Universität Bremen
Faculty 3: Mathematics and Computer Science

Bachelor's Thesis
Computer Science

Design and Implementation of a Coral Reef Simulation with Focus on Scientific Value and Usability

at
Department of Computergraphics and Virtual Reality
in association with
ZMT Bremen

Fabian Schneekloth
<fabs@uni.bremen.de>
Matriculation No. 4015377
October 2019

Primary examiner:	Prof. Dr.-Ing. Gabriel Zachmann
Secondary examiner:	PD Dr. Hauke Reuter
Supervisor:	M.Sc. Philipp Dittmann

Abstract

This bachelor's thesis covers the re-implementation of the coral reef simulation, called Siccom, implemented by Andreas Kubicek, from the ZMT Bremen. The focus will be on scientific value and usability. This thesis will be written at the University of Bremen, Germany in cooperation with the ZMT Bremen. Siccom simulates individual corals and algae as well as the interactions between individuals, in a coral reef. It also includes destructive events. However the implementation was never designed to be expandable or communicate with other software and therefore limited in its possible uses, in for example the VR CoralReef. The VR CoralReef is an educational software that visualizes the data from Siccom in a 3D environment, implemented by student projects at the University Bremen and also in association with the ZMT Bremen. The goal is to create a simulation that can be easily used and expanded i.e. by future student projects, especially for the VR CoralReef educational software, as well as scientist whose field of expertise will most likely be focused in biology, rather than computer science. While the original Simulation was written in Java, this version will be written in C++ 11. Since a high performance is of significant value in both use cases, the implementation will be optimized to a certain degree. However since the scientific Value and the usability are the main focus of this implementation, the optimization will be limited to 'simple' aspects, such as CPU multi-threading, optimizing algorithms and using data structures to reduce unnecessary calculations. So this optimization won't include high performance elements such as assembler programming or disabling certain C++-features (i.e. exceptions). Also elements/algorithms won't be substituted for alternatives if that would impact the realism of the simulation in a significant way. At some points even a lower performance solution might be chosen if that supports the expandability and/or usability in a relevant manner. To reach the goals, the implementation uses the Domain Driven Design architecture style and a grid data structure, as well as refactoring of certain class structures. The results are a software that improves on all aspects of the original and enables an easier, further development of new features.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendete Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Bremen, den

Fabian Schneekloth

Contents

Chapter 1

Introduction

This chapter will elaborate the motivation and goal of this thesis. Additionally in Chapter 1.3 will describe the structure of the following chapters.

1.1 Motivation

The first Version of Siccom was published in 2012, in a paper by A. Kubicek, C. Muhando and H.Reuter from the ZMT Bremen.[1] The ZMT, or Leibniz centre for tropical marine research is a german research institute for the study of tropical marine ecosystem. The goal was to provide a coral reef simulation for further 'understanding of coral reef resilience and the identification of key processes, drivers and respective thresholds, responsible for changes in local situations'. These processes can include the shifting from one dominant species to another or massive coral die offs. Siccom was developed as an individual based model, where higher system properties arise through competition and self-organization.

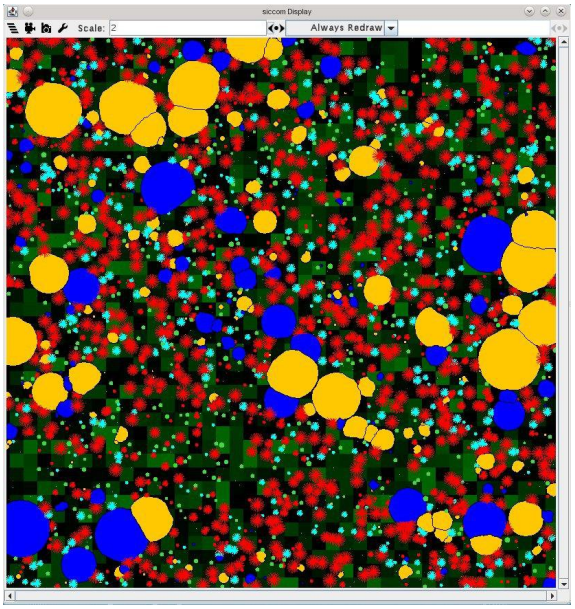

Figure 1.1: The visualization of a running Simulation of the original Java implementation of Siccom. The squares represent turf algae, simulated as cell with the brightness representing the percentile coverage. The circles represent the simulated massive corals, while the star shapes represent branching corals.[1]

