festival. We were lucky enough to be accepted, and this gave us a solid goal to work towards. It's fortunate that Imperial is situated right next to museums like the Natural History and Science Museums, and taking advantage of that to show the project to the general public was a unique opportunity to test the game on a large audience. I'll outline our experience with the Festival in detail in the Testing section of this report, along with the corresponding graphs and statistics in the Results section.

The technical background to the project that follows consists of two parts: the model, and the tool. The model will cover the mathematics behind the simulation of the ecosystem, and the tool will introduce the Unity game engine, along with any of its relevant features. Afterwards, there will also be an exploration of similar games that others have made before, along with a discussion work on how gaming has previously been used in the world of teaching.

2.1 The Model

We're using Lotka-Volterra type equations to model population changes in our ecosystem. [3]

$$\frac{dx_i}{dt} = x_i \left(b_i - d_i + \sum_{j=0}^{n} a_{ij} x_j \right), i = 1, 2, ..., n \tag{2.1}$$

x_i in this equation represents the population of a species. It is multiplied by its birth rate b_i, its death rate d_i, and the sum of products of every interaction rate a_{ij} and its corresponding species' population x_j. You can imagine each of these terms as the animal reproducing, dying, competing for space, and hunting or being hunted, respectively. The collection of a coefficients that describe interactions between every species can be arranged in a matrix known as the community matrix.

There are a few interesting points to note about this equation, one of which being that species have a coefficient a_{ii} which represents interaction with itself. This is because any given species will be in competition with itself for resources such as space. Another point of interest is that a_{ij} does not need to have the opposite sign to a_{ji}. This means that each interaction is not as simple as a predator-prey relationship; if both signs are positive, it means that the two species coexist symbiotically, and if both are negative, then the two species kill each other without consumption to produce a gain in biomass.

An example of the former would be fungi and algae, collectively known as lichens, where the algae eat fungi that grow through photosynthesis, and the algae benefit from being protected from the environment in the filaments of the fungus [4]. An example of the latter would be dolphins killing porpoises seemingly for fun, and leaving them to wash up ashore [5], or even humans hunting animals for sport. Both of these situations could provide excellent teaching points inside the game for the user as they see interesting new interactions between species inside their ecosystems.

To accurately model a real ecosystem, the parameters b, d, a, must be very carefully chosen, or else the system fluctuates wildly and often becomes unstable. This is where Orestes would contribute by helping with the mathematics, as his supervisor, Samraat Pawar, is behind a lot of research on how to accurately model real ecosystems using this Lotka-Volterra type of model. For birth and death rates, we will use

$$b_i = b_0 m_i^{\beta - 1}$$

and

$$d_i = d_0 m_i^{\beta-1}$$

where b_0 and d_0 are normalisation constants, m_i is the species' average adult body mass, and $\beta = \frac{3}{4}$. These relationships have been used in the past, and are empirically well supported [6] [7]. For the coefficient of interaction between species that have a predator-prey relationship, we will use

$$a_{ij} = -a_0 m_j^{-0.25} \varphi_{ij}$$

and

$$a_{ji} = e a_0 m_j^{-0.25} \varphi_{ij}$$

for prey and predator respectively, where a_0 is a normalisation constant, m_j is the mass of the consumer, and φ_{ij} is the attack success probability, defined as

$$\varphi_{ij} = exp\left(-(slog(m_j m_i^{-1}/k))^2\right)$$

which is a unimodal gaussian relationship between the predator and prey's body mass [3] [8].

Finally, the interaction between a species and itself is defined as

$$a_{ii} = -a_{ii,0} m_i^{-\gamma}$$

where $a_{ii,0}$ is the normalisation constant for that species, and γ is a scaling exponent between $\frac{1}{4}$ and $\frac{1}{2}$ [9].

These parameterisations are proven empirically to provide a good model of an animal ecosystem [3]. The real world is more complex than a set of differential equations, but for the purposes of this project it provides a coherent environment for the user to learn about the dynamics between the species involved.

Lastly, the model then needs a method to perform numerical integration on these equations. The one chosen was the fourth order Runge-Kutta method, which will be explained further in the Implementation Section.

2.2 The Tool

The decision to use a third party game engine as the platform to develop on was an easy one. Designing a proprietary engine has its benefits, such as having complete control over what features are implemented, but this would have been a huge task to take on, and so the only choice was to take advantage of one of the engines already on the market. The immediate choice is Unity, as it takes a 45% share of the market, which is approximately 3 times as much as its nearest competitor [10]. This was also our final choice, but as popular as it is, it's still important to consider other game engines to make the right choice.

I looked mainly at two others before making a final decision: 'GameMaker: Studio' and 'Unreal Engine 4'. GameMaker has been used to create games such as 'Hotline Miami' by Dennaton Games, and 'Undertale' by Toby Fox [11]. It is a very simple tool to use, and is event driven, i.e. you define events, such as button presses, and give instructions, such as movement in a direction, when the events are triggered. These are often not written in code, and whenever code is required, it's written in a language unique to the engine, called 'Game Maker Language' [12]. This makes the development time very quick, but restricts how much control the developer has at lower levels. It's very specialised towards making traditional 2D games [13], and is very good at doing so, but I decided against it in the end due to the restrictions that it would have placed on programming the mathematical model under the hood.

The Unreal Engine is the other big player in the game engine market, with hugely popular games such as 'Batman: Arkham Asylum' and 'Gears of War' using it [14]. It's extremely powerful, with complex graphics rendering features optimised to make it suitable for the most demanding applications that video games require [15]. The scripting for unreal is done in C++, and gives very low level access to engine features for optimisation of code [2]. However since we're targeting a mobile platform, there's no need for the graphical fidelity that the Unreal Engine provides, and the extra development time that it would entail wouldn't be worth it. Similarly to GameMaker, Unreal is very good at what it specialises in, which is high end, high budget games, and is unnecessarily powerful for this project.

Unity is a good balance between the two, and the reason it's so popular is a combination of flexibility, and good documentation. It's able to compile to almost any platform you'd want to for a game, and the creators acknowledge the benefits of being easily accessible by producing free, professional tutorials on everything from specific physics engine features, to videos on how to make an entire game from start to finish. Even the documentation on the scripting API has the option to look at either just a list of member functions/variables, or a more detailed description on how to use the class you're looking at. The scripting can be written in UnityScript, Unity's proprietary language that looks like JavaScript syntactically, or C#. It also has deprecated support for Boo, a language syntactically similar to Python that's compatible with .NET and Mono frameworks [16].

Unity isn't perfect. It's not the most accessible platform for making mobile games, with programs like GameMaker or GameSalad filling that niche. It also isn't as powerful graphically as Unreal, but the amount of options that Unity provides is key, because the end version of this project will change over time. For example, the interface is currently in 2D, but 3D could end up being a more intuitive visualisation; Unity can do things like this all within the same framework, and this flexibility is the reason why we're using it.

I'll be writing scripts in C#, because it's easier to pick up from my C++ background. It's also faster than UnityScript, which is important for performance in case the integration algorithm in the model ends up being computationally expensive.

2.2.1 Unity

The interface for Unity is split into two parts, the editor and scripting, and the idea behind the workflow is that anything you create in the editor can also be done inside a script. This is because they both act as wrappers that interface with the engine's API. The reason there are two components that do the same thing, is that writing code inside a script is very tedious for a lot of common tasks when creating a game.

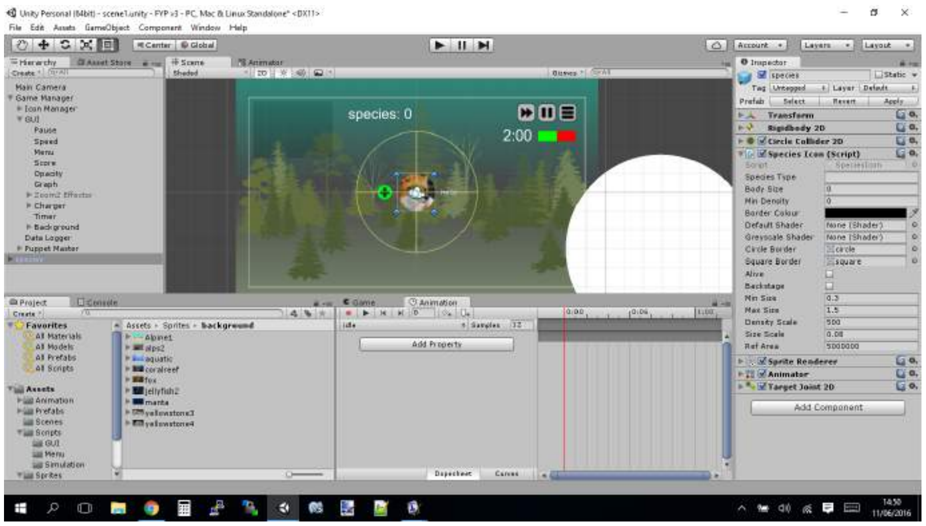
A good example is the `AnimatorController`. This is what Unity uses to decide what animation to play for an object in the game, depending on the state it's in. It does this using what's called an animation state machine, which works similarly to a finite state machine. If your character is standing still, playing an idle animation, and you want to switch to a running animation as the character is moving, you'd create two states with the two animations. You'd then assign the transitions to a condition on a variable stored inside the character; this could be a boolean that represents a button being pressed turning true, or a float that represents your character's speed going over a threshold value.

Writing each of these states and transitions inside a script is long-winded and unintuitive, but the editor lets you drag and drop states straight into a flow chart, and it even highlights transitions between states as the game is being playtested in the editor.

Doing things in the editor makes a lot of the development process easier, but to manipulate the data behind the scenes, you need to write code. To supplement this, another nice feature editor is that you can change public variables as the game is being played. So if you wanted to, say, change a species' population in real time, you would make it a public variable, and when the object it's attached to is selected in the editor, the description shows a text box where you can change its value.

An interesting side note is that the core engine is written in C/C++, but since the scripting is in C# and the GUI is actually written in C# too [17], how does it interface with the engine? The answer is Mono, "an open source implementation of Microsoft's .NET Framework" [18], that compiles the C# in your game into the native C/C++ for the platform you're targeting [19]. Mono is reason why Unity is able to export to so many platforms while still being able to take advantage of the benefits of C#, such as garbage collection and the .NET Framework class library.

This additional level of abstraction in C# makes it more accessible than C/C++, making it more suitable for higher level game design, which focuses on creating gameplay features that function well without worrying too much about performance. Code for the core engine is better suited to C/C++, as it prioritises performance for components like the graphics pipeline or physics engine. If you want to add features to the core engine, you can even write your own C++ DLLs and C# wrappers to import them into Unity [20].



species — Static
Tag: Untagged — Layer: Default
Prefab: Select Revert Apply
Transform
Rigidbody 2D
Circle Collider 2D
Species Icon (Script)
Script: SpeciesIcon
Species Type:
Body Size: 0
Min Density: 0
Border Colour:
Default Shader: None (Shader)
Greyscale Shader: None (Shader)
Circle Border: circle
Square Border: square
Alive: ☐
Backstage: ☐
Min Size: 0.3
Max Size: 1.5
Density Scale: 500
Size Scale: 0.08
Ref Area: 5000000
Sprite Renderer
Animator
Target Joint 2D

Figure 2.1: The Unity Editor GUI

2.2.2 The Editor

The image above is a screenshot of the editor GUI, and this section will explain some of the features relevant to my game. Note that these are features concerning the editor in 2D mode, which I've been using so far. All of this information is cited from the official Unity documentation [21].

Scenes and GameObjects

The basis for all games in Unity are Scenes, which hold `GameObject`s, the base object for all components of your game. `GameObject`s are everything, from moving characters, to the floors they stand on; from the camera the world view is projected from, to sound effects and where they originate from. How a `GameObject` becomes so many things, is by attaching what are called Components, which give a `GameObject` useful properties. For example, if you attach a Component called a `SpriteRenderer`, then the `GameObject` gains the ability to render a sprite, and will now appear on screen in your game. Doing this is as simple as creating an empty object from the `GameObject` drop-down menu, and pressing the big 'Add Component' button in the inspector window.

Sprites and Animation

`Sprite`s are the main method of rendering pictures on screen for a 2D game, and the editor provides various tools to make importing them easier. You can import any image to use as a sprite, and the standard workflow is to place lots of sprites inside one image file, as the tool provides multiple options on how to cut up the image into multiple sprites. An issue to note is that the engine is set by default to smooth the edges of sprite by interpolating using a bilinear filter, which can lead to tearing around the borders between sprites. This option can be turned off, and needs to be if you want to tile sprites next to each other.

These sprites are then used as keyframes within `Animations`, and can then be tied to states within an `AnimatorController`, which were explained earlier. Another useful feature of the `AnimatorController` is the blend tree, which allow a group of different animations to be tied to one state, with each one being activated with different values of a variable. An example of this would be if you want to have different animations for different movement speed thresholds.

Prefabs

Just like how it's redundant to rewrite functions in code, there are times when you'll want to reuse `GameObjects` that you created in the GUI. The way the editor facilitates this is Prefabs. If you create a `GameObject` that you'd like to reuse, then you can drag it from the Hierarchy window into the Project window and it will be saved. You can then drag it back into any other scene and it will create a copy of the original; this is known as instantiating the `GameObject`.

However the real power of Prefabs is that they can be instantiated inside scripts which means that you can automate the instantiation inside code. One good use of this is if you want to randomise where enemies spawn in a level, something that you can't do just using the editor. The way this works is that you can declare a variable of type `GameObject` inside a script, and if it's declared as public, then in the editor you can assign it as a Prefab (which is just a `GameObject` in the eyes of the script). This Prefab can then be instantiated with the `Instantiate()` function, and any object derived from a base `GameObject` can be cloned in this way. A more in-depth discussion of Prefabs can be found at [22].

2.2.3 Scripting

This section will go into detail about the `UnityEngine` API, and how it's implemented within scripts in a game. The philosophy behind programming for Unity is that there's no concept of where your 'main' function is. The structure of the code is decentralised; scripts are seen as another component that can be attached to a `GameObject`, and each script attached to an instantiated `GameObject` has functions that automatically get called at certain times, such as `Update()`, that gets called once a frame.

MonoBehaviour

C# is an object oriented language, so it makes sense that every script contains a class. In Unity, every script attached to a `GameObject` inherits from the class `MonoBehaviour`, which contains the functions that get called by the engine. The main ones are: `Awake()`, which gets called as soon as the `GameObject` is instantiated; `Start()`, which gets called when the `GameObject` is enabled, and always after `Awake()`; `Update()`, which gets called once per frame; and `FixedUpdate()`, which gets called "every fixed framerate frame" [23], meaning that it gets called at fixed timesteps regardless of the frame rate. This is used for physics updates, as the physics engine works with a fixed time difference between updates of the game world; these act in sync with the fixed timesteps of `FixedUpdate()`.

Physics

The physics engine in Unity acts upon the `RigidBody` component. Once it's attached to a `GameObject`, it's assigned a mass and drag coefficients, and is acted upon by forces such as gravity. A `RigidBody` also has a boolean called `isKinematic` which, when set to true, means the object's position is unaffected by the physics engine, but it can still affect other `RigidBodies` through collisions. An example of this would be a platform that a character stands on top of, but shouldn't be able to move by pushing it.

To interact with other `GameObjects`, a `Collider` component is used. This gives it a 'shape', upon which collisions in the physics engine are calculated. The default shapes, a circle and box, give the best performance, but a `Collider` can take any shape drawn by a set of vertices by the user. They can also be assigned a material, which affects properties such as friction between other `Colliders`.

`Collider`s also have a boolean called `isTrigger` which, when true, allow other objects to pass through the `Collider`, and activate functions such as `OnTriggerEnter()` when another `Collider` overlaps with it. An example of this being useful is if you want to spawn an enemy when the player enters a room, by placing such a `Collider` at the entrance.

They can also be 'searched' for, through the use of `Raycast`s. Similar to the concept of ray tracing, you can shoot a ray through the scene and return the first `Collider` it hits, if there exists one in the direction it was shot. An example of this in use would be detecting what object a bullet would hit when shot in a scene.

One last feature of the physics engine are `Joint`s and `Effector`s. `Joint`s allow `RigidBody` components to be attached to each other in a number of ways, such as by a spring or a hinge. `Effector`s give `Collider`s extra properties, such as an `AreaEffector` applying a force to everything inside a `Collider`, or a `BuoyancyEffector` giving the effect of being a fluid, making other `Collider` float and slow down within it.

GameObject

Since each script is a component attached to a `GameObject`, it makes sense for the other components attached to the `GameObject` to be accessible from the script. This is also made possible through the inherited `MonoBehaviour` class, which has the member variable `gameObject`, that references the `GameObject` the script is attached to. From there you can then reach other components, for example to get a reference to a `SpriteRenderer`, you'd call the function like "`myRenderer = gameObject.GetComponent<SpriteRenderer>();`". You can also add components like `AddComponent<SpriteRenderer>()`, which does the same thing as pressing the 'Add Component' button in the Inspector window. Using functions like these, along with `Instantiate()`, you can start to see the how actions in the editor all have an equivalent in code.

Coroutines
Coroutines are similar to threads in Unity, and are described as “like a function that has the ability to pause execution and return control to Unity but then to continue where it left off on the following frame” [24]. A typical use case of this is to pause in the middle of each iteration in a for loop, spreading its execution over multiple frames, which could be used for things like gradually changing the size of an object over the course of several frames.

WebGL
A WebGL player is one of the platforms that we’re targeting, and the way it works is both interesting and relevant to the game so I’ll briefly go over it here. WebGL is a JavaScript implementation of OpenGL, allowing access to graphics pipelines through the browser. This means that an application can run directly inside the browser without any extra downloads, which can be cumbersome and unsafe for the user, making it greatly more accessible as now only a standard browser is needed.

This is relevant because it means that the game will typically run at around half speed on a chrome browser [25], and will also not have access to as much RAM on the players computer, and so more consideration needs to be placed into both performance and memory usage.

One of the ways some browsers achieve such good performance from JavaScript is asm.js, which is a subset of JavaScript written in such a way that forces faster features, such as explicitly typing as an integer. It may seem strange to have C compiled into asm.js, which is technically moving to a higher level language and then back down again, but it leads to a considerable gain in portability due to the prevalence of web browsers, at relatively low cost in performance.

2.3 Related Work

Since showing the user how and why an ecosystem works is the objective of the game, it’s also important to do some background on how similar topics have been taught in the past.

A search for the word ’ecosystem’ in the national curriculum for science on the gov.uk website [26], shows that ecosystems are taught at Key Stage 3 with the description “the interdependence of organisms in an ecosystem, including food webs and insect pollinated crops”, and further in Key Stage 4 (GCSE) as an entire topic called ’Ecosystems’, with learning objectives such as “levels of organisation within an ecosystem” and “the importance of interactions between organisms in a community”. I know that I was taught about foodwebs in class at school, and to me it was one of the more relevant topics to the real world; it’s important for students to not only have awareness of environmental issues around us, but to know the reasons why they exist.

2.3.1 Similar Games

A search for other games with ecosystems as the theme finds a lot of learning games with interactive lectures followed by a quiz, such as BBC Bitesize [27] and Scholastic [28]. These serve as direct revision tools to study for exams, which is related to our game, but the core concept differs as there is no model being simulated.

There also exist games that allow interacting with an ecosystem such as 'Jungle Jeopardy' by PBS, where you select 5 species to add per day and attempt to keep as many alive as possible [29]. This game has a similar idea to this project, but doesn't explicitly give the user details of why changes are happening between timesteps, it just says what eats what. This is presumably to provide abstraction so that the user isn't confused, but the lack of information hinders what the user can learn. The lesson to be taken from this is to provide the user with the means to access all relevant information about the system, but not to force it upon them if it would cause overcrowding and confusion in the interface. The only way to know will be to let people play the game and to get their opinions, and this was a key process in development.

There is also a high budget game called 'Eco' that's currently in the alpha stages of development and was funded by Kickstarter [30]. It's a cooperative multiplayer game, and the underlying ecosystem in the game is based on models from data on biomes in the Pacific North West; this relationship to the real world gives it similarities to our game. However the goal is to create a healthy civilisation between a whole classroom of students playing the game, and the learning objective is the importance of making decisions together, together with the impact that those decisions can have, by allowing them to pass laws within the game only if they all agree. It's a very impressive project, with a strong focus on teaching by providing tools to teachers to use in classrooms to help stimulate this discussion between students. Our game will not have such a wide scope, but it's useful to see the feedback on their forums to help us make the right decisions when encountering similar problems.

I'm also hoping for our game to provide opportunities for tangential learning, which is "the process by which people will self-educate if a topic is exposed to them in a context that they already enjoy" [31], for example a player wanting to learn to play a musical instrument after playing a game like Guitar Hero. In the case of this game, this would be supplemented further learning by providing links and references to external resources. An example would be unlocking interesting clips from documentaries if a level is completed, and giving the player a link to watch the documentary if they're interested.

2.3.2 Gamification

Since the game is interactive, the project is also relevant to the area of gamification, which is "the use of game mechanics and experience design to digitally engage and motivate people to achieve their goals" [32]. In this case, the goal is learning, and the majority of research into the area is specifically geared towards teaching. One literature review states that “gamification does work, but some caveats exist” [33]. These caveats include a number of papers that find insignificant or negative results, or that the positive effect wears off as the novelty of the game diminishes. It also goes on recognise limitations in the reviewed studies, such as a lack of validated psychometric measurements.

The exact effect that games can have is also complex, as shown by Hamari et al [34]. It separates the effects into skill & challenge, engagement, and immersion, but recognises problems in classification like this, such as immersion being “a manifold construct, conceptualized in terms of sensory immersion, challenge-based immersion and imaginative immersion [35]”. In general, it shows that there’s a relationship between learning performance and effective game mechanics, but it’s hard to pinpoint exactly what element of gameplay is the cause.

The answer probably varies greatly between users, for example introducing competition through a leaderboard will be more engaging for different people, and introducing a narrative to immerse the user more engaging for others. Fortunately, the results in the Hamari paper do show a correlation between perceived engagement and learning, measured by asking questions such as “How hard were you concentrating?” and “Did you feel you were learning?”, respectively. Gamification as an area of research is a very new one, and theory behind how to implement it is still young.

3 Requirements

The requirements for this project were quite open ended, and it was clear from the start that the potential scope is huge. Our initial discussions resulted in a large pool of ideas, and it quickly became a matter of choosing which ones to focus on, and which ones would be desirable additions. This is a common symptom especially of game type projects, as one of the end goals is the player enjoying themselves, and it's as much art as science to work out how this can be achieved. The good side of this, however, is that many features outside the core implementation can be dropped if they either become unnecessary, or start to cause unforeseen complications.

For the sake of clarity, I've split the project into three top level components that fit together to form the whole. Simulation contains everything to do with the mathematics of the model, Gameplay involves how the user interacts with this model, and Visualisation is how the game shows all of this to the user.

So it can be thought of as a stack, with each section building on top of what's below. In practice, they aren't as independent as this, but it serves as a nice structure to understand the general framework of the project.

3.1 Simulation

Base Equations and Integration
The first task was to create a working version of the underlying model. The implementation of a class that can run Lotka-Volterra equations and be used by the game is the foundation upon which everything else is built, along with a routine that performs the ODE integration.

Mass-based Scaling
The next step with regards to the simulation itself was to make it more scalable. For the basic simulation, each new species you'd want to add requires more effort than the last, because each addition has more and more other species to interact with, which means more and more coefficients to come up with values for. The bulk of the coefficients are arranged in the community matrix, so the complexity of adding each new species is $O(n^2)$, which quickly becomes unfeasible to do by hand, even with the matrix generally being sparsely populated.

As mentioned in the Background section, the equations used are written such that these coefficients can be defined using only the mass of each species. This saves the effort of hand coding the interaction rate between each species, meaning that we'd only need to come up with a list of animal names, how much they weigh, and what they eat. This generalisation approach did also have its downsides, which will be discussed in the Balance analysis section.

Temperature
Along with helping to design the model used in the simulation, the other main focus of the project on Orestes' end was to investigate the effects of changing the temperature on a system like this. The angle this would take in the context of the game is global warming, and if it turned out that there were meaningful consequences for the player's ecosystem when the temperature rose, then it would both add depth to the gameplay, and raise awareness for the equivalent real world environmental issue.

It's a great idea in theory, but the potential issue here was that adding this extra component to a model like this is treading new ground, and whether the effects would work as a game mechanic or not was largely out of our control.

3.2 Gameplay

Adding and Removing Species
The primary game mechanic for our simulation was always going to be the introduction or removal of species, to give the player their own small experience of being an ecologist introducing wolves to Yellowstone Park. This was therefore the first gameplay feature to add, and everything else would build on top of how well this worked.

Tutorial
Teaching the user how to play is a standard part of any game, and is particularly important here due to the complexities of the underlying system. It's not trivial to create a framework to tell the user what to do and react according to their actions, and we even found out later at the festival that text isn't all that good a method for conveying information to the user, as it often goes unread. These were interesting observations, and will be discussed in the Evaluation section.

Challenge Modes
And as with any game, the player should also have a challenge to work towards completing in order to keep them engaged. Lots of ideas were thrown around as to how to best do this, including:

- starting with a dying ecosystem, and being asked to save it
- keeping an ecosystem that gradually gets worse due to global warming alive for as long as possible
- having animals randomly migrate into your system, and attempt to keep them alive
- given a healthy ecosystem, and being asked to cull a species indirectly
- an ecosystem that's empty, where the player attempts to make it flourish as fast as possible

- a sandbox mode where the user is free to set their own goals

It was clear from the start that the concept had room for many interesting challenges to give the player, and that it would be unfeasible to implement them all. On the other hand, there was even the possibility for the underlying model to be too complex and inherently not fun; more detail on our process of designing the game to be enjoyable will be in the Gameplay analysis section.

3.3 Visualisation

Graphs
The basic representation of any set of data is a graph, and so this was the first visualisation that we aimed to implement. It also meant that I could easily compare my results with Orestes as the output of his simulation is a graph too. There are graph plotters available in the Unity Asset Store, but I decided to create my own because the ones on the market cost money and have a lot of extra features that the game wouldn't need. The details of how I coded the graph plotter are in the Implementation section.

Species
How to convey the properties of each species, such as its population, body mass, or health, is important so that the user can understand why they behave like they do. The main concern while planning was how to not overwhelm the user, and so we concentrated our efforts on how to keep extraneous information either hidden away or unobtrusive.

How the user interacts with the species must also be as intuitive as possible. To this end we planned to make use of some common constructs that users of modern technology have been conditioned to use, such as the notion of dragging icons around on touch screens.

Interactions
Along with knowing how each separate species behaves, how they interact with each other must also be shown. The interconnectivity between species in ecosystems is quite rich, so it's important for the player to see only the most relevant information at any time.

This can be thought of as how to display the information inside the community matrix, which as mentioned before, grows as $O(n^2)$, quickly becoming too much for to take in at one time. More discussion on visualisation ideas for both species and interactions is in the Appearance analysis section.

3.4 Other Ideas

There were also a lot of other ideas that we considered from the beginning, but were more clearly desirables that would have lower priority. These included, but weren't limited to:

- including additional awareness angles for environmental issues in the game, by having extra game mechanics such as humans hunting lions, or overfishing fish in the sea
- developing two branches of the game - one would be the more ‘game’ version targeted towards teaching and fun, and the other would be more research oriented, with less abstraction and a basic interface whose purpose is to efficiently test intuitions on the general behaviour of the model
- having a mode where, instead of using real species, have random procedural generation of species and their interactions so that size the system is scalable to any size on its own
- exporting and distributing the game to the iOS and Android mobile platforms
- allowing the user to zoom in even further to let them interact with the animals, in a similar vein to the game nintendogs [36]
- introducing multiplayer, so users could either compete for the best ecosystem, or have one trying to save it while the other takes it apart
- adding spatial context to the underlying model [37]

4 Analysis and Design

This section will explain the higher level decisions for the game, before going deeper into detail on the Implementation itself. It'll follow the same general structure as the Requirements, by sticking to the idea of having three main components.

The first thing to note here is that when we got accepted into the Imperial Festival, the type of users that we expected there became our target audience. From going to the festival in previous years, I knew that the visitors are mostly members of the general public that drop by on their visit to the museums, generally families with relatively young children. We therefore targeted most of our design choices here, which led to a less research-oriented final product. However the second largest group at the festival is alumni, and they, along with the more interested parents, would be interested in the research end; this became a focus outside the game itself, by including research detail on posters put up around our stand and on the website.

The way we made and revised these decisions was quite iterative. A benefit of using a game engine like Unity is that it's high enough level to be able to make changes fast, and so smaller features can be quickly implemented for feedback. This process had similarities with an agile development cycle [38], although not in a planned, formal manner. More details on how it evolved, with some screenshots for illustration, are in the Visualisation analysis below.

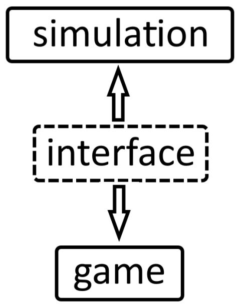
4.1 Simulation

The design of the simulation was based around making collaboration with Orestes as efficient as possible. For the first proof of concept version, the data structures for the model were lumped together with the rest of the code for the game, but as things became more complex, it was clear that separation between the two, so that changes could be made to one without breaking the other, and vice versa.

4.1.1 Object Oriented Programming (OOP)

In the beginning, the plan was to let Orestes code everything on the simulation side, and I'd handle everything on the game side. This requires a method for the two to talk to each other, so one of the first constructs I used was an interface. A C# interface can be thought of as a contract, as it's just a list of function declarations without any implementations, similar to an abstract base class in C++.

In the context of the game, this means that if Orestes and I agree on using the same contract, then I'm guaranteed to be able to call its functions on my end without worrying about any of his changes causing my compiler to complain. It's not guaranteed that the code will do the right thing, but using the same interface makes sure that the two sides fit together.



As the work on the project progressed, the main issue with the plans to split up the work like this was that Orestes doesn't come from a software background at all, and so learning C# and OOP was too big a step for him within the time frame. What we ended up doing was to have him use Python, which he had experience with, and to use the `scipy.integrate.odeint` [39] package to run the simulation on his end. This meant that he would still contribute towards the design of the model and testing its parameters, and I'd take this design and write it in C#.

This did mean that I was working on both sides of my own interface, but this had its benefits too, such as Polymorphism, another OOP principle. This is the idea that a single interface can have different implementations, which can be chosen from by using a different constructor. In the context of the game, this means that I could switch between different versions of the model by simply changing the line of code where it's constructed; for example, I'd often switch between versions with different integration methods to test the difference between them.

4.1.2 Integration

In terms of high level design, there was a decision to make regarding whether to write my own code to integrate ODEs, or to use a library to do it for me. After we decided that I'd handle all the code in C#, I had to make this decision, and in the end chose to write my own.

Initially I did attempt to use an open source ODE solver called Oslo [40], but the installation steps involved having to use Visual Studio, which was a bother as I'd been using MonoDevelop, the editor that ships with Unity. The algorithm for Runge-Kutta integration isn't too complex, and I'd already created a working version for Euler integration at this point, so I decided to try and adapt this instead. For the sake of the less research oriented version we were aiming at for the festival, the game also didn't need the precision of the more heavyweight integration methods in the library, which would've also put more strain on the game to lower the frame rate.

Greater detail on the data structures used for this will be in the Simulation implementation section below, but the design choice was between a graph or a matrix representation. I chose the graph, as it's more intuitive when imagining a foodweb than an arrangement of coefficients in a matrix, and since the community matrix is sparsely populated, it leaves out a lot of unnecessary multiplications with 0 that operations that would have been carried out during the matrix multiplication. Adding a new species is also simpler than in a matrix, as you simply add a new node instead of a whole new row and column. The downside is that having the matrix is a great way of listing coefficients, and the mathematics behind the integration become simpler to code as it performs matrix multiplications to run through the steps.

4.2 Gameplay

This will mainly concern how the design of the game evolved to make the game enjoyable for the player. In my interim report I wrote of concerns about the game being inherently not fun to play:

> *"One very worrying risk is whether it's possible to make the system of differential equations that we're using fun for the player. The mechanics in puzzle games are usually easy to learn, difficult to master, but even learning the concept of how all the differential equations inside the model interact with each other may be too high a barrier of entry to the skill floor where the user will feel challenged, and not frustrated."*

My thoughts on whether or not the final product is fun will be in the Evaluation section; here I'll outline our design process towards that goal.

The first step was to make the system, without any user input, behave well. An example of this would be having things move at a good speed, so that the ecosystem doesn't move so slow that the user gets impatient, or so fast that they don't have enough time to correct bad decisions. I've called this Balance, because it mostly involved tweaking values of parameters until the game reached this Goldilocks state. Focus can then be moved to the Mechanics, which concerns how the game will be played, and how much control is given to the player.

4.2.1 Balance

Parameter Scaling

To make the game play well, we had to tweak constants in the Model, such as b_0, which scales the birthrate of all basal species. Increasing this generally makes the game easier because more biomass is introduced to the ecosystem for the species to consume, and we needed to test values until it felt like the right amount of challenge for the user. This section could have gone into the Simulation analysis above, but I placed it here because the end goal was to improve gameplay.

The challenge here was that the values can't be tested independently; for example, how fast a species dies on its own could feel good for some value of d_0, but then once inter-species interactions are involved, the interaction rate a_0 will also contribute to that species dying. Our approach to finding good values was to start off with some past values that have been known to work in similar models, and to use trial and error to fit them to the game. Orestes would be able to test parameters quickly on his end, which could then be plugged into the game.

This was probably the most long-winded part of designing game design as there was no way to analyse what values would feel good without testing them. To skip the process of compiling the code for each set of values, I placed a secret debug menu in the start screen. If you click around the top left corner, then you'll see this overlay, where you're able to change the values by dragging the sliders to the right up and down:

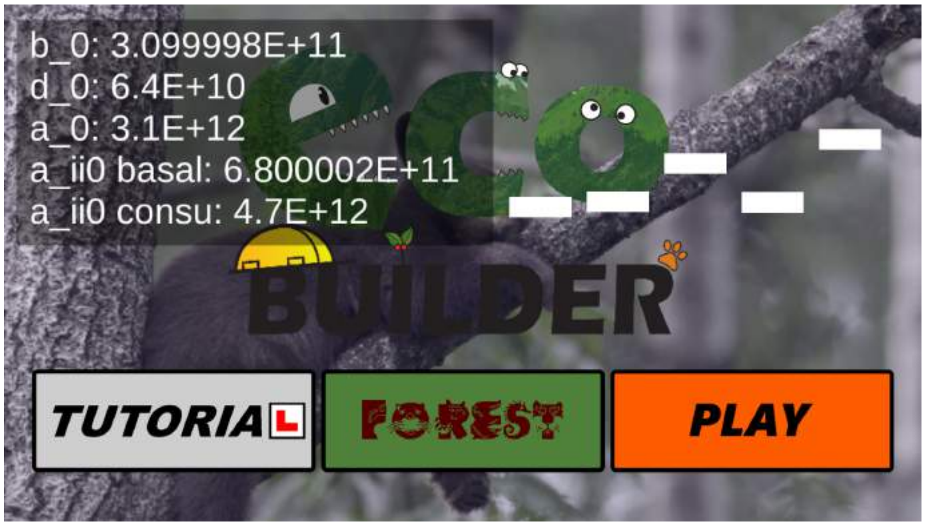

b_0: 3.099998E+11
d_0: 6.4E+10
a_0: 3.1E+12
a_ii0 basal: 6.800002E+11
a_ii0 consu: 4.7E+12

Figure 4.1: secret debug menu to test parameters

There are some more specific tweaks that are worth mentioning. As you can see in the screenshot, there are two values for $a_{ii,0}$. We needed to separate the values for basal and consumer species, because no shared value would give pleasing results. The values of $a_{ii,0}$ are also set very high in general; a higher value makes the system more stable, which made the game simpler to follow and more suited toward the audience at the festival.

This process also revealed the pros and cons of mass based scaling. It generalises the model, but under the assumption that every species of similar weight behaves in the same way. This is clearly not true, for example a bear weighs the same as a deer but should be harder for a predator to hunt, and this incongruity with reality can be confusing.

In a perfect model, real measured values for all of these parameters would be available for each species and every interaction in an environment, but this is understandably impossible as, for each species, the total number of every other species it consumes would need to be tracked. This real world measurement to test model accuracy is a current area of research, known as validation; in fact, experiments are being run over at Silwood using mesocosms [41] to get such measurements for ecosystems on a small scale.

Scoring Metrics

For the score, we tried to find a balance between a number that best represents ecosystem health, but is still comprehensible to the player. The first number we tried was the total biomass contained in the system, as intuitively more biomass equals more life. However, this didn't account for biodiversity, so for that we tried two things: number of species, and flux. Flux in this context is the total amount of biomass transfer, so a system with a lot of interaction between species would have a high flux value.

We also needed a way of combining the metrics together, so our solution for that was to normalise the metrics using a reference high score. To get this reference, we'd play the game and assemble what looked to us like a 'good' ecosystem, and then look at the values at that point. They'd then be combined like so, if we were using all three metrics:

$$\frac{1}{3} \cdot \left(\frac{totalbiomass}{ref.totalbiomass} + \frac{no.of species}{ref.no.of species} + \frac{total flux}{ref.total flux} \right) \tag{4.1}$$

and possibly also taking the log of the biomass and/or flux values as they don't rise linearly. This means that, assuming our reference score was perfect, that their score would be scaled from 0 to 1. We wouldn't expect our score to be perfect, in fact it'd be disappointing if it were so easy to get a perfect score within the complexities of the system, and in that case their score could surpass 1, which would actually be a nice way of telling the user they did well if their score went above 100%.

However in the end we didn't use this method. The reason was that the total biomass of a system actually goes down with respect to the amount of species, because all consumers only remove energy from the system. If one metric goes down as the other goes up, then combining them doesn't give a meaningful result. Because of this, we settled on only using the number of species for the festival, because it's a whole number that's easy to understand, and that was more important for our target audience.

Extinction thresholds

We also needed to decide on a threshold at which a species exits the system, because otherwise a dying species' biomass never actually reaches zero, just tends towards it. Our method was to have a species go extinct when its population of a species dropped below one; the units in system are density of biomass, *kilograms/area*, so to get the dimensionless number for population, we divide by the body mass and multiply by a reference area.

The body mass of the species is known, but the area is arbitrary to the model itself. In our case, we first tried the area of Yellowstone Park, and then just tweaked it until the point of extinction felt right. Population works better as an extinction threshold than directly using density, because smaller species lose their biomass much quicker, and so if the same threshold is used in terms of density, they'd die off too fast compared to the heavier species.