However since the original simulation was visualization wise limited to functionality and therefore not that suited for usage in educational cases or display on public conferences and fairs, the idea for a external 3D visualization of the data came up. Figure 1.1 shows a snapshot visualization of a simulated coral reef in Siccom, the circle and star shapes represent the different corals, while the squares represent the turf algae, simulated as cells with a percentile coverage. This idea was then first realized in 2015 by the student project 'VR CoralReef'[2], in cooperation between the CGVRR Department[3] of the University of Bremen and the ZMT Bremen[4]. Figure 1.2 shows the visualization in Ogre3D, from the finished project.

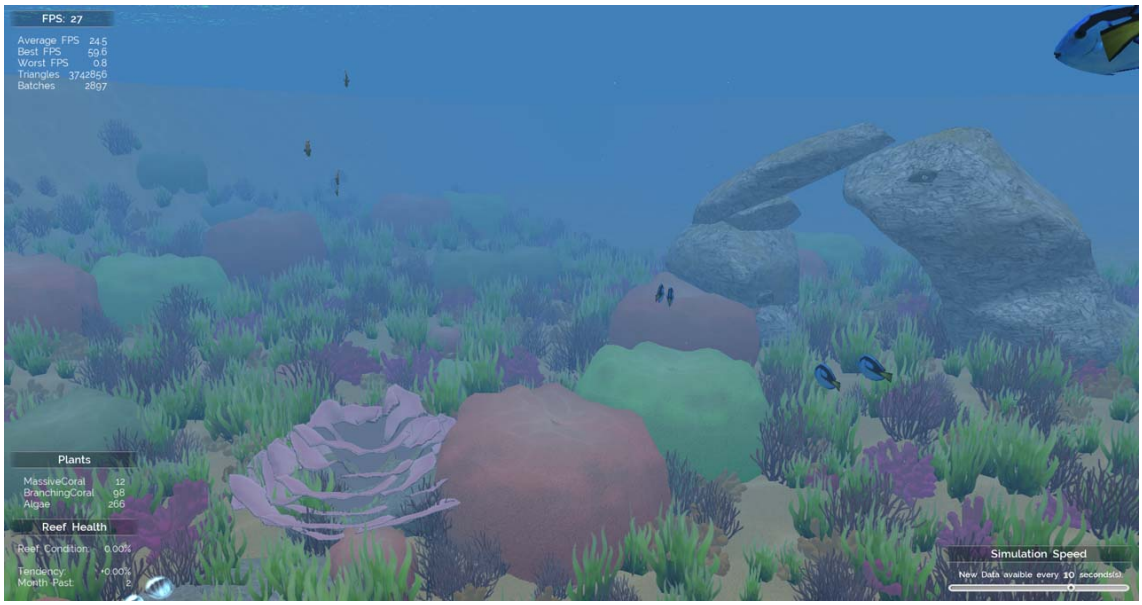

Figure 1.2: This image shows the first iteration of the VR CoralReef, implemented in the Ogre3D engine. [2]

[5]

Since then the project has gone through multiple iterations, including a conversion from Ogre3D[2], to Unreal Engine[5], see Figure 1.3, as well as an Re-Implementation of Siccom. The author of this thesis participated at least partly in both of the mentioned iterations.

So why is a reimplementation necessary, especially if there has already been one?

For that let's first take a look to the original Java implementation. The following observations are purely an objective take on the current state of the software itself. (By no means is this supposed to devalue the simulation or be a critique on its author(s).)