However for the amount of a species added to the system when it's introduced, density works better than population. This is because if, for example, 100 bears and 100 rabbits are introduced to a system, there's a lot more bear in terms of biomass, which makes the mechanic unbalanced.

We also needed a way of telling the user when a species was in danger of dying. In the festival version of the game, the icon of the species flashes red when it's population drops below 10 individuals. This isn't ideal because, as mentioned before, heavier species die slower, which means that their warnings flash last for a lot longer. A solution for the future would be to instead predict the trajectory of its biomass to predict the time of extinction, and then make the species flash when the time is close enough. This trajectory isn't linear so predicting it isn't trivial; a possible solution is to map an exponential, with an added time threshold to prevent oscillation from predicting extinction.

4.2.2 Game Mechanics

Touch Controls

The decision to use touch as the method of interaction was quite an early decision. One of the big design goals was to make the system as intuitive as possible, and interaction through touch was the natural solution.

Despite using touch from the outset, the control scheme made a big change around the middle of development. It didn't feel quite right beforehand, and so we made use of the learned behaviours in smart phones; nowadays a touch screen phone as become an extension of most peoples' arms, and when showing people the game almost everyone tried to drag the little species icons around as soon as they knew it was a touch interface. In the end we removed all the buttons except for the pause, fast forward, and menu buttons, because it felt coherent if all interactions involved dragging.

Time Constraints

In the older versions of the game the player was allowed to add species whenever they wanted to the system, but what tended to happen was that they'd add all the species at once without thinking. This prompted the use of time constraints, and we decided to allow the addition of one species every 5 seconds. On top of that is a 2 minute time limit, and their high score would be recorded after this time ran out. This was actually quite a late addition to the game, and you can see the first version with it in figure 4.7. Adding these constraints was one of the big missing pieces of the puzzle, because it was the change that turned the project from a toy into a game.

Tutorial

To teach the user how to play, a separate mode that shows the user the basics of the game was created. In it there's a text box at the bottom of the screen which guides them through the steps of creating the simplest possible ecosystem, which contains one producer and one consumer. I also wanted the user to learn by doing, so I had the text tell the user to perform certain icons, and had the game wait until they did this to progress further, up until they built their first ecosystem.

Procedural Generation

Another mechanic that was planned early on was to procedurally generate new species. This would mean that the size of the ecosystem in the game could grow forever, instead of using predetermined sets of animals. Some of Orestes' work is also related to this, modelling what's called invasion of species to an ecosystem. This would give unlimited scalability to the system, as the computer would then be able to create new species on its own.

I did implement a rudimentary version of this, which you can see in the below images. To generate a face, I drew sets of different features, such as ears or noses, and chose random combinations of each. For names I had a set of adjectives and a set of animal names to join together. This idea has potential, because everyone I showed it to was entertained by the results, even despite how roughly it was implemented. I think the appeal comes from the randomness giving a uniqueness to every player's game.

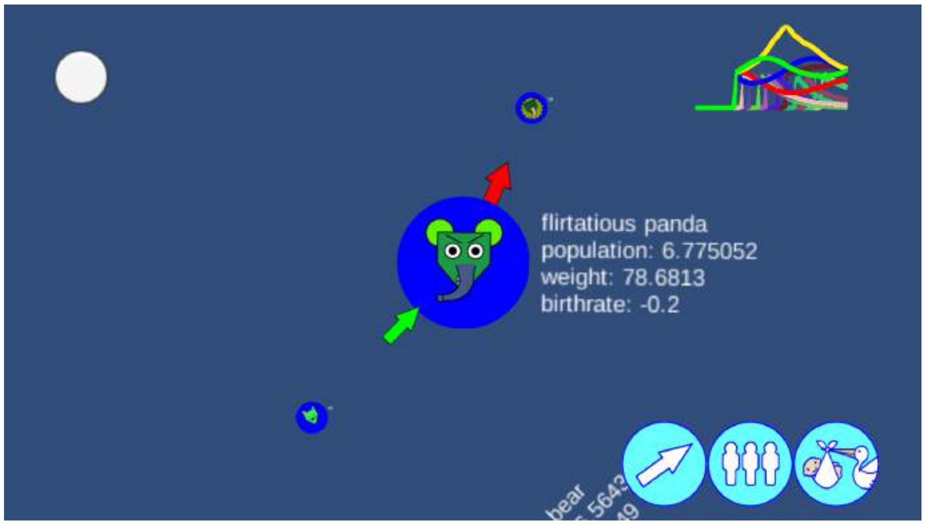


Figure 4.2: An example of a randomly generated animal

We unprioritised this feature, along with many others from the Other Ideas section above, for the festival because our to-do list was already quite long, and it would've been very time-consuming to fix up properly. The decision to use pictures of real animals in the game was at this point too, so having cartoon animals in the mix would have been incoherent.

4.3 Visualisation

Here I'll describe how we designed the way information is presented to the player, separated into two sections: Appearance, which concerns the static elements you'd see in a screenshot, like colours and shapes, and Physics, which is the manner in which things move around on screen.

4.3.1 Appearance

I'll outline the design process of the appearance in this section by including screenshots of the game at various stages of completeness, and by describing the changes from version to version. Below is a mock-up from January that was placed in our application to appear at the Imperial Festival; it's interesting to see what I was roughly aiming for originally in my head, and to compare it with the finished version.

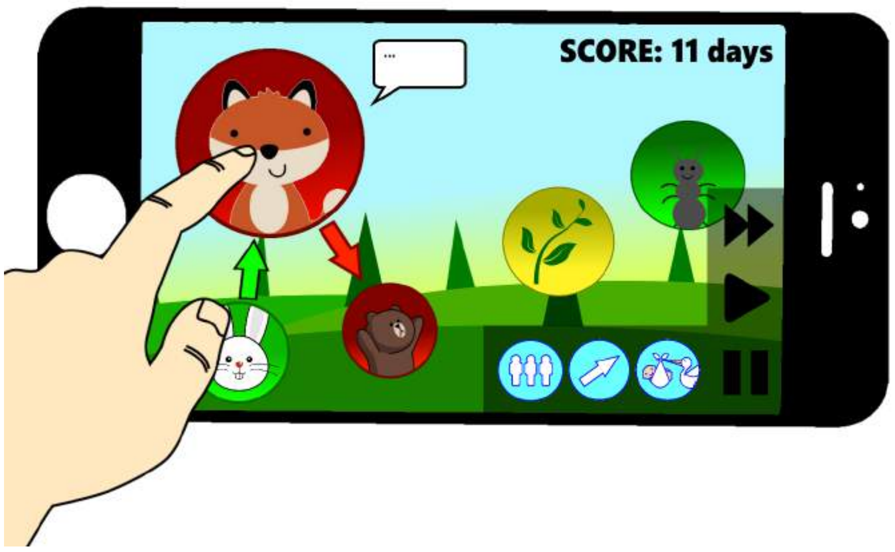


Figure 4.3: Mock-up for our Festival Application

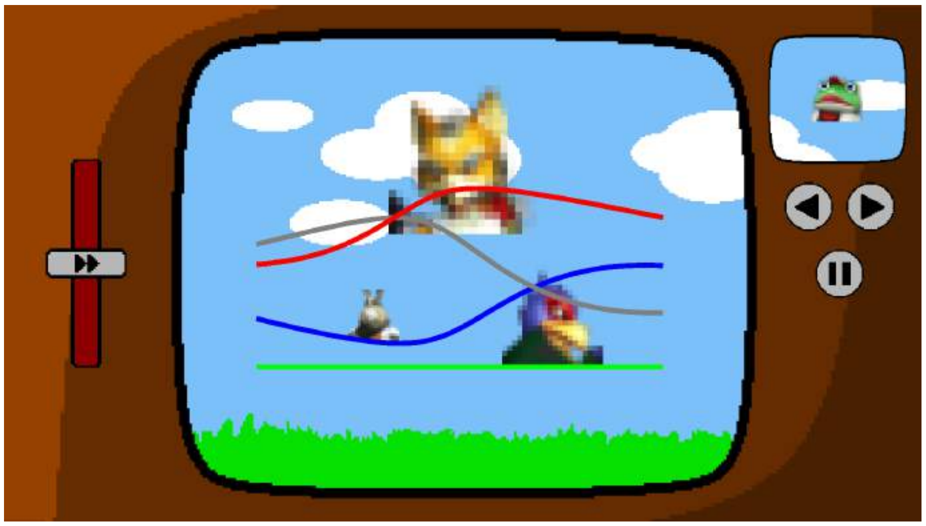


Figure 4.4: **Proof of Concept.** This is what the very first version of the game looked like, from even before the Festival application. The TV frame comes from the fact that the idea for the entire project came from a documentary.

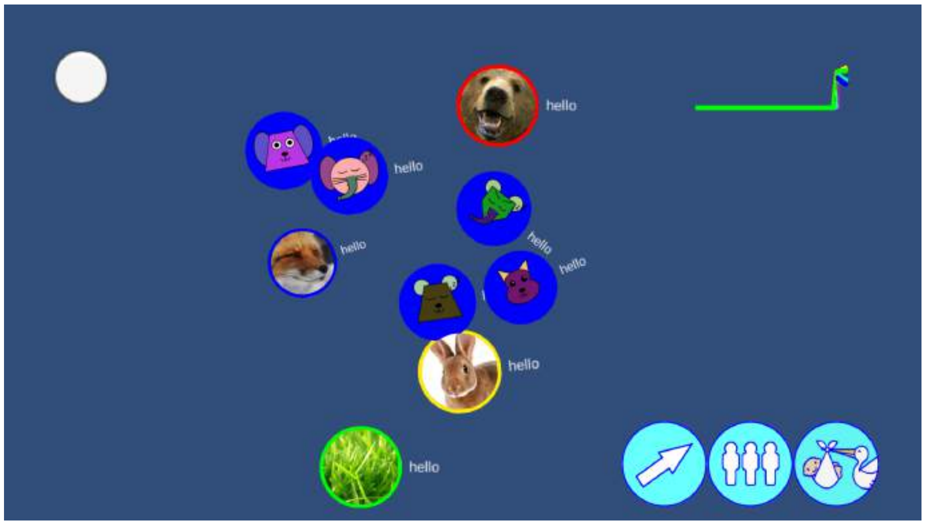


Figure 4.5: **Early Version with Random Generation.** At this point buttons were placed in the bottom right of the screen for the user to press to interact with the ecosystem. The only one that worked was the stork on the right; pressing it introduced a randomly generated animal to the system, and you can see a few of them on screen.

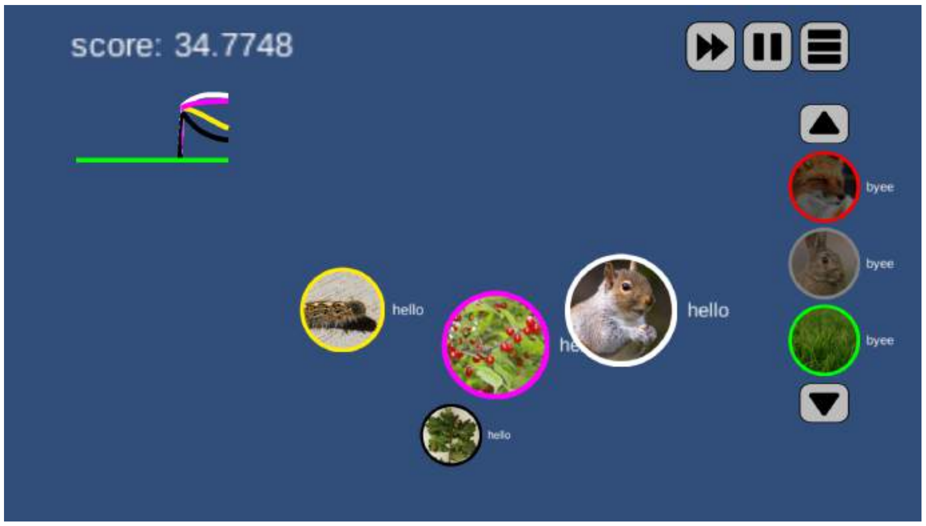
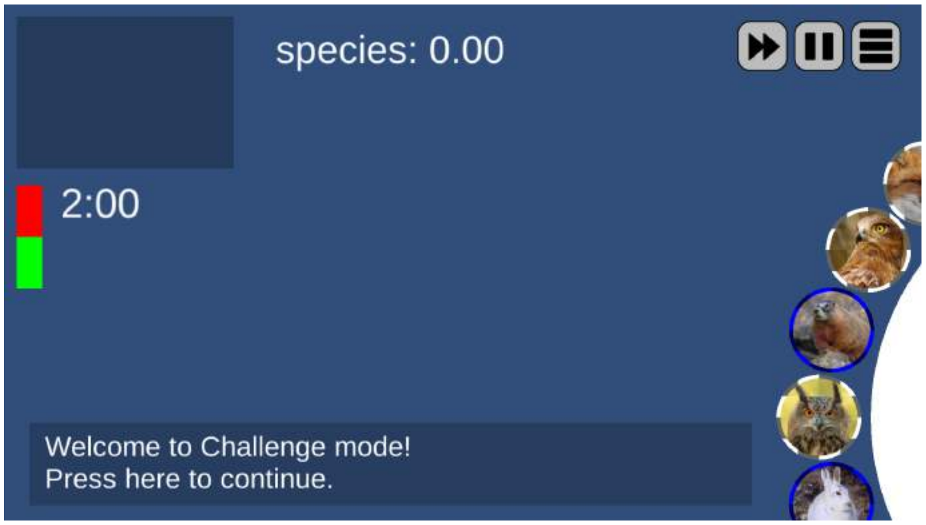


Figure 4.6: **Predetermined Set of Animals Added.** For this iteration, we created a set of 20 animals to choose from, and scrapped the random generation. The buttons on the right scroll through the species by moving them up and down like on a conveyor belt.



Welcome to Challenge mode!
Press here to continue.

Figure 4.7: **Dragging and Tutorial Implemented.** At this point most actions now involved dragging instead of buttons, and the species are now placed on a disc that's also moved through dragging. The time constraints can be seen on the left, and you can also see the tutorial text at the bottom. The indicator of currently extinct species is a chequered border, as the previously darker icons were hard to see.

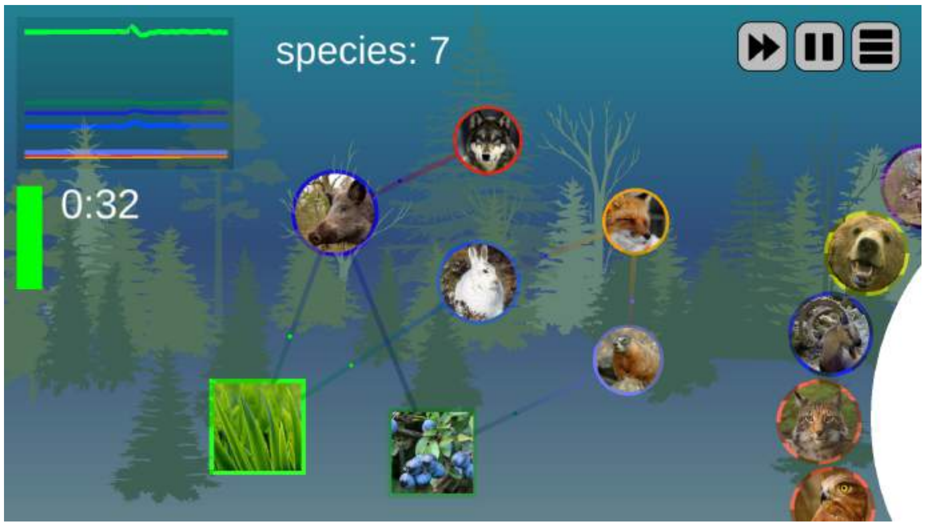


Figure 4.8: **Festival Version.** This is the version of the game we presented at the festival. Adding the background image and lines between icons were subtle differences that made the game look finished. The colour of the borders of each species are also now related to their type, for example plants were green, and carnivores were red.

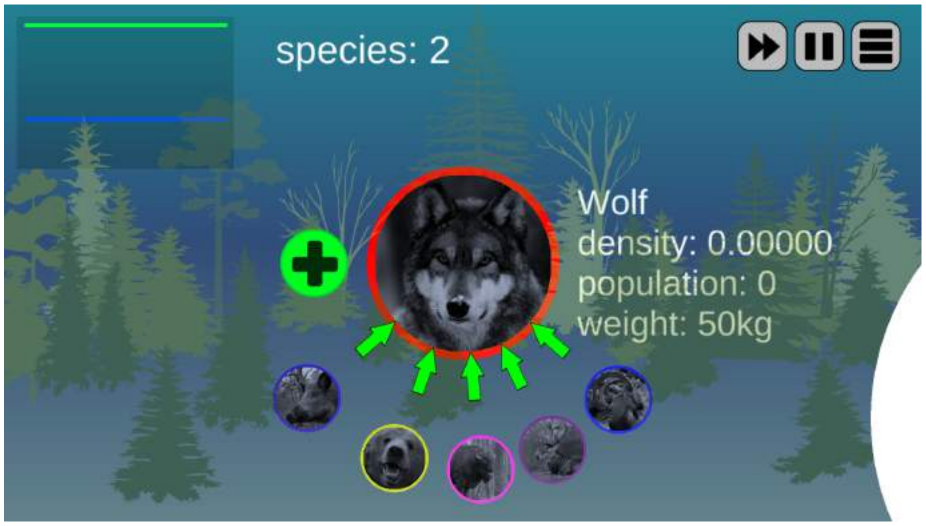


Figure 4.9: **Zoom1.** I call the standard view seen above zoom0, and this is an example of zoom1. If an icon is pressed, it moves into the centre of the screen to show extra information, including its interactions which are indicated by arrows pointing to other icons. These other icons can then also be pressed to traverse the food chain.

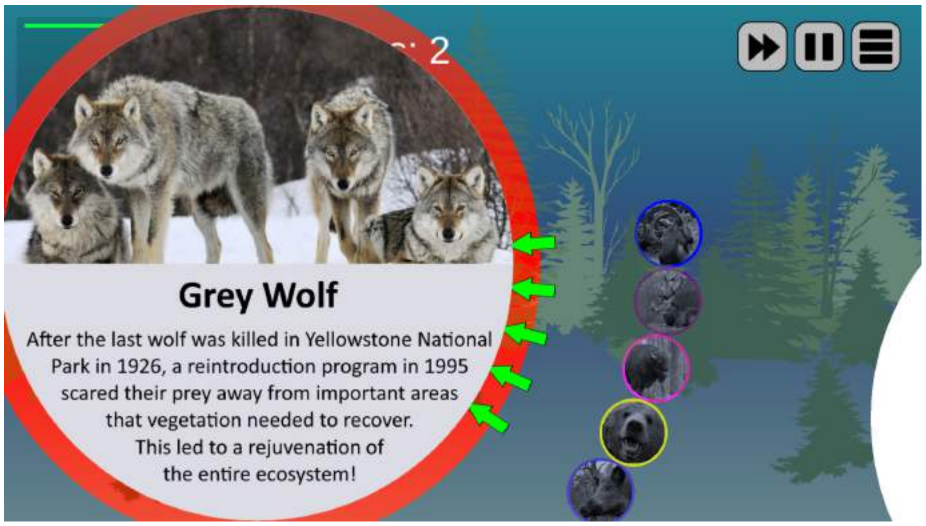

Grey Wolf
After the last wolf was killed in Yellowstone National Park in 1926, a reintroduction program in 1995 scared their prey away from important areas that vegetation needed to recover. This led to a rejuvenation of the entire ecosystem!

Figure 4.10: **Zoom2.** If the same icon is pressed again from zoom1, then the magnification is increased to zoom2. It further enlarges the icon, and the image then turns into another picture of the species, along with some flavour text of a fun fact. The purpose of this was to encourage tangential learning, which is the possibility of the fact intriguing the player enough to go learn on their own outside of the game.

4.3.2 Physics

A lot of effort was placed into making movement feel smooth. Ideally the UI would be satisfying enough such that the player could enjoy just moving things around, even without any underlying model. This idea came from an anecdote from the development of a classic game called Super Mario 64:

> *"one of the most oft-told stories is that in the opening stretch of development all that existed was a moving cuboid, which soon enough was animated and became Mario - more or less as he is in the final product. Depending on which source you read,* [the lead developer] *spent months and months just playing with this character in an open space, or a garden, or early level prototypes, with nothing else to do but move."* [42]

There's quite a difference between EcoBuilder and a platforming game like Mario 64, but the lesson is still applicable. The basic mechanics should feel satisfying just as a toy, without game features such as competition or challenge.

A Heartbeat

Since our game is relevant to living things, I wanted the icons to somehow ‘feel alive’. The way this was done was to not have the icons grow smoothly, but instead to regularly have a sudden change in size, as if the ecosystem’s heart was beating along to a rhythm. This idea had quite pragmatic origins; having the simulation separated from the game through an interface means that every time the simulation is asked for up-to-date values it has to construct a new data structure and pass it over to the game, which is wasteful if done too often. This meant that the rate at the the game updates would be slower than to the simulation anyway, and from that came the idea of slowing it down even further, to give the game a ‘heartbeat’.

It’s usually a good idea to look at past work for inspiration, and this project was no different. A game that became popular around 2015 was agar.io [43], and influence was taken from the physics in that game for my design here. In the game you play as a circular cell in a petri dish that can move around and consume other cells, so there’s already similarities in that both games involve life. The relevant feature is the ability to shoot a little part of yourself into another cell to make them grow bigger, and since also have icons that consume each other, I also made them shoot little circles of the same colour to each other, indicating the movement of biomass.

The way this was implemented was to use the physics engine to attach an invisible spring between the little circles and the predator icons. These predators weigh more, and so the little circles are flung towards them. An unintentional consequence of this came from me not lowering the mass of the little circles as low as intended, which makes the large icons also move a small amount in the opposite direction, making them seem to pulsate in movement as well as size, which is quite aesthetically pleasing.

Moving and Zooming

Almost all of the user’s core interaction involves icons moving around and changing size, so these needed to feel as natural as possible. This meant that these motions should avoid either having discontinuities or being linear, as both of these feel artificial; this is a principle used in animation to make objects feel more realistic in their movement, and this section could just as easily be titled ‘animation’. In general lots of exponentials were used for this and the exact functions used to model the movement will be discussed in the Implementation section.

Icon Placement

Where the icons were placed on screen also posed a challenging problem to solve. Ideally, species that interact with each other would be placed closer together than species far apart in the food chain. However, designing a graph drawing algorithm like this isn’t trivial, and brute forcing the solution isn’t practical, so heuristics were used to get a good, but not optimal solution.

The first method attempted was to try and let the physics system handle everything. If species interacted with each other, I simply placed a small force between them so that they'd gradually move closer together. However it's easy to see the problem here, as a richly connected system would make everything move towards almost everything else, meaning that they all eventually clumped together in the middle.

This was avoided by having the game apply a force that, instead of just pulling icons towards each other, worked to maintain a distance apart, scaled inversely to the strength of the interaction. This meant that strong interactions would work to maintain a shorter distance, and weaker or no interactions a longer one. This worked better, but leaving everything to the physics engine often ended up 'tangling' the system, with icons and forces getting stuck on each other leaving what's .

The next approach to fix the icons' locations in place, and to use heuristics to determine where. The best result I got was from using the body mass of each species to determine its position on the y-axis, as smaller animals tend to be lower on the food chain than larger ones, meaning that vertically sorting by mass generally created a layout close to how they'd line up within trophic levels. For the x axis position, applying forces between interacting species was an option, but randomly choosing a horizontal position gave better results, and also made the layout different for each play, which added novelty to each play.

For the festival, however, everything to do with placement was left up to the user, by fixing the position of the icon to wherever they dragged it to when introducing the species. I expected, perhaps naively, for them to use their intuitions to place the icons in a way that felt right to them, which worked for the more thoughtful players, but often meant that the system ended up becoming clustered in the middle, as that's were it feels natural to drag towards.

4.4 Website

The web is the most accessible place in the world, so the game should on that platform to reach as many people as possible. Unity makes this simple, as exporting to a WebGL build automatically creates the HTML that you need to embed to let users play the game.

We wanted to have our own URL, so we chose ecobuildergame.org, and hosted the website on Imperial web servers. Data should also be logged from online plays too, so we planned to use Google Data analytics and to send statistics from the game into an SQL database. Leaderboards as well as more social media features could then be added to connect players together.

5 Implementation

The mindset used for coding the game was geared towards making full use of the Unity engine, and using the built in functions wherever possible. Thankfully there's good documentation and a strong community behind the Unity API, as there are a lot of quirks to the way it works, like when exactly code is run and how collisions are dealt with. Despite this, in general it's very well designed, and gives the programmer a lot of freedom that stems from having access to the .NET Framework through C#.

I'll continue with the same three part structure as I've used in previous sections, by starting with a description of my code for the underlying simulation, which is kept separate from the Unity engine and its features. Throughout this chapter, classes and functions will be typeset in the style of `MyClass`, so that direct references to code are easier to see.

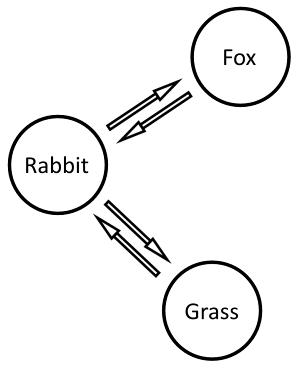
5.1 Simulation

Since the simulation is purely for computing a mathematical model, it makes sense for it to be decoupled from the game engine. In fact, you could take the code for the simulation and run it outside of Unity, which was the original intention in order to let Orestes develop independently.

5.1.1 Data Structures

It was mentioned earlier that a graph tree structure was chosen as the data structure to hold the information, and I'll go into more detail about how this was coded here.

The diagram on the right shows a simple system of three species, where grass grows on its own, rabbits eat grass, and foxes eat rabbits. The shapes represent the different classes inside the code; a circle represents the `Species` class, which acts as the nodes of my graph, and the arrows represent the `Interaction` class, which acts as the edges between nodes.



A couple of iterations were made before settling on the exact structure of the graph, and the final product has the nice property of being able to calculate the new value at each node in one traversal of the tree. In previous iterations the tree had to be traversed twice to write back the new values at each node, and I changed this at the cost of making the edges between nodes bidirectional, which is indicated in the diagram by having arrows pointing both directions.

An important .NET data structure throughout the code is the Dictionary, which is a list of variables indexed by another. In this case, strings were most often used as the indexing type, so that I could have a mapping from the species' name as the string, to some other variable. This was to be able to ask questions such as "what is the current population of rabbit" without having to go through an entire list to find the rabbit, and using a Dictionary gives this operation $O(1)$ access.

A Dictionary was also used to store the connected edges at each node, and the edge contains the interaction rate between the two. The grass node in the diagram would then, for example, contain a Dictionary with only one entry, where an `Interaction` would be indexed by the string "rabbit", and be assigned a negative value because it causes the grass to lose biomass as the prey. The rabbit, on the other hand, would have 2 entries: one indexed by "fox" with a negative value, and one by "grass" with a positive value.

Outside of this graph, I then have another Dictionary that contains references to every species' corresponding node indexed by their name. This means that for the question "what is the interaction rate of foxes towards rabbits", then this outer Dictionary is queried with "fox" to get to the fox's `Species` node, and the inner Dictionary with "rabbit", to get the `Interaction` edge pointing to rabbits, and in there is stored the value.

5.1.2 Mass-based Scaling

A few extra additions have to be made to incorporate body mass into the simulation. Now, in addition to a name, a `Species` also now needs a value for mass. This then simplifies creating an `Interaction`, which now only needs the names of the two nodes it's connecting, and the strength of the interaction calculated from the values of mass now stored inside these nodes.

Previously the strength of each interaction was hand coded, so this step made the system a lot more scalable. However as the number of species in the system increased, it still became less and less convenient to hard code the list of species, so the change was made to read a .csv spreadsheet to make it easier to edit these lists in an external program. This also meant that the task of deciding which animals to put into the ecosystem could be outsourced to Orestes, and he would then be able to sync up the values in his system to by editing the spreadsheet accordingly.

I'll also note here that we didn't manage to incorporate temperature into the simulation in time, as we didn't return desirable consequences for the system in the context of the game. Orestes will be continuing his research on this for his Master's dissertation in the next few months, which will hopefully open the door for future work in this area for the game.

5.1.3 Integration

Having a Dictionary with a reference to every `Species` made it easy to traverse the tree by simply iterating through every entry, because the order of traversal didn't matter in this case. Once at a node, the algorithm simply looks at every `Interaction` it has, and calculates then stores its new value at the end of the timestep.

It can therefore perform the integration in one traversal of the tree, but this is only for first order Euler integration. Runge-Kutta is more complicated as it splits the calculation into four stages, that each require the entire previous stage to be completed before moving on to the next. The equations are:

$$\begin{aligned} y_{n+1} &= y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4), \\ k_1 &= f(y_n), \\ k_2 &= f(y_n + \frac{h}{2}k_1), \\ k_3 &= f(y_n + \frac{h}{2}k_2), \\ k_4 &= f(y_n + hk_3) \end{aligned} \tag{5.1}$$

where y is the species density, and h is the timestep through the integration. This can be thought of as each k value being a slope calculated using the previous slopes, then combined for a weighted average of their respective first-order integrations. The implementation in the game therefore traverses the tree four times to finish one integration timestep, with pseudocode that looks like this:

```
foreach species
    calculate and store k_1
foreach species
    use stored k_1 to calculate and store k_2
foreach species
    use stored k_2 to calculate and store k_3
foreach species
    use stored k_3 to calculate k_4
    use all k values to calculate new y
```

This is four times slower than the Euler method, and also uses more memory to store k values between traversals, but the error is $O(h^4)$, where h is the timestep [44], and so this was a great compromise. Systems that caused instabilities due to integration error at the speeds we wanted to run at now worked without issue, so this is the method we used for the final version.

It's still possible to make the values blow up if large enough timesteps are used in the integration, which does cause the instabilities to come back if the system is sped up enough. This speed is still slow enough to be worth giving as an option to the user, and so in the future an even more accurate method should be used. A method that can be quickly implemented would be to perform the integration multiple times in the same step, for example using half the timestep but integrating twice. This would take twice as long for the same total integration step, but the order of the error means it shrinks by a lot more than twice as much.

At this point we also ran into some issues with datatype precision. The original integration routine used the float datatype because Unity is optimised for its `Mathf` library, which uses floats for things like vector or physics calculations. The approximate range of a float in C# is ±1.5e45 to ±3.4e38 [45], and the numbers being passed around the system actually went lower than this when mass-based scaling was introduced. To remedy this, all the variables being passed around the simulation were changed into the double datatype, and then cast into float when passing the variables to the game.

A final note is that for the sake of the game, some extra variables are stored inside the simulation to keep some more information that could help the user. The total amount of biomass transferred across an `Interaction` is stored so that the user can see how much has been transferred across that link. In the game, this was planned on being used to calculate the amount transferred at each 'heartbeat', which would be used to scale the size of the little circles that move between icons, giving the user extra information on how strong each interaction was.

5.1.4 Interface

As previously mentioned, the simulation and the game were separated, and talked to each other through an interface, and I'll go into more detail on how this communication worked. The code for this interface is in the Appendix.

Unity works like any other game engine, in that its main purpose is to consistently output frames to the player's screen. This means the framerate is also the rate at which most of the code gets run, and so this was also the rate at which the simulation is integrated, by running a timestep through the simulation on each frame. The time difference from the last frame was used as this timestep, and the logic behind this was that the integration would therefore run at the same real world speed independent of the framerate that their machine could handle. However this intrinsically tied the simulation to the game, meaning that if the player's machine slowed down for a few frames, then some extra integration error would be introduced into the calculations. The accuracy of the model was therefore proportional to power of the user's machine.

Keeping this in the game did turn out to be a mistake, as the framerate did occasionally drop enough to introduce instabilities to make the numbers blow up. This was only an issue in the Aquatic system, due to the much larger range of body masses on offer, from plankton to blue whales. We increased the mass of plankton by a few orders of magnitude to prevent this from happening, but this turned out to not be enough, as the players on the day inevitably found combinations of species that brought the errors back. Afterwards this tie between the game and the simulation was removed, and now the same timestep is run through the integration every time.

5.2 Gameplay and Visualisation

For this section the Gameplay and Visualisation components will be grouped together because at this level they become very interdependent, as they're both coded within the same set of C# scripts, making it easier to describe the implementation through their hierarchy in Unity.

5.2.1 Hierarchy

The design for the hierarchy of my code is inspired by one of the Unity tutorials on the official website, namely the "2D Roguelike Tutorial" [46]. In their code, there are scripts called managers, which have control over different aspects of the game, including other managers. In the case of the tutorial, there is what's called a `GameManager` at the top of the hierarchy that controls another manager called the `BoardManager`, which randomly generates a level layout. The `GameManager` tracks top level information such as how many levels the player has completed, and it feeds this to the `BoardManager` which then generates the next level. This structure was used to organise my code inside Unity, resulting in the following tree structure:

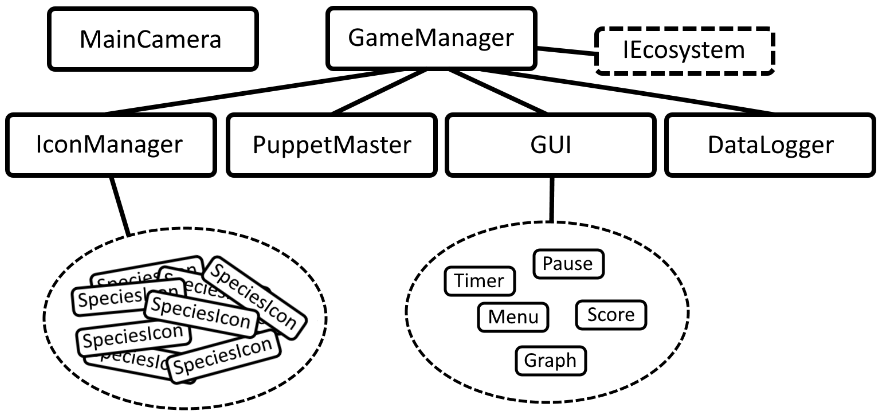


Figure 5.1: Code Hierarchy within Unity

In the diagram, each box with a solid outline represents a `GameObject`. The `GameManager` controls four sub-managers:

- `IconManager`, which controls `SpeciesIcon`s. A `SpeciesIcon` is the icon that represents each species in the system, and the `IconManager` herds them into the positions they need to be when the `GameManager` tells it to
- `PuppetMaster`, which is fed information on the user's actions, so that it can update things like the tutorial text or the timer.

- `GUI`, which contains all the `GameObjects` that are buttons or displays, such as the pause button, or the graph
- `DataLogger`, which is periodically fed data from the simulation to log into text files or send to SQL databases.

Lastly, it has access to the simulation interface called `IEcosystem`. The camera is on its own because it doesn't need to be a direct part of the hierarchy, because if anything needs to interact with it, then its reference can be obtained using `Camera.main`, which is available in the Unity API from any position in the code.

This tree structure works well for a small game like this, but keeping track of all the references you need starts to become tedious if the hierarchy gets big enough. The main issue is that the control flow goes in both directions, for example when a Species Icon is clicked, its script is the one notified, not `GameManager`, meaning that `SpeciesIcon` has to send this information back up the tree using a reference to `GameManager`, which can then control the other managers accordingly.

The control flow starts to become messy if the branches of the tree become long, and gets broken if any objects are shuffled around in the hierarchy. The standard solution to this in Unity is an event system, another feature of the Unity API, which can send or receive messages to and from any `GameObject` in the game without storing a reference [47]. This means that if a `SpeciesIcon` was clicked, instead of having a reference to `GameManager` and calling one of its functions itself, it would simply send a message to the event system, which would then look at it and work out that it needs to send a message to `GameManager`.

This mediator of communication between scripts would remove the need for much of the hierarchy, and would modularise the code to make it more easily organised; tt wasn't used in this case because the size of the code is small enough to manage the references.

5.2.2 Graph

Since a graph was the first visualisation made in Unity, its mechanics will be explained first. The basic way to draw lines in a Unity scene is by using the `LineRenderer` component, which essentially connects the dots; given a list of coordinates along with a thickness and colour, it draws a line between each point in the order it appears in the list. This means that a list of values is easily converted into a line on screen. However the goal wasn't to just plot a static graph, but to have the graph update live as the game progresses.

There are two approaches for how to display this graph. The easy version would be to draw it like an Electrocardiogram (EKG), which is the heart rate monitor commonly seen in hospitals. It works the same way as an oscilloscope, in that it redraws one point at a time, moving across the display from left to right in a loop. The more complicated version is to always draw the new point on the right hand side of the screen, and then move every other point back by one position. The second version was chosen because it's clearer to the user, as the newest value is always on the right, and the oldest always on the left. This was a little harder to code, as the EKG version would just have to replace one value at a time, while this needs to push every other plotted point back.

This was done by using a `Queue` [48], which is a simple first in first out (FIFO) buffer. This made adding a value to the graph simple, as adding a new value was as simple as pushing a value onto the queue, and popping one off at the end. However the game needs to update every value inside the `LineRenderer` when this happens, and so to do this every value in the queue is popped, and simultaneously pushed back into the queue, whilst updating the values in the `LineRenderer` accordingly. This is admittedly not very efficient, and a better solution would be to use an array as a circular buffer. However not that much computation is done here, and the queue was used because an array would have been a fixed size, removing the option to make the graph wider on command. The best solution to both of these problems would have been an implementation of a linked list that can be treated as a queue for adding a new value, while retaining the ability to traverse the list and read the values, unlike in a queue. However this ended up being low on the list of priorities because a functional version was already working at the time.

One last feature of the graph is to be able to scale the y-axis according the the largest value currently plotted. This is because having a fixed height meant that if every species had a low population relative to the scale, then the entire graph would be squished in the y-axis. The important information to the user is the populations of their species relative to each other, and so it makes sense to have the line with the highest population at the top of the graph.

To do this, the largest value in the graph needs to be known at all times, and this was implemented using the `SortedDictionary` .NET class, which is the same as a normal `Dictionary`, except that the entries are sorted with respect to the Keys. This acts as a hash mapping from the values on the graph, to an integer that counts the number appearances of this value. Every time the graph is given a value, it increments the corresponding integer, and also decrements the integer mapped to the value that gets popped off the end of the line. This therefore counts the number of appearances of each value on the graph, and since the dictionary is sorted by these values, a look at the last entry in the dictionary returns the highest value in the entire graph.