When taking a look at the source code, it becomes apparent that the simulation started quite a bit simpler than it is now and was then expanded and modified bit by bit to accommodate new features, without it ever being designed for that. As a result we see redundant code, for example in the Branching and Massive Coral classes, which are two distinct classes despite them being logically mostly the same.

[1] https://sourceforge.net/projects/siccom/
[2] https://cgvr.cs.uni-bremen.de/teaching/studentprojects/vrcoralreef
[3] https://cgvr.cs.uni-bremen.de/
[4] https://www.leibniz-zmt.de/de/
[5] https://cgvr.cs.uni-bremen.de/teaching/studentprojects/vrcoralreef2/

Figure 1.3: The second iteration of the VR CoralReef, implemented in the Unreal Engine 4, as a student master's project.[5]

In addition the coral classes are divided into the (Branching-/Massive-)Coral and (Branching-/Massive-)Group classes, which might be understandable from a biology standpoint, but from a technical it's quite confusing. Taking a closer look, it seems the coral classes represents an individual coral while the group classes contains the information and logic relevant for the whole species. Other than that we find that the disturbance event is just one of two hard coded functions and all in all there doesn't seem to be a clear structure between the classes and their role in the simulation, except the distinction between GUI and Simulation.
For the VR CoralReef however one of the biggest issues was the network communicator, which was added by Anthea Sander for her master thesis[2], which itself was only a means to move the coral information from the Java simulation to the C++ program. Therefore it was quite limited, in that it could only send coral and alga data to the visualization, but in turn couldn't interpret data, except for temperature changes. In addition the data could only be sent in big chunks all at once and the simulation couldn't be properly paused, only the sending of data, which however caused even more data to accumulate when pausing.

Because of these issues, especially the communication, during the VR CoralReef 2 student's project it was decided that Siccom should be reimplemented for better use with the VR CoralReef software, preferably in C++ for usage with the Unreal Engine. This task was accomplished by Tobias Brandt and Stefan Heitmann.
The new version resolved some of the issues, for example due to the C++ nature it was now possible to directly access and process the data instead of using a network communication. However due to the very limited time frame of the project, they tried to keep actual changes to a minimum, as to avoid adding possible new sources of errors. Therefore there weren't many structural changes or options for easily adding potential future features. As a substitute for the Agent-Modeling Framework MASON[3], which was used by the Java Simulation, they used the C++

Agent-Framework FLAME[4].
Even though the implementation achieved its goal, it wasn't quite ideal and the development process during the project seemed to often be slowed down due to the, in my opinion, hard to follow structure of the code base. And while the FLAME framework already supports parallelization, the reimplementation was reported to be perceived slower than the Java version.[5] This might be due to the Message Board structure used for Agent-communication by FLAME, compared to the geometric approach used by MASON, a design choice that can be argued is not really suitable for Agent based modeling, at least not in this specific case. This this view will be elaborated in the next chapter.

So none of the current versions allow truly easy understanding and expansion of the code base, while being usable for 3D visualization and scientific Simulation at the same time. Both factors can be quite important as the typical developers for the simulation will mostly consists of students with a limited time frame and scientist with a focus other than computer science.
And while currently a professional company, that may not be limited as much by time and programming knowledge, develops a more specialized educational Version of the VR CoralReef, the improvements will still be helpful and might even make new options available.
This thesis's approach tries to achieve both factors with various elements, one being generalization of entities (such as Corals and Algae). As a result of the generalization it should even be quite easy to adapt the simulation into different ecological fields, aside from coral reefs. The approach of creating a simulation for science as well as educational usage, gains also the benefit that adjustments and new features can be easily exchanged between both fields.

1.2 Goals

The goal of this thesis's implementation is to deliver the same basic functionality like the original Java implementation of Siccom, while making it easier to interact with a running simulation and to add new functionality, without sacrificing performance. The original Java version of Siccom will be used as reference standard. Here are the key functionalities of Siccom and how this implementation will approach them:

- **Individual Agents:** Each coral and alga is simulated as a single autonomous object, called agent. The agents compete with each other over space and can be killed through competition. Agents are defined through a number of fixed variables.
  This feature will be implemented mostly the same, but be expanded by generalizing the management of agents. Agents will be generically managed as Entities, for an easier addition of new types of agents, aside of corals and algae.