This is better than just maintaining a sorted list of every value in the graph, because a dictionary allows for $O(1)$ access to decrement values popped off the end, and you'd have to search through a list in $O(log(n))$ for the same action. The values are also rounded to two decimal places to act as a hash function, because similar values will then be placed into the same entry, keeping the total number of entries lower by letting similar values increment the same integer. The trade off is less precision, but it's more than precise enough for scaling the height of the graph.

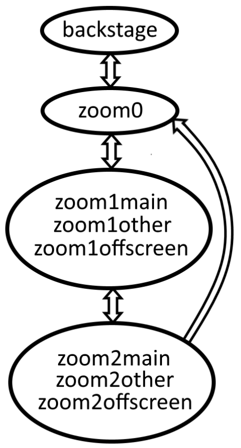
5.2.3 Species Icons

Each species in the game is represented by a `SpeciesIcon`, which is the longest class in the project in terms of length of code. It handles a state machine to keep track of where the icon should be, and also needs to work with things like animation and physics to make it look good.

The `IconManager` is directly above all of these in the hierarchy, and is the one that calls the functions in each icon. For example, if an icon is pressed, then the `GameManager` will be notified, and will tell `IconManager` the name of the species pressed along with how far to zoom in. `IconManager` then iterates through every species to move everything to the right place.

Zoom States

These states are used to keep track of how far zoomed in each icon is. The default state for the game has every icon in zoom0, and when one is pressed, then every icon moves to a zoom1 state. The pressed icon becomes zoom1main, every species that interacts with it becomes zoom1other, and the remaining species become zoom1offscreen. When the zoom1main icon is pressed again, then the icons move into the corresponding zoom2 states in the same way. The scene goes back to zoom0 if the user presses anywhere else on the screen, and backstage is a special state reserved only for the tutorial mode, to ensure that icons the user shouldn't have access to stay off screen.



These states are kept track of using an enumerated type, called `ZoomState`, which essentially maps a list of integers to more readable names that can be used in comparisons. The diagram roughly illustrates this state machine, and it's quite straightforward as the zooming in process is quite one-dimensional, almost like moving a slider to increase the magnification.

Animation

The `Animator` component was used to change elements of an icon's appearance. Animation is also essentially also a state machine, and Unity gives a nice interface to edit them. Screenshots of these are placed below to help describe how they function.

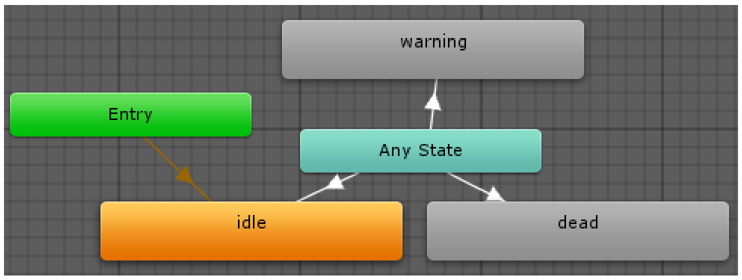


Figure 5.2: Icon Animator

Above is the state machine that controls the icon's appearance. The 'Any State' box is used to interconnect every state together, so that it can transition anywhere at any time. The transitions are caused by 'triggers', so for example if I want to make the icon flash red using the warning state, the function `SetTrigger("warning")` is used, and this causes a transition across that arrow.

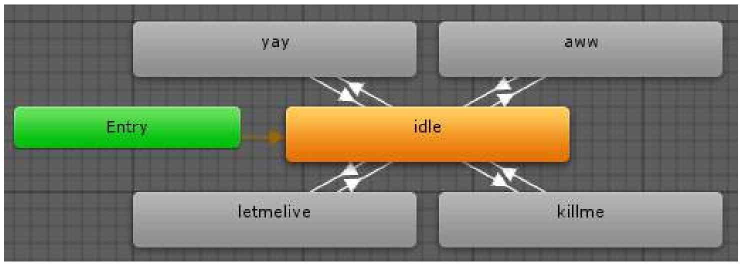


Figure 5.3: Halo Animator

Above is the state machine for controlling the 'halo' of the icons. The user is given more information on when they can perform certain actions, and this is indicated by adding a glow that surrounds the icon whenever they're about to perform an action. For example, when they're able to introduce a species, the transition to the state called 'letmelive' is triggered, and this creates a green glow. If they do introduce the species, it then moves to the 'yay' state, which makes the glow flash white.

In an ideal world the zoom state functionality would be placed entirely inside these state machines, with the transitions between them triggered by the `IconManager`. However, while you can animate things like scale and colour in the editor, there are things that are out of your control, such as shaders which have to be changed in script. It is possible to use Animation Events, which allow the `Animator` to call functions inside scripts, and in future iterations this is how animation would be handled.

Physics

To make the movement of icons feel organic and consistent, as much of the Unity physics engine as possible was used for moving things around. The main feature used to move an icon from A to B was the `SpringJoint2D`, which essentially attaches an invisible spring between two objects.

There are a couple of parameters that one of these springs needs to model its behaviour, and the main ones are distance, frequency, and damping ratio. Distance is the length the spring should attempt to maintain between the two attached objects; frequency is essentially the spring constant in Hooke's law, which determines how stiff the spring is; damping ratio is how much energy the spring loses, which changes how much the spring should oscillate. This was used to make the objects move smoothly towards each other by setting the distance to almost zero, the frequency to a small value, and the damping ratio to 1, which creates critical damping, preventing the objects from bouncing around each other.

There's another class that functions almost exactly like the `SpringJoint2D`, except it's designed to move things towards point the user is specifying, called a `TargetJoint2D`. The difference is that instead of connecting two physics objects together, it's attached to one object, and is pulled towards a position in world space. These two components are used for all movement of icons in the game, so that every motion stays coherent to the same scheme.

Size Scaling

We decided that the size of the icons would change proportional to the total biomass of that species in the system. Since this value ranges across multiple orders of magnitude, its logarithm is used, along with a clamp between a minimum and maximum size. However the icons shouldn't instantly jump in size, and so the following function was used:

$$1 - B_0 exp(-l_0 t) + B_1 exp(-l_1 t) \tag{5.2}$$

to smoothly jump without any discontinuities. The function overshoots a little to stick to the theme of a heartbeat, which expands quickly and then slowly shrinks afterwards. This is why two exponentials are combined, as one causes an overshoot while the other shrinks it back down. The constants $B_{0/1}$ and $l_{0/1}$ were then tweaked until it looked good in the game.

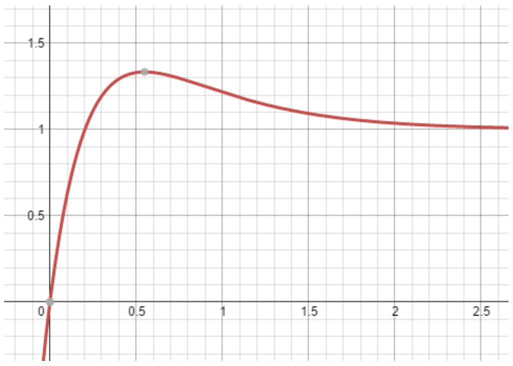


Figure 5.4: $1 - 3 * exp(-5x) + 2 * exp(-2x)$

To make the icons change size over time, a coroutine is used to call different stages of the function over the course of multiple frames, in this case changing the size every frame. One problem with this was the possibility of the user pressing an icon twice in quick succession, which calls another coroutine before the previous one is finished, causing conflict in the calculations between the now concurrent processes. To avoid this, a mutual exclusion (mutex) lock is used to make sure that only one could change the size of the icon at a time. A new coroutine requests access to the lock when it spawns, and when an existing process see this request, it exits to let the new one take over.

Heartbeat

The idea of the system having a Heartbeat is explained in the analysis section, and its implementation will be detailed here. The beat is created using the `InvokeRepeating()` function, which calls a function at regular time intervals. In this case, it's used to grab data from the simulation every 1.5 seconds, and every time this happens, the size of the icons is updated.

The first part of the resulting animation is to shoot a little circle from the prey to the predator to show the direction the biomass moves in. The class that does this is called `TransferIcon`, and it spawns itself at the location of the prey, then shoots towards the predator using an attached `SpringJoint2D`. It then starts a coroutine to gradually shrink the ball into nothingness, which both shows how biomass is lost in the transition, and also eventually makes the circle disappear if it gets stuck on something in transit.

The second part is to connect the two interacting species using a line, and the class that does this is called `TransferLine`. It simply creates a line with `LineRenderer`, which has the option to set different colours for the start and end points of the line to draw a gradient between the two. To get the intended effect, the line starts with the colour of the prey's border, and ends with the colour of the predator's. A coroutine is then started to make the line slowly fade away, by changing the opacity of these colours.

5.2.4 GUI

`GUI` is the class used to hold everything else that the user sees, including things like the background and buttons, and acts as the path through which the `GameManager` talks to these elements. These individual elements, apart from the graph, are very simple in their implementation; each script is only a few dozen lines, mostly only being simple text boxes and buttons that change colour when pressed.

One interesting feature of the GUI is how it hides elements from the user. The Camera uses orthographic projection, and is placed slightly in front of the scene. This means that an object's size doesn't depend on how far away it's placed from the camera, which makes placing objects in front of each other easy by just giving them a smaller z-axis position, which is how UI elements aer placed in front of other objects. It also means that if something is placed behind the camera, it doesn't get drawn at all, which is how certain elements are hidden away when they aren't needed. This was also how gaps are created inside single `LineRenderer` lines, by having the line shoot behind the camera and then back at another point to hide the space in between.

5.2.5 Tutorial & Challenge Modes

The `PuppetMaster` class has control over everything to do with the game elements like the tutorial and the time constraints. It keeps track of the variables such as the time remaining in the challenge, and this is how the control flow generally works: whenever the `GameManager` wants to perform an action, such as introduce a species, it asks the `PuppetMaster` for permission, which returns a yes or a no. If the answer is yes, then the action is carried out, and the state of the game is updated to reflect this.

It's called a puppet master because it also has control over which species are in the game at all. For example in the tutorial mode, it needs to hide away all the species except the ones needed to show the user how to play the game. In hindsight, the hierarchy doesn't really make sense for this, because if the `PuppetMaster` is actually controlling the `GameManager`, then it should probably be above it in the tree.

For the tutorial text itself, coroutines are used to pause execution and wait for user actions before moving on to the next stages. The control flow in the code did get a little messy, because the user has the choice to do a number of actions at any point, and the code has to recognise all of these at all times, and then jump to the correct next stage in the code. At this point `goto`s were used to jump to different parts of the tutorial, as they greatly improved the code's readability over tracking the start and end curly braces of endless if statements.

5.2.6 Logging Data

The `DataLogger` is the class that dumps all the data from gameplay into .txt files. C# makes it very easy to write to a file; all that was needed was to call `System.IO.File.AppendAllText()`, and it would handle the OS. What's written to these files was also reasonably simple:

- a header .txt file which contains the time, mode, and level played
- which species were introduced or removed and at what time
- the location of every click, up and down, that the user input
- the total biomass density for every species, at 0.5 second intervals

The names of these files would be a string that represents the date & time, along with the type of data within. For example one of the files to track clicks, written at 9:53am on the 7th of May was named "`20160507095353clicks.txt`". The data inside the file is then formatted as space separated values, for example the first line of the clicks file above is "`1224.752 7.925926 0.1388889 d`", which means at 1224 seconds into the game being booted up, a click was registered at the (x, y) coordinate (7.93, 0.14), as a down click on the mouse button.

We also had plans to store this data into an SQL database on the Imperial servers, but didn't manage to complete it in time. Inside Unity there is a `WWW` class that sends HTTP requests, and a `WWWForm` class to construct POST data to place inside it. This was necessary since the game runs on the client side, and needs a way to send it back to the server. On the server side we would've then run a php script to insert into the database, and from there the data could even be visualised live for the player.

5.2.7 Assets

Creating assets to use in a game is one of the most important tasks, and also usually the most time consuming, so I'll make a quick note about that here. Luckily in our case the most important sprites were pictures of real animals, so no overly artistic drawing was required. For designing GUI elements such as buttons, I used paint.net [49], which is essentially a very lightweight version of photoshop.

We also made use of vector graphics, which can be scaled without loss, for the logo that you can see on the title page of this report; it was Dan and his wife who kindly designed the logo, and I think it looks great.

In the final version of the game there are over a hundred sprites, and it did take up a good chunk of my time to create them all, especially the larger sprites that contain fun facts about the species. Luckily I had Orestes' help in finding these pictures and facts, or else they might have not all been finished in time.

A final note is that import options when moving these into the game required some care, because Unity is set up as default to not compress textures at all, which led to the WebGL build using far more memory than any browser was willing to allocate. You're able to see the amount of space each component in the game takes up in an editor log file, and choosing the right compression on each imported image shrank the space used by textures from 750mb to 100mb. These logs can be seen in the Appendix.

6 Testing

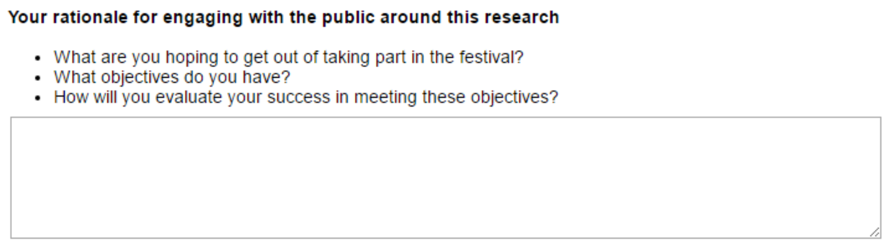
The first step of the testing process is to determine what exactly to assess. Our overall objective had two sides - one was to help teach players about ecosystems, and the other was to help research by using the results to reveal insights into the model. With these in mind, the Imperial Festival was the perfect place to present the game, as it both attracts the general public, and is a big gathering of researchers who can offer novel insights into how best to use the tool we've created.

Our approach towards data was to just record almost everything that happened in the gamed into files, without much too much thought towards saving space or formatting it perfectly, and then deal with it afterwards. By the end of the festival, we had 29.5mb just in .txt files, which corresponded to over thirty million characters, and this section will outline how we went about collecting and then analysing all this data.

6.1 Imperial Festival

I'll start with a brief outline of the process of applying, then preparing and manning our stand for the day. It will be mostly anecdotal, but should serve to give background to our testing process.

The application was relatively simple, and involved filling out an online form [50] with questions such as



Your rationale for engaging with the public around this research

- What are you hoping to get out of taking part in the festival?
- What objectives do you have?
- How will you evaluate your success in meeting these objectives?

Figure 6.1: Example of an Application Question

Since the project was still in its early stages at this point, filling out and thinking about this application was quite good for focussing our direction towards the right objectives.

For the festival itself, we procured two large touch screen monitors for visitors to play the game. We chose these because having large displays with eye-catching images on them would attract visitors better than static posters, or tablets with the game on them. They were set up on tables around a metre off the ground, which ended up being a good height as it was at eye level for children, but not too high that adults couldn't be involved either. Alongside the game, we also had some live insects and microscopes set up as another activity and draw to the stand, and two A0 posters on noticeboards on either side of our stand (which are in the Appendix).

We also got custom magnets made with the logo and images from the game to give away to visitors who played the game. Thankfully, we also had lots of volunteers around the stand to answer questions and help players having trouble.

Figure 6.2: Magnets!

As for quantitative questing outside of the game logging data, we tracked the number of adults and children who visited and/or played on a smartphone app. Qualitatively, thanks to the others helping out at the stand, I was able to stand back and make some notes on observations of typical player behaviour, which will be discussed in the Evaluation section. Simply talking to the more interested visitors and researchers presenting at other stands was also great for getting feedback and ideas about the project.

6.2 Graphing the Data

After our hectic schedule at the Festival was over, the data could then be combed through to extract any interesting information from the heaps of logged text. Python was used to do this because it's very good for quickly writing scripts to run through text, and the excellent `matplotlib.pyplot` package grants the ability to plot graphs as if using MatLab too.

I hadn't used matplotlib before, but it's well documented and, like Unity, is widespread in its use, so almost any question you might have has likely been asked already. The only real issue when parsing the data was corrupted .txt files, which was likely caused by the game either crashing or being turned off at some point, and this was dealt with by catching exceptions in my scripts. The resulting graphs will be in the Results below.

6.3 Results

We manually tracked the number of visitors to our stand on the Saturday, which was the first day, to get the following numbers:

• adult visitors:	202
• adult players:	97
• child visitors:	115
• child players:	102

This method was bound to miss out some visitors, and looking in the dumped data finds a total of **430 plays** with at least one introduced species over both days. If we then take the ratio of visitors to plays from the manual data, then an estimate of the total visitors to our stand was around 1.6 times the number of plays, which gives us **688 visitors**. This was a lot of exposure for the project, and it was great to be able to present it to so many people. Here are some more stats from the dumped data:

• total challenge mode plays:	346
• total completed challenges:	117
• total tutorials played:	107
• total species added:	5317
• total species removed:	293
• total clicks:	30233

Since there were two levels, the forest and the ocean, I've split their data into green and blue charts respectively, starting with a distribution of the scores that players achieved:

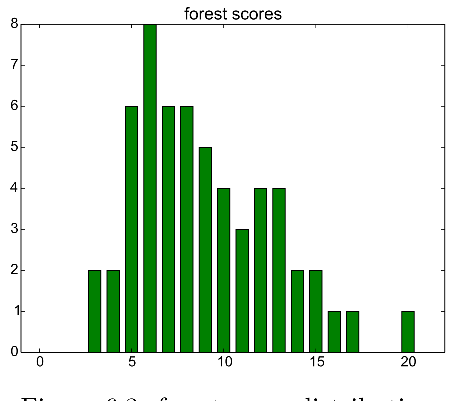


Figure 6.3: forest score distribution

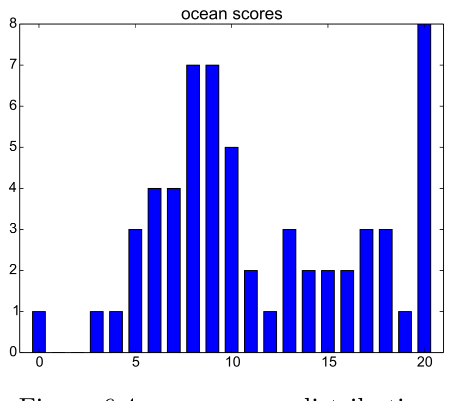


Figure 6.4: ocean score distribution

The forest follows a nice distribution that you'd expect, apart from one remarkable outlier score of 20 from one of the children who played the game. Unfortunately the game itself didn't ask for the age of the player, as seeing what age groups performed best would've also been interesting. However the highest scores on both levels were achieved by children, which is nice to see as it shows that background knowledge gained from age wasn't necessary to play the game.

The ocean's scores are more sporadic, as not as much testing went into balancing its parameters due to the level being a relatively late addition close to the festival, and it was noticeably easier to get a better score. The large spike at the maximum score of 20 is due to multiple players using the tactic of just introducing all the species ascending in order of body mass, which worked to just about sustain all of them just before the 2 minute timer ran out.

Here's the order of introduction of species for that score of 20 in the forest:

```
+ Grass 22962.99
+ Acorns 22968.66
+ Sunflower 22974.99
+ Strawberries 22980.59
+ Hare 22987.16
+ Blackthorn 22992.27
+ Boar 22999.19
+ Deer 23004.91
+ Grouse 23015.97
+ Snake 23022.52
+ Owl 23028.89
+ Wolf 23035.51
+ Ibex 23040.89
+ Lynx 23046.77
+ Fox 23052.47
+ Bear 23058.86
+ Bison 23065.19
+ Eagle 23070.94
+ Marmot 23076.77
+ Caterpillar 23082.46
```

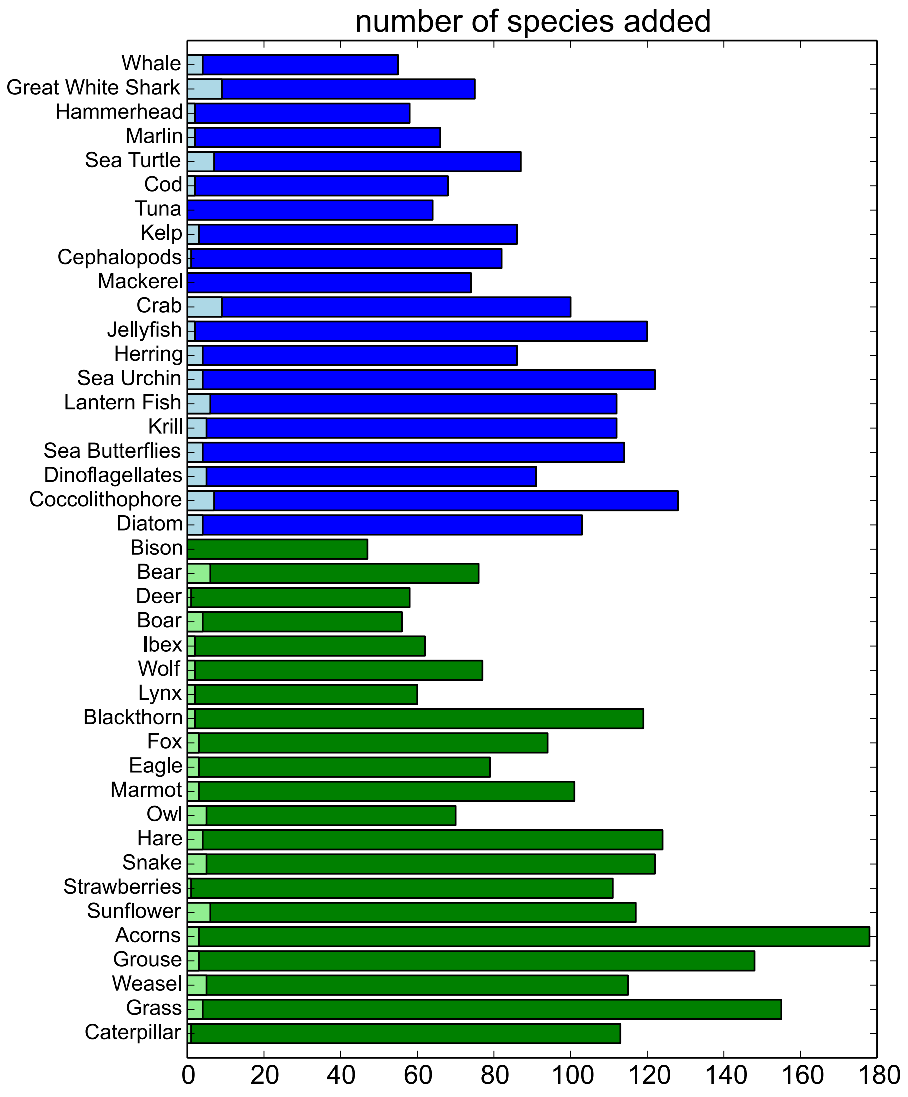


Figure 6.5: The total number of times each species was introduced, with the lighter colour at the base of the bars being the number of times each species was removed. The names are sorted from heaviest to lowest body mass, which is also the order in which they're sorted on the selection wheel in the game; you can see a correlation between weight and frequency of introduction, which is likely caused by this ordering on the wheel, but also due to users building their ecosystems starting with smaller species which are normally lower on the food chain.

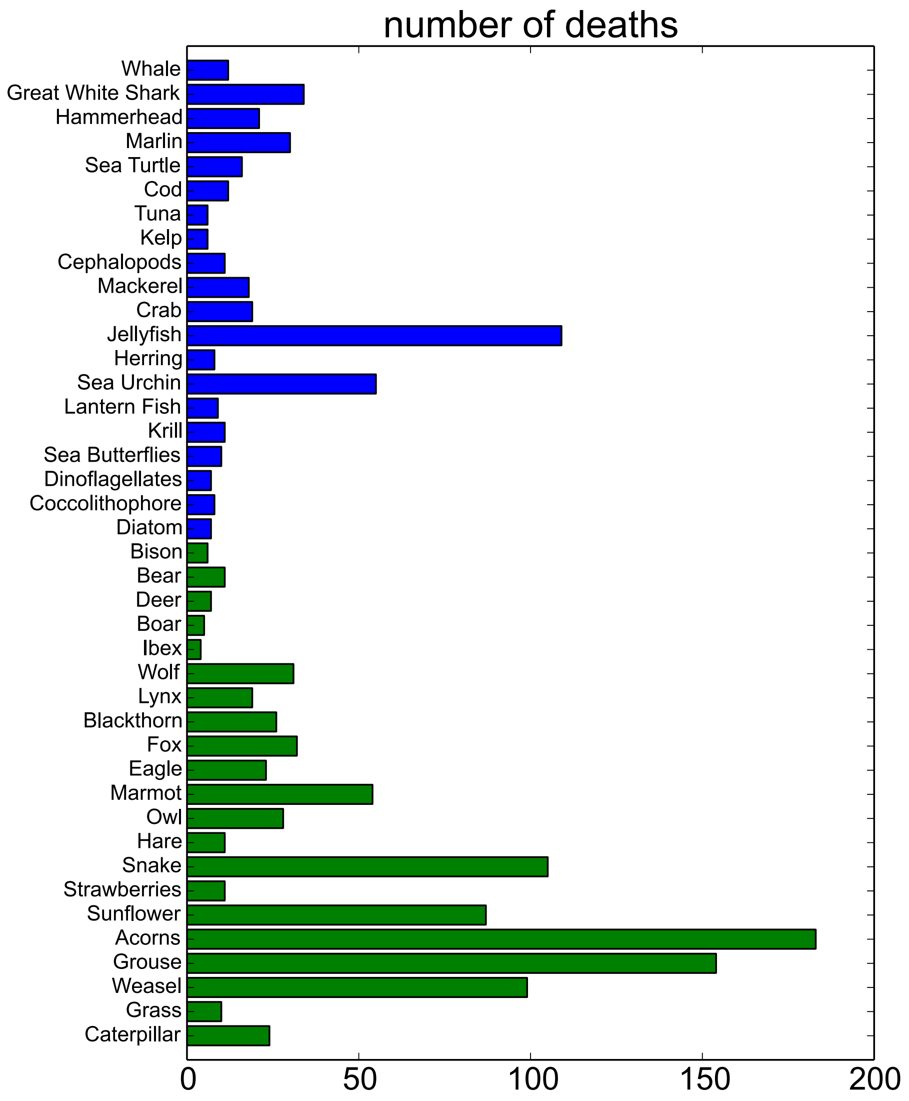


Figure 6.6: The total number of times each species died. The interesting thing here is that instead of a smooth distribution, it contains peaks from certain species that were a lot harder to keep alive than others. Lighter species do lose biomass quicker, and that definitely contributed to the high death rates of some of these smaller species, but I still don't have a good reason for why the Jellyfish was so hard to keep alive in the ocean. This shows it's not trivial to predict how some species will behave due to the complexities of the system, so this kind of data could be of some use for understanding the model better.

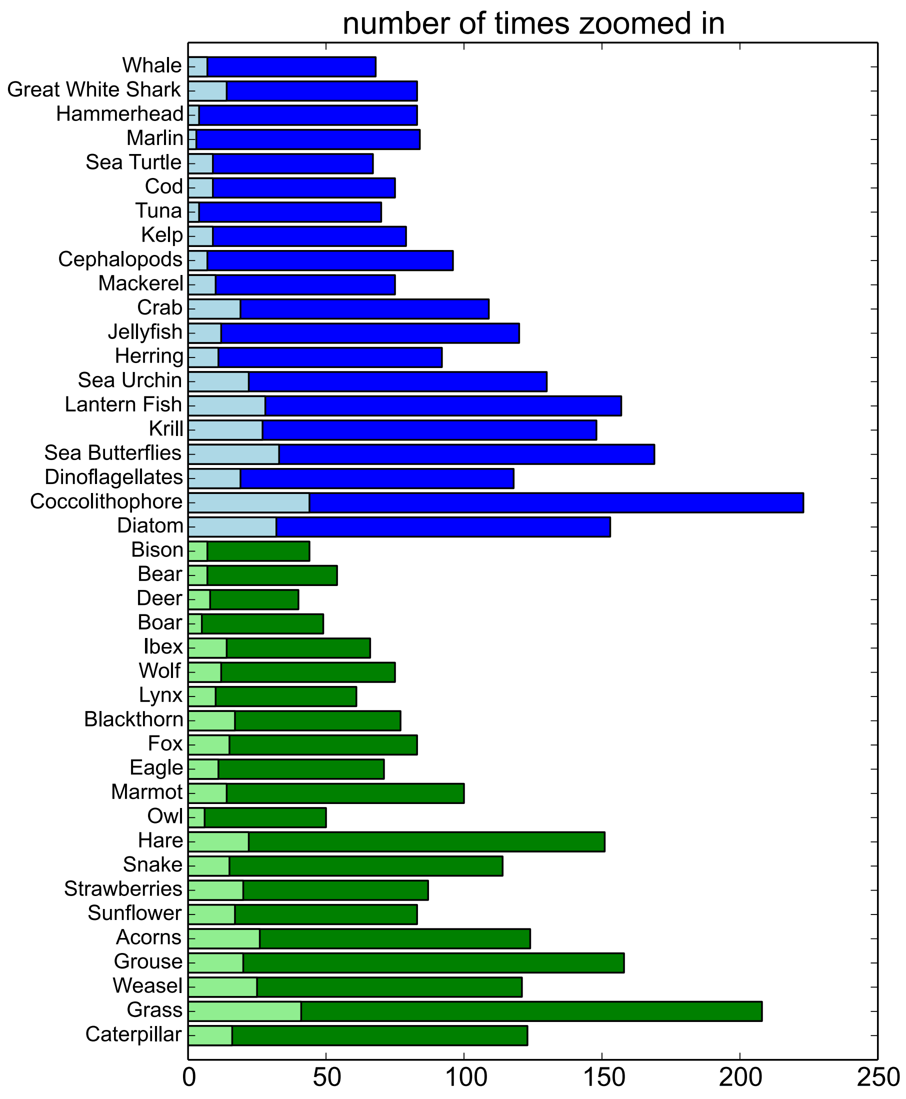


Figure 6.7: The total number of times each species was examined by zooming in from pressing its icon. The lighter colour at the base in here is the amount of times the user zoomed in further to see the fun fact about the species, which was a lot less than expected. This was likely due to the time constraints of the game, as players were more concentrated on playing the game than finding out more about the species themselves, which is good in that the game was engaging, but also a bit sad because of the amount of time we spent creating those extra assets. However you can still see that there were some users who did take advantage of this feature, which is nice to see.

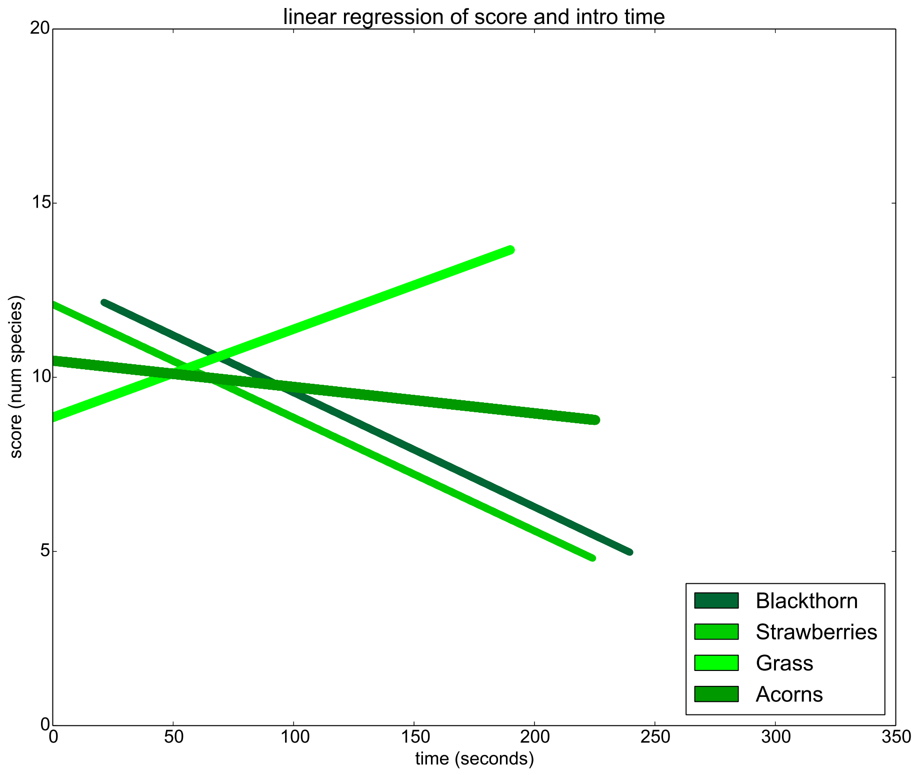


Figure 6.8: A linear regression of introduction time against score for producers in the forest. This graph plots score against the time of introduction inside the challenge modes, and what we expected was for animals at the bottom of the food chain to produce higher scores if they were introduced earlier, as they could provide more food for the larger species earlier on. Because of this, the regressions of the producers are plotted separately from the consumers, as placing them all on one graph got messy, and their behaviours are expected to contrast. The species with more data points are drawn thicker and on top of species that didn't get added as often too, and their colours match up to the icon border colours within the game.

I expected downward slopes for these, and this is how they generally behave except for grass, which I think has an upward slope for two reasons. One, grass was the first species introduced for almost every playthrough, which was likely due to both the tutorial and its low body mass, causing its points to almost all be clustered at 0 seconds. Two, there's one outlier data point for a good score where the user introduced grass very late, which skewed the whole line upwards. It's also a possibility that a straight line doesn't model the distribution very well, and that a higher level polynomial could fit better, which could make sense in that you'd expect a 'bump' at a good time to introduce the animal. However a species' success also depends on the introduction time of all the other species too, which complicates it even further; a linear regression in general isn't a perfect fit for data like this, but it still shows a potentially useful pattern through the somewhat chaotic data.

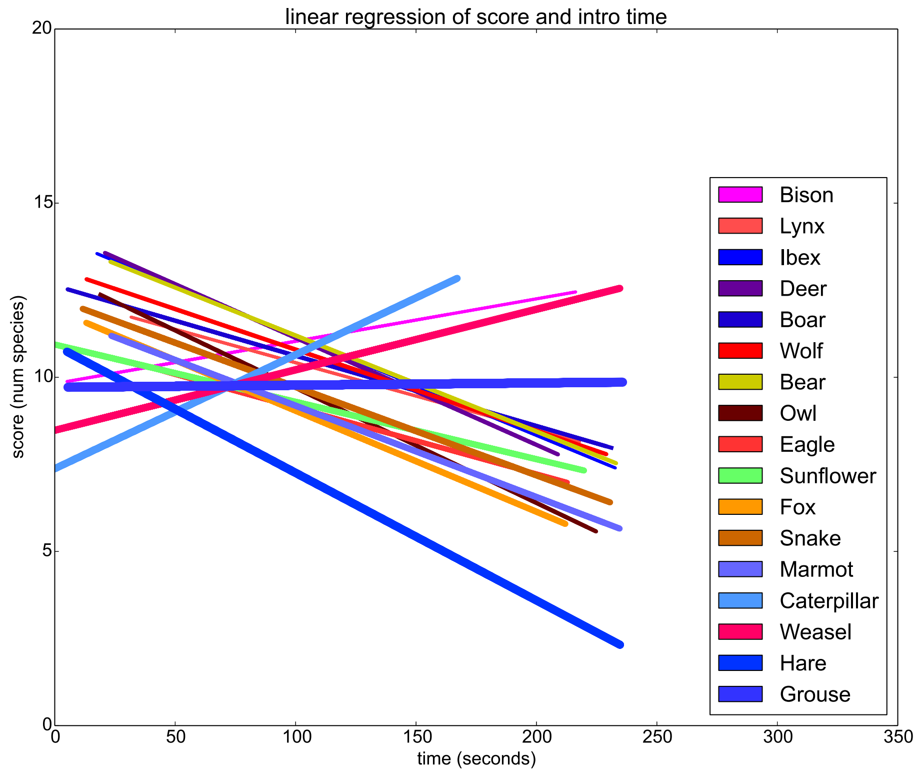


Figure 6.9: Linear regression for consumers in the forest. In this case I then expected species further up the food chain to have upwards slopes, as they should require a more complete ecosystem beneath them before introduction. This turned out to be generally untrue, as almost all of the lines slope downwards. This may have something to do with our scoring system: because it ends at 2 minutes and their score is the number of species at that point, then even if a bunch of species are on the brink of extinction, they still count towards the total number. A solution to this would be to run the simulation through more time afterwards, to see how their system would stabilise. The matching regressions of the ocean level are in the Appendix if you want to take a look at them too.

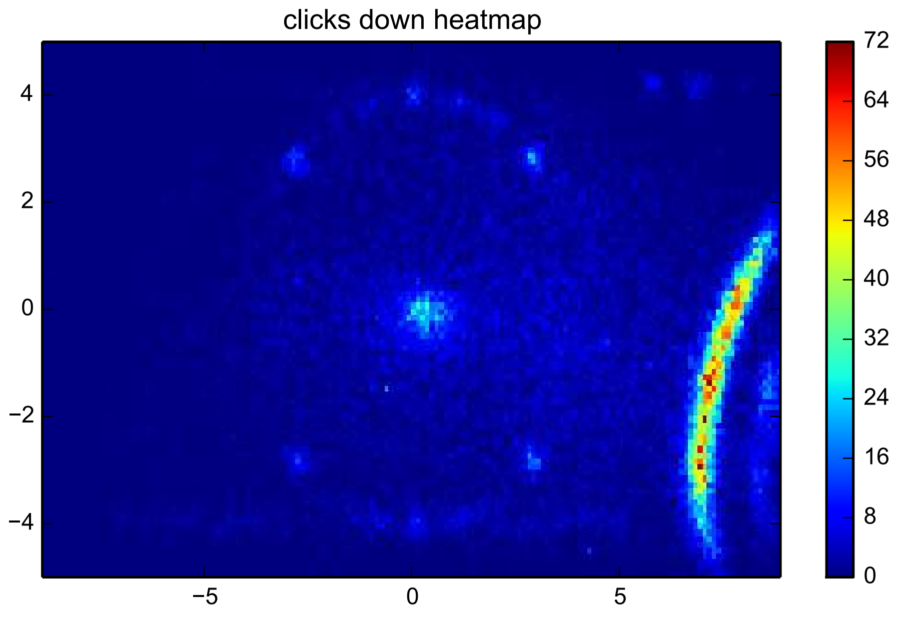


Figure 6.10: Heatmap of all clicks down on the screen throughout the weekend. You can see an affinity to the right hand side of the screen, both because the interface objects were there and that most people are right-handed. There's also quite a lack of clicks towards the bottom of the screen where the tutorial text exists, as users tended to not read it as often as expected.