- **Disturbance Events:** The simulated coral reef can be suspect to destructive events, like an anchor running through or dynamite fishing, which cause corals and algae in the affected area to be destroyed. Such an event is called Disturbance.
  The disturbance functionality will also be expanded by enabling them to be

defined through a fixed number of values. This makes it possible for disturbances to be defined in a generic manner, similar to agents and also to create disturbances based on real world events.

- **Interaction:** After a simulation run is started the original Siccom has no possibilities of further interaction. In the VR CoralReef bachelor's project a network communicator was implemented, that enabled modification of the temperature and requesting data about current corals and algae.
  The goal is to expand this interaction possibilities by also enabling to manually kill or spawn agents and trigger disturbances.

- **Performance:** For proper usage with both, the VR CoralReef and scientific purposes, the simulation needs to be run in at least real time, which it currently does.
  The goal for the new implementation is to at least achieve the same performance, while trying to improve on it, using parallelization.

- **Turfalgae:** As a supportive element, the simulation also includes turfalgae that are simulated as cells, covering the ground. Currently these cells only include a coverage value and a simple growing function.
  These cells functionality will not directly be expanded upon, but be implemented as objects, so their functionality may be easily expanded in future works.

- **Temperature:** The temperature in the simulated reef changes based on real world data and can increase over time to accommodate for climate change.
  This functionality won't be changed from the original implementation.

- **Environment:** Aside from supportive elements, as turfalgae and temperature, the agents are placed in a fixed 2D plane, called the world. The two dimensions of the world can be defined independently from each other.
  This element will mostly stay the same, with the addition of a third dimension. The third dimension will however only be added as a technical possibility, the algorithms won't be adjusted. Similar to the turfcells this is just an addition for possible future work.

- **Configuration:** All information needed for a simulation can be stored in non-code files, this includes the definition of agents and the environmental properties.
  In the new implementation the disturbance information will also be included in the configuration files.

- **Data output:** For further processing the state of the simulation can be written into persistent files.
  This feature won't be changed, but will still be present in the new implementation.

- **Expandability:** This aspect was not considered in the original Siccom. The goal is to create an architecture that enables an easy addition of new agent types in to the code base, without creating unnecessary redundancy and dependencies to other parts of the simulation. Similar functionality, such as

the growing behaviour of different coral species will be unified and mostly be configurable through the configuration files.

1.3 Structure of this thesis

The general structure of the thesis, after the introduction, is to first build a knowledge base for the elements used in this thesis, followed by an explanation of how these elements were implemented, first on an architectural level, then on an algorithmic level. After that the thesis will compare the new implementation with the original Java version and evaluate the results. It will then finish with a summary and a report on possible future works.

- **Chapter 2 - State of the Art:** In chapter 2, this thesis will give an overview of the current state of the art, explaining the key aspects of simulations, different simulation types, the Domain Driven Design architecture(DDD) used in this thesis and coral reefs in general.
- **Chapter 3 - Software Architecture:** After a knowledge base has been build in chapter 2, the thesis follows up in chapter 3 with an explanation of the implemented architecture, describing which classes exist, what roles do they take and how they work together. It will start with the architecture in general, then follow up with an example update process and finish with an explanation of the interaction with external sources. chapter 3 will make use of class and sequence graphs to visualize these aspects.
- **Chapter 4 - Algorithms & Improvements:** Following the architecture, is chapter 4 with an explanation of the algorithms used in the implementation. This chapter will explain the key elements of the algorithms used and how they might have changed, compared to the original ones. It will also explain why these algorithms have or haven't changed and how they might be improved in future works.
- **Chapter 5 - Performance Comparison:** Chapter 5 will then deliver a simple comparison between the runtime of the original Java Siccom and the newly implemented version of this thesis. The performance comparison will include the total runtime and a few specific elements, such as average time needed for a coral update. This chapter will not include an interpretation of the performance data.
- **Chapter 6 - Evaluation:** The interpretation of the data will happen in the following chapter 6. This chapter will also include an evaluation of the goals harder to quantify, such as expandability and usability.
- **Chapter 7 - Summary:** After the evaluation, chapter 7 will summarize the thesis: what was the goal? Was the goal achieved? How was it done?
- **Chapter 8 - Future works:** Following the summary, the thesis will finish with a report on possible future works in chapter 8.