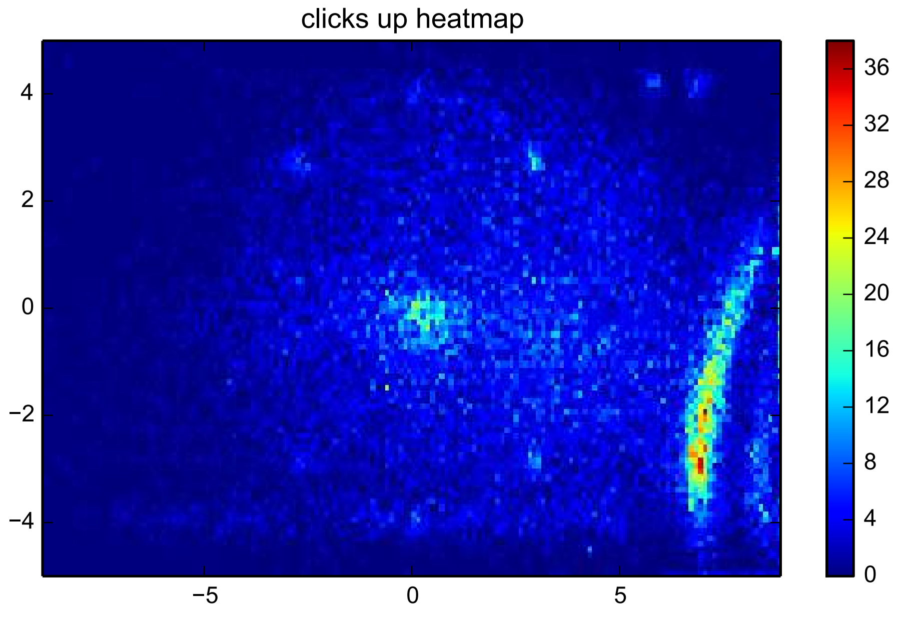


Figure 6.11: Heatmap of all clicks up. You can see that they're more spread out, because users were releasing the dragged species on the up clicks. Again you can see the strong affinity towards the right of the screen.

7 Evaluation

There's a lot to discuss here, so I'll split it up into the separate goals that we had, and discuss how successful the game was at achieving them. I'll then end with some ideas for future work, along with some final conclusions.

7.1 Intuitiveness of the Visualisation

In general, feedback was positive about the game's appearance. A lot of people complimented the aesthetic of the game, and although it's nice to hear that it looks good, the real aim was to convey the characteristics of the system. While a pretty interface does make it easier for the player to become engaged to better absorb information, it's also important to review how effective individual elements were at telling the user their piece of information. In this regard, there were lots of subtle observations I was able to make from watching so many people play the game, and I'll discuss some of them here.

It was often not clear what some aspects of the icons' appearance meant. The fact that producers were square and consumers were circles was a good idea, but not quite intuitive enough to be able to not explicitly tell the user in the tutorial. Future versions would likely make more use of the shape of the icon, such that it becomes a core feature of the visualisation; for example the number of sides on the polygon could indicate the number of interactions, or the trophic level etc.

The size scaling and positioning of the icons was also something that needs work. For size, the scaling function used for the Festival was simply proportional to the log of the species' total biomass, and this wasn't perfectly intuitive. One issue is the extinction thresholds, as heavier species have more biomass per individual and therefore their icons are larger when about to go extinct, which is inconsistent if we want to use this size to indicate how healthy a species is. However the issue with using population to scale size is that heavier species in this case have a lot less individuals per biomass, and you don't want their icons to be small even if they're reasonably healthy. The ideal solution is a problem that needs to be explored further.

For the positioning of icons, a problem was that people tended to drag species right into the middle, which led to a big clump of icons around the middle and isn't good for visualising the food web structure. A better solution would be a combination of letting the user place icons where they want, but to also move obviously strong interactions closer together to give the player a little push towards a nice arrangement. This sort of algorithm is known as force-directed graph drawing [51], and an exploration of this could be the best solution towards spacing correctly.

The icons also filled up the entire screen when close to the maximum number of species was added to the system. This would be solved by scaling the entire view of the system with the number of species that are in there, which would require some changes in the architecture of the code.

Finally, the biggest observation for me was how users interacted with the tutorial. It's impossible for the system to be intuitive enough to not require any guidance, so it's important for the information in the tutorial to be conveyed well. Unfortunately, what we found was that the method of using a text box at the bottom of the screen wasn't actually very effective, and that players often didn't read the text at all, instead working through the tutorial by just moving icons around until they did the right thing for the game to progress.

This is a difficult problem to solve, one which is likely caused by our expectations of modern smartphone interfaces, which don't really ever involve reading any more. A solution could be to make the text more intrusive, forcing them to read it before continuing, which isn't ideal as the whole experience should be more organic. The other solution, which I like more but is harder to do well, to more directly have the player learn by doing. In this case the tutorial would present a number of specific situations where they have to make use of a feature to solve the problem, which would force them to take notice because then it's clearly beneficial to their success to follow the instructions.

Either way, the tutorial definitely needs to go through more extensive testing and iterations before it's as effective as it should be. Thankfully we had people around to help during the festival, so any gaps in information that didn't stick from the tutorial could be filled in with help in person.

All in all, the visualisation was good enough to show the majority of users most of the core concepts of the model. Once users understood the concept of seeing the interactions through zooming in on species, they started to play the game properly, and seeing that 'lightbulb moment' was gratifying. It's making this happen that's difficult, but observing at the festival went a long way towards this goal.

7.2 Effects of including Game Mechanics

In my mind, the two core game mechanics were letting the user introduce species to their ecosystem, and introducing time constraints and scoring metrics to add challenge. The first question to ask is "*Was it fun?*", and I think the answer is a tentative yes. The mechanics were effective at creating player engagement, and some visitors really got into the game and tried to understand the system. However in its current state, it's not something that you'd play more than once, as the game doesn't give the player novel goals to aim for. Extending this duration of engagement would therefore be high on the list of priorities for future work.

In terms of encouraging learning, this concept of including game mechanics was definitely well received. There was a lot of interest in how a more complete version could be used in classrooms, and the competitive element got a good number of the kids wanting to learn how the system worked in order to beat their friends. However this competition also likely contributed to the lack of reading the zoom2 text, which limited tangential learning.

Overall, the mechanics of the game did work towards showing users the consequences and chain reactions of changing ecosystems to the vast majority of players, with some requiring more extra help outside the game than others. If we were to do the festival again now, I'd definitely have some sort of survey to record feedback on these effects more quantitatively, but it is clear at least that there's a noticeable impact on the player.

7.3 Helpfulness for Research

As you can see in the graphs in the Results above, there is a lot of data that can be collected from a game like this, and the incorporation of human intuition into the results makes it different from any tests run purely on a computer. Whether or not this data is useful for helping the research effort is subject to further consideration, but it's clear that there are noticeable patterns to how humans interact with a tool like this using their intuition, and taking advantage of this to home in on potentially useful aspects of the model is definitely a possibility. To go further in this area, close collaboration with researchers in relevant fields would be required to find what exactly the game should focus collecting data for.

7.4 Outreach

The following was written in my interim report:

> *"This ability to see the impacts of changes to the model also lends itself well to raising awareness for real world issues, such as the overhunting of a species, or the negative effects of global warming, leading to systems that are unstable or rendered extinct. The game itself will be able to simulate these problems in the user's model, and the goal will be to fight these undesirable outcomes by making changes to their system, within predetermined constraints to make the game challenging."*

In the end this was a step too far for the time frame, and we didn't manage to get these mechanics done in time. The main reason for this was being unable to implement useable effects of temperature in time, which left us unable to take the global warming angle. However developing the game did make it clear that these extra mechanics would fit in perfectly, and are first on the list of features I'd like to add.

Despite this, we were broadly successful in terms of research outreach. Presenting a project like this at the festival brought a lot of interest in the work being done in both the EEE and Life Sciences departments, and this led to members of the public learning more about the research being done at Imperial. The game was good at drawing a crowd, as I think we were successful in creating a marketable product that was effective at capturing the public's interest, which bodes well for further outreach opportunities in the future.

7.5 Future Work

It's still unclear whether data gathered from the game can contribute to a research effort, but if it can then the game has the potential to crowd source this endeavour. This would be in a similar vein to the game 'fold.it', which has players attempt to use their intuitions to fold protein structures, helping to solve complex geometric problems that computers have trouble with [52]. It's been very successful, and shows that the public is keen on helping research if it's made accessible.

There's also a lot of work to be done in the visualisation of neural networks, which have similarities in their underlying models, especially in terms of the complexities involved. The expansion of the EcoBuilder name into computational neuroscience could be called BrainBuilder, and I really like this shared brand that would tie together the body of work. The concept could really be taken to any lengths, and the interest is there to take advantage of the benefits it could provide - some of which are likely still yet to be discovered.

7.6 Conclusions

Looking at things from the right perspective is hugely important, like how looking in the frequency domain doesn't actually change any information, but presents it in a way that shows unique insights. The visualisation that we're aiming at creating is not as fundamental as something like a Fourier transform, but the idea behind it is the same, and it's an important step towards understanding complex systems like the ecosystems we've examined.

We've seen from our testing that the concept of game mechanics facilitating understanding of the model can work, and also that the public takes a keen interest in the idea. The boxes have been ticked for checking its potential, and it's clear that building upon it in the future would be worthwhile.

8 User Guide

The WebGL version of the game can be played at www.ecobuildergame.org. The entire project can also be run in the Unity Editor using the public git repository at https://gitlab.doc.ic.ac.uk/jxz12/FYP. Simply create a new project in Unity, then clone the repository into the project folder, and Unity will import and sync everything needed to make changes yourself. The version of Unity I used was 5.3.4.

9 References

[1] "History of wolves in yellowstone - wikipedia." https://en.wikipedia.org/wiki/History_of_wolves_in_Yellowstone. [Online: accessed 31/5/2016].

[2] "Unreal engine - wikipedia." https://en.wikipedia.org/wiki/Unreal_Engine. [Online: accessed 1/2/2016].

[3] S. Pawar, "The role of body size variation in community assembly," *Advances in Ecological Research*, vol. 52, pp. 201–248, 2015.

[4] R. Honegger, "Functional aspects of the lichen symbiosis," *Annu. rev. Plant. Physiol. Plant. Mol. Biol.*, vol. 42, pp. 553–578, 1991.

[5] M. Brown, "Superintendent's fy 2011 4th quarterly report," tech. rep., Gulf of the Farallones National Marine Sanctuary, September 2011.

[6] R. Peters, *The Ecological Implications of Body Size*. Cambridge University Press, 1986.

[7] J. H. Brown, J. F. Gillooly, A. P. Allen, V. M. Savage, and G. B. West, "Toward a metabolic theory of ecology," *Ecology*, vol. 85, pp. 1771–1789, 2004.

[8] A. A. Aljetlawi, E. Sparrevik, and K. Leonardsson, "Prey-predator size-dependent functional response: derivation and rescaling to the real world," *J. Anim. Ecol.*, vol. 73, pp. 239–252, 2004.

[9] J. DeLong, "The body-size dependence of mutual interference," *Biol. Lett.*, vol. 10, 2014.

[10] "The leading global game industry software." https://unity3d.com/public-relations. [Online: accessed 1/2/2016].

[11] "Yoyo games showcase." http://www.yoyogames.com/showcase. [Online: accessed 1/2/2016].

[12] "Gamemaker: Studio user manual." http://docs.yoyogames.com/. [Online: accessed 1/2/2016].

[13] N. Landry, "Getting started with 2d game development using gamemaker." http://www.ageofmobility.com/2014/07/11/getting-started-with-2d-game-development-using-gamemaker/, July 2014. [Online: accessed 1/2/2016].

[14] "Unreal engine showcase." https://www.unrealengine.com/showcase. [Online: accessed 1/2/2016].

[15] M. Masters, "Unity, source 2, unreal engine 4, or cryengine - which game engine should i choose?." http://blog.digitaltutors.com/unity-udk-cryengine-game-engine-choose/, 2015. [Online: accessed 1/2/2016].

[16] A. Borisov, "Documentation, unity scripting languages and you." http://blogs.unity3d.com/2014/09/03/documentation-unity-scripting-languages-and-you/, September 2014. [Online: accessed 1/2/2016].

[17] L. Meijer, "Is unity engine written in mono/c#? or c++." http://answers.unity3d.com/answers/9695/view.html, January 2010. [Online: accessed 1/2/2016].

[18] "Mono." http://www.mono-project.com/. [Online: accessed 1/2/2016].

[19] M. Amazonas, "How does unity export to so many platforms?." https://sometimesicode.wordpress.com/2014/02/19/how-does-unity-export-to-so-many-platforms/, February 2014. [Online: accessed 1/2/2016].

[20] M. Amazonas, "How does unity3d scripting work under the hood?." https://sometimesicode.wordpress.com/2014/12/22/how-does-unity-work-under-the-hood/, December 2014. [Online: accessed 1/2/2016].

[21] "Unity manual." http://docs.unity3d.com/. [Online: accessed 1/2/2016].

[22] "Instantiating prefabs at runtime." http://docs.unity3d.com/Manual/InstantiatingPrefabs.html. [Online: accessed 1/2/2016].

[23] "Monobehaviour.fixedupdate()." https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html. [Online: accessed 11/6/2016].

[24] "Monobehaviour.fixedupdate()." http://docs.unity3d.com/Manual/Coroutines.html. [Online: accessed 11/6/2016].

[25] "Benchmarking unity performance in webgl." http://blogs.unity3d.com/2014/10/07/benchmarking-unity-performance-in-webgl/. [Online: accessed 11/6/2016].

[26] "National curriculum in england: science programmes of study." https://www.gov.uk/government/publications/national-curriculum-in-england-science-programmes-of-study/national-curriculum-in-england-science-programmes-of-study. [Online: accessed 1/2/2016].

[27] "Bbc bitesize: Food chains activity." http://www.bbc.co.uk/education/guides/z2m39j6/activity. [Online: accessed 1/2/2016].

[28] "Ecosystems: 11 studyjams! interactive science activities." http://www.scholastic.com/teachers/activity/ecosystems-11-studyjams-interactive-science-activities. [Online: accessed 1/2/2016].

[29] "Pbs kids: Jungle jeopardy." http://pbskids.org/plumlanding/games/ecosystem/jungle_jeopardy.html. [Online: accessed 1/2/2016].

[30] "Kickstarter: Eco - global survival game." https://www.kickstarter.com/projects/1037798999/eco-global-survival-game. [Online: accessed 1/2/2016].

[31] "Wikipedia: Tangential learning." https://en.wikipedia.org/wiki/Learning#Tangential_learning. [Online: accessed 1/2/2016].

[32] B. Burke, "Gartner redefines gamification." http://blogs.gartner.com/brian_burke/2014/04/04/gartner-redefines-gamification/, April 2014. [Online: accessed 1/2/2016].

[33] H. S. Juho Hamari, Jonna Koivisto, "Does gamification work? - a literature review of empirical studies on gamification," *Hawaii International Conference on System Science*, vol. 47, 2014.

[34] H. S. Juho Hamari, Jonna Koivisto, "Challenging games help students learn: An empirical study on engagement, flow and immersion in game-based learning," *Computers in Human Behaviour*, vol. 54, 2015.

[35] DiGRA and University of Vancouver, *Fundamental components of the gameplay experience: analysing immersion*, Changing Views - Worlds in Play, (Vancouver), 2005.

[36] "paint.net." https://en.wikipedia.org/wiki/Nintendogs. [Online: accessed 13/6/2016].

[37] D. Tilman and P. M. Kareiva, *Spatial ecology: the role of space in population dynamics and interspecific interactions*, vol. 30. Princeton University Press, 1997.

[38] "Agile software development." https://en.wikipedia.org/wiki/Agile_software_development. [Online: accessed 14/6/2016].

[39] "scipy.integrate.odeint." http://docs.scipy.org/doc/scipy-0.16.0/reference/generated/scipy.integrate.odeint.html. [Online: accessed 14/6/2016].

[40] "Open solving library for odes." http://research.microsoft.com/en-us/projects/oslo/. [Online: accessed 14/6/2016].

[41] R. I. Stewart, M. Dossena, D. A. Bohan, E. Jeppesen, R. L. Kordas, M. E. Ledger, M. Meerhoff, B. Moss, C. Mulder, J. B. Shurin, *et al.*, "Mesocosm experiments as a tool for ecological climate-change research," *Adv. Ecol. Res*, vol. 48, pp. 71–181, 2013.

[42] R. Stanton, "What made super mario 64 so special?." http://www.eurogamer.net/articles/2015-04-12-what-made-super-mario-64-so-special, April 2015. [Online: accessed 2/6/2016].

[43] "Agar.io." www.agar.io. [Online: accessed 15/6/2016].

[44] "Wikipedia: Runge-kutta methods." https://en.wikipedia.org/wiki/Runge-Kutta_methods. [Online: accessed 1/2/2016].

[45] "Floating-point types table (c# reference)." https://msdn.microsoft.com/en-us/library/9ahet949.aspx. [Online: accessed 8/6/2016].

[46] "2d roguelike tutorial." https://unity3d.com/learn/tutorials/projects/2d-roguelike-tutorial. [Online: accessed 15/6/2016].

[47] "Unity events." https://unity3d.com/learn/tutorials/topics/scripting/events. [Online: accessed 15/6/2016].

[48] ".net queue class." https://msdn.microsoft.com/en-us/library/system.collections.queue(v=vs.110).aspx. [Online: accessed 15/6/2016].

[49] "paint.net." http://www.getpaint.net/index.html. [Online: accessed 13/6/2016].

[50] "Imperial festival 2016 application." https://imperial.eu.qualtrics.com/jfe/form/SV_7VfQBkP2qWMzyvP. [Online: accessed 12/6/2016].

[51] "Force-directed graph drawing." https://en.wikipedia.org/wiki/Force-directed_graph_drawing. [Online: accessed 13/6/2016].

[52] F. Khatib, S. Cooper, M. D. Tyka, K. Xu, I. Makedon, Z. Popović, D. Baker, and F. Players, "Algorithm discovery by protein folding game players," *Proceedings of the National Academy of Sciences*, vol. 108, no. 47, pp. 18949–18953, 2011.

10 Appendix

```
public interface IEcosystem {
    void AddSpecies(string name, float selfInteraction, float birthRate);
    void AddSpecies(string name, float m_i, bool isBasal);
    void RemoveSpecies(string name);

    void AddInteraction(string predator, string prey, float interactionRate, float conversionEfficiency);
    void AddInteraction(string predator, string prey);

    void Integrate(float timestep);

    float GetPopulation(string name);
    float GetBirthRate(string name);
    void SetPopulation(string name, float population);

    Dictionary<string, float> GetTransferredAmounts(string name);
    Dictionary<string, float> GetInteractionRates(string name);
    Dictionary<string, float> GetAllPopulations();
}
```

Figure 10.1: Interface for the Simulation

```
public partial class GameManager : MonoBehaviour {
    IEcosystem foodWeb;
    IconManager iconWeb;
    GUI gui;
    DataLogger dataLogger;
    PuppetMaster puppetMaster;
```

```
public class IconManager : MonoBehaviour {
    Dictionary<string, SpeciesIcon> icons;
```

```
public class GUI : MonoBehaviour {
    Pause pause;
    FastForward speed;
    Menu menu;
    Score score;
    Graph graph;
    Charger charger;
    Timer timer;
    Background background;
    GameObject zoom2effector, opacity;
```

Figure 10.2: Top lines of some Classes to show the Hierarchy

uncompressed:

```
Textures      749.7 mb  99.3%
Meshes        0.0 kb  0.0%
Animations    19.8 kb  0.0%
Sounds        0.0 kb  0.0%
Shaders       20.2 kb  0.0%
Other Assets  456.9 kb  0.1%
Levels        19.1 kb  0.0%
Scripts       539.7 kb  0.1%
Included DLLs 3.9 mb  0.5%
File headers  23.8 kb  0.0%
Complete size 754.7 mb  100.0%
```

compressed:

```
Textures      103.0 mb  95.6%
Meshes        0.0 kb  0.0%
Animations    19.8 kb  0.0%
Sounds        0.0 kb  0.0%
Shaders       20.2 kb  0.0%
Other Assets  177.3 kb  0.2%
Levels        19.1 kb  0.0%
Scripts       539.7 kb  0.5%
Included DLLs 3.9 mb  3.7%
File headers  23.8 kb  0.0%
Complete size 107.7 mb  100.0%
```

Figure 10.3: logs from Unity building the game, to illustrate the importance of compression

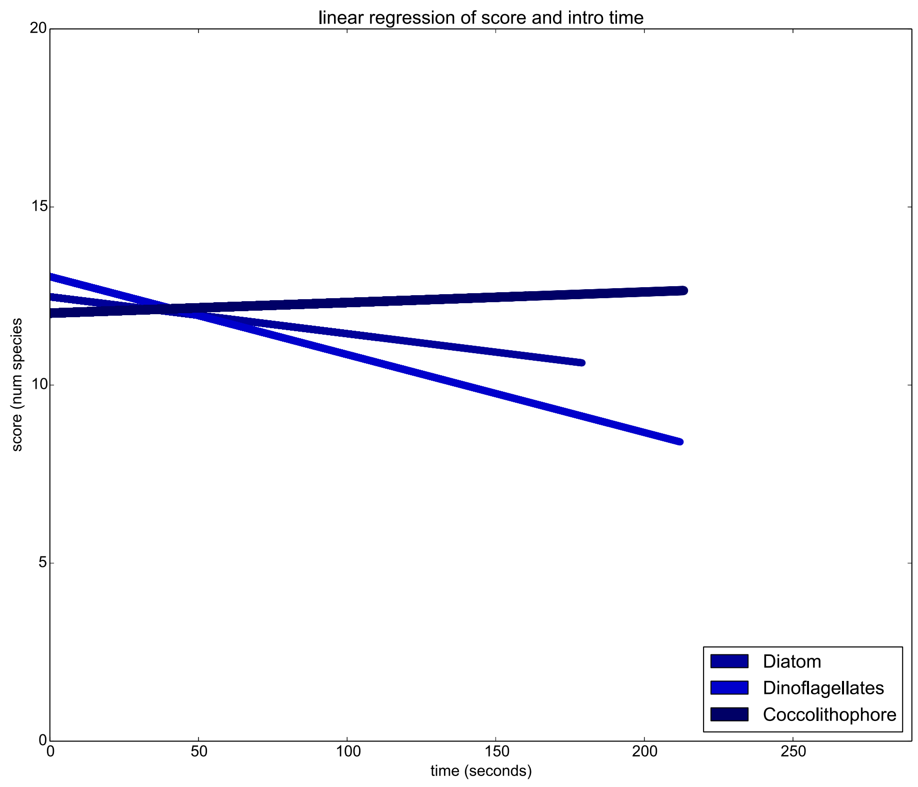


Figure 10.4: Linear Regression of Basal Species in the Ocean Level

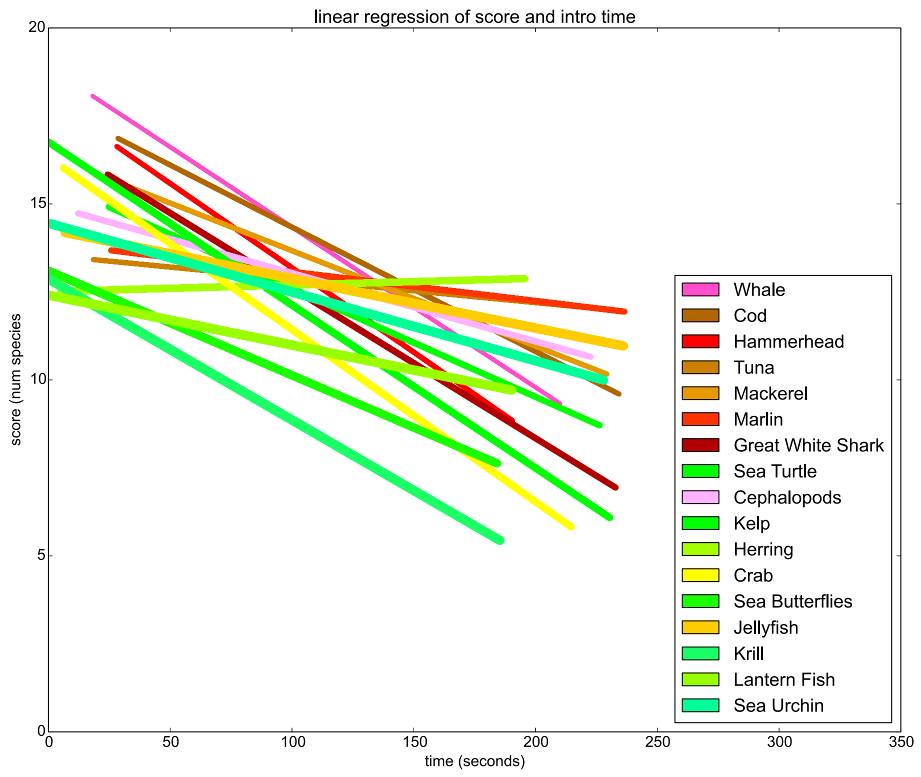
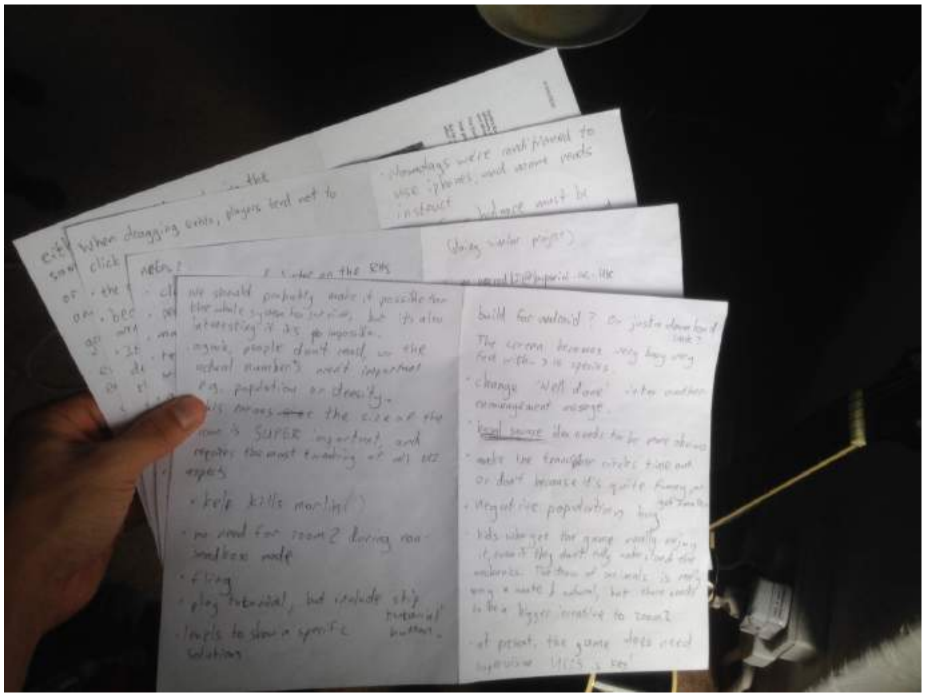


Figure 10.5: Linear Regression of Consumers in the Ocean Level



When dragging [illegible], players tend not to [illegible]

- Nowadays we're conditioned to use iPhones, and [illegible] instruct

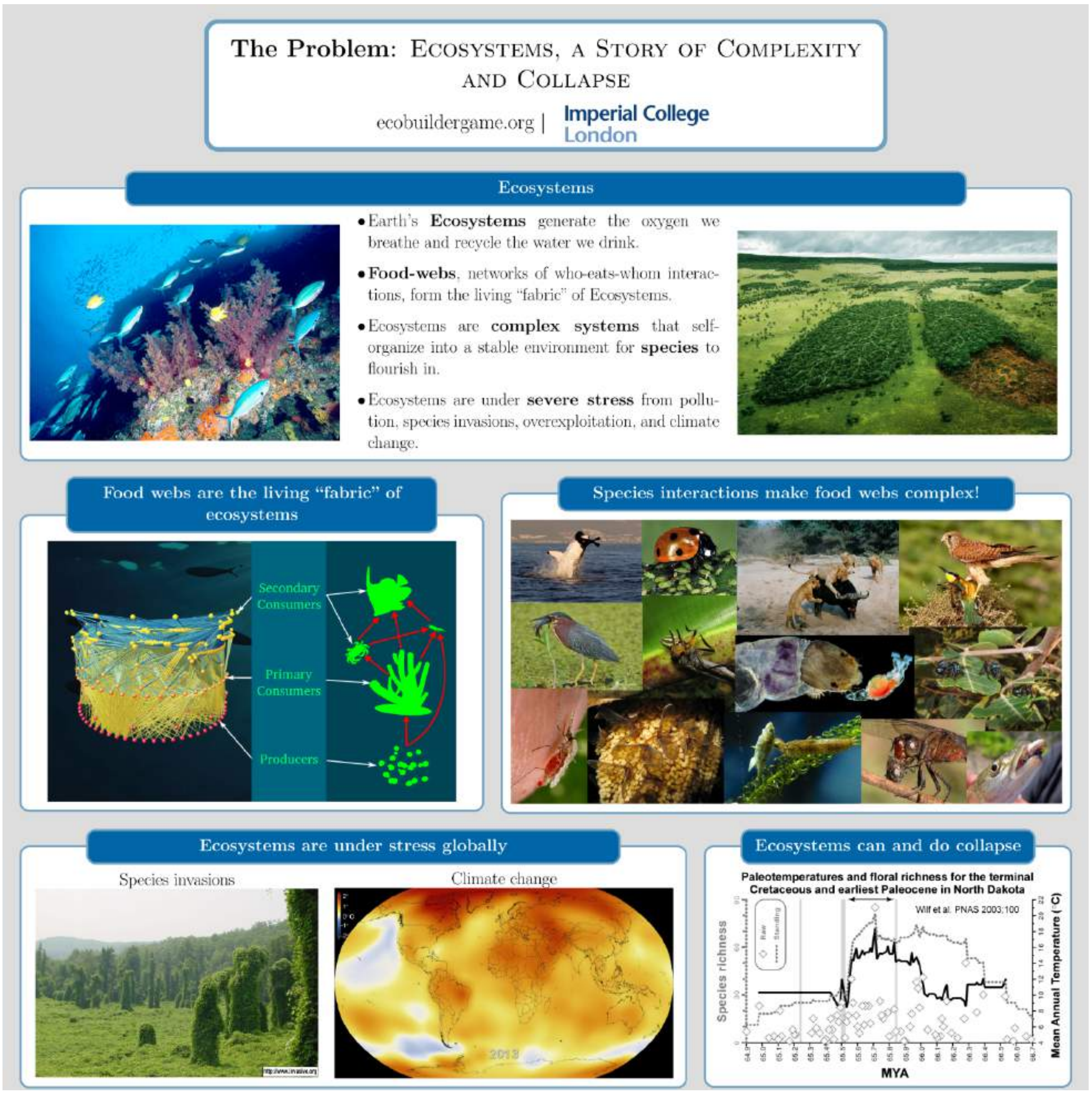
(during water project)

we should probably make it possible for the whole system to survive, but it's also interesting if it's ~~po~~ impossible.
- again, people don't really use the actual numbers/aren't important e.g. population or density.
- [illegible] ~~[illegible]~~ the size of the [illegible] is SUPER important, and requires the most tweaking of all the aspects
- kelp kills marlin(?)
- no need for zoom? during non-sandbox mode
- fling
- play tutorial, but include skip tutorial button.
- levels to show a specific solution

build for android? Or just a download link?

The screen becomes very busy very fast with > 10 species.
- change 'Well done' into another encouragement message.
- [illegible] [illegible] [illegible] needs to be more obvious
- make the [illegible] circles time out or don't because it's quite funny [illegible] [illegible]
- Negative populations bug
- kids who get the game really enjoy it, even if they don't fully understand the [illegible]. The [illegible] of animals is really [illegible] [illegible], but there could be a bigger incentive to zoom?
- at present, the game does need supervision [illegible]

Figure 10.6: Notes made from Observation during Festival



The Problem: Ecosystems, a Story of Complexity and Collapse

ecobuildergame.org | Imperial College London

Ecosystems

- Earth's **Ecosystems** generate the oxygen we breathe and recycle the water we drink.
- **Food-webs**, networks of who-eats-whom interactions, form the living "fabric" of Ecosystems.
- Ecosystems are **complex systems** that self-organize into a stable environment for **species** to flourish in.
- Ecosystems are under **severe stress** from pollution, species invasions, overexploitation, and climate change.

Food webs are the living "fabric" of ecosystems

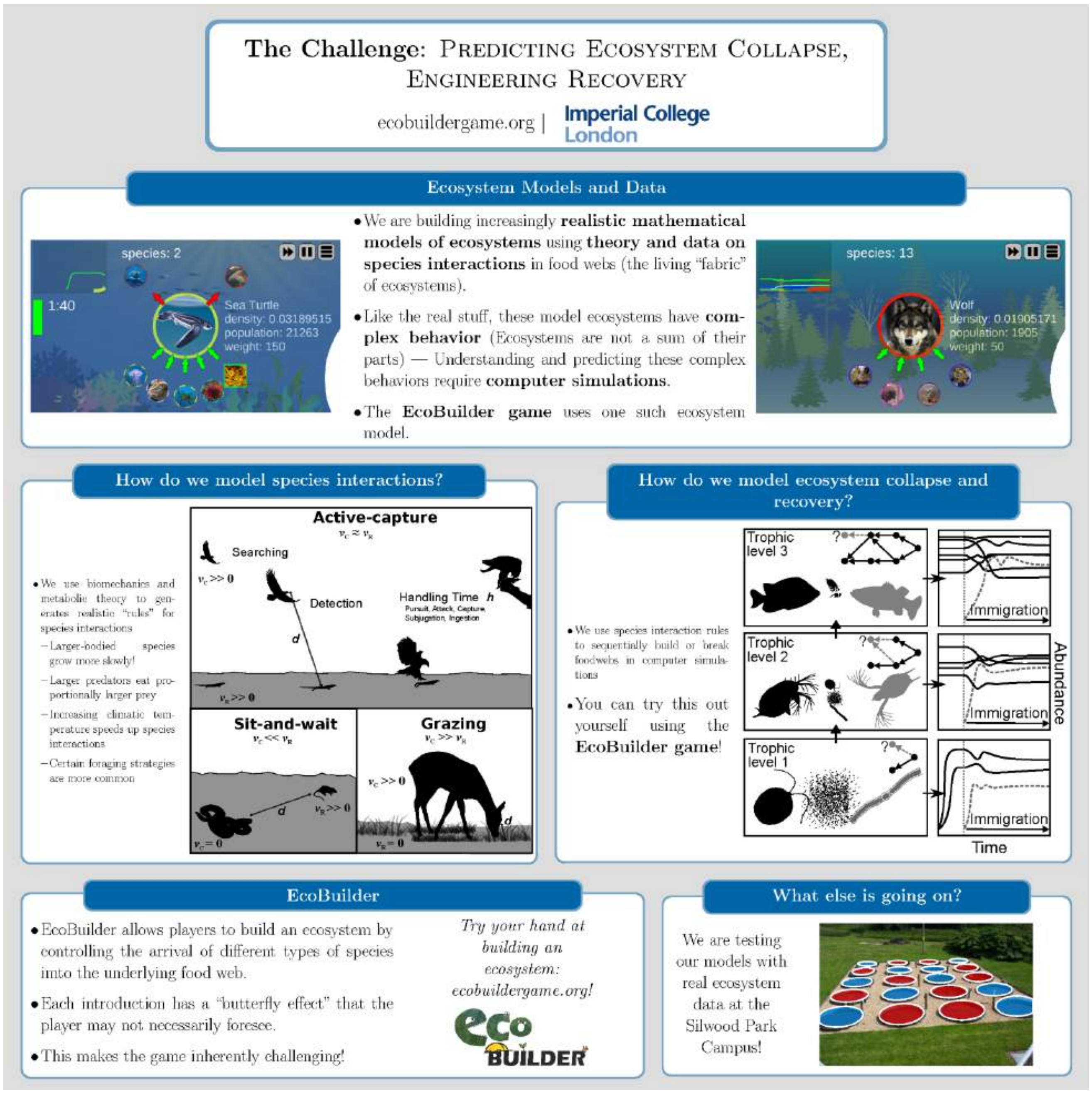
Species interactions make food webs complex!

Ecosystems are under stress globally

Ecosystems can and do collapse

Paleotemperatures and floral richness for the terminal Cretaceous and earliest Paleocene in North Dakota
Wilf et al. PNAS 2003;100
Raw
Standing
Species richness
Mean Annual Temperature (°C)
MYA

Figure 10.7: Poster 1



The Challenge: Predicting Ecosystem Collapse, Engineering Recovery

ecobuildergame.org | Imperial College London

Ecosystem Models and Data

- We are building increasingly **realistic mathematical models of ecosystems** using **theory and data on species interactions** in food webs (the living "fabric" of ecosystems).
- Like the real stuff, these model ecosystems have **complex behavior** (Ecosystems are not a sum of their parts) — Understanding and predicting these complex behaviors require **computer simulations**.
- The **EcoBuilder game** uses one such ecosystem model.

How do we model species interactions?

- We use biomechanics and metabolic theory to generate realistic "rules" for species interactions
  - Larger-bodied species grow more slowly!
  - Larger predators eat proportionally larger prey
  - Increasing climatic temperature speeds up species interactions
  - Certain foraging strategies are more common

How do we model ecosystem collapse and recovery?

- We use species interaction rules to sequentially build or break foodwebs in computer simulations
- You can try this out yourself using the **EcoBuilder game**!

EcoBuilder

- EcoBuilder allows players to build an ecosystem by controlling the arrival of different types of species into the underlying food web.
- Each introduction has a "butterfly effect" that the player may not necessarily foresee.
- This makes the game inherently challenging!

Try your hand at building an ecosystem: ecobuildergame.org!

eco BUILDER

What else is going on?

We are testing our models with real ecosystem data at the Silwood Park Campus!

Figure 10.8: Poster 2