Chapter 2

State of the Art

In this chapter several basics are explained, i.e. what are coral reefs in chapter 2.1 and what is Siccom and it's limitations in chapter 2.3. In chapter 2.2, it will also explain basics about computer simulations in general and what common types there are. , The last to sections will deliver an overview of similar software to Siccom, in chapter 2.4 and cover the architecture style, Domain Driven Design, in chapter 2.5.

2.1 Coral Reefs

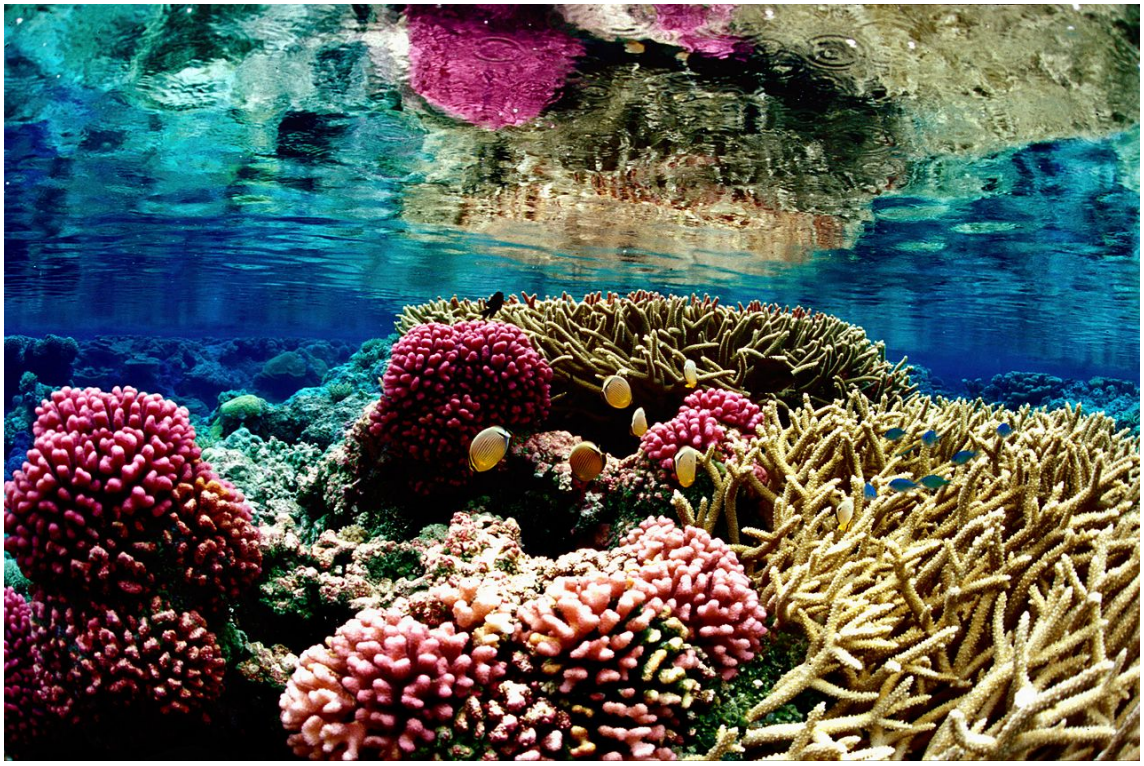

Figure 2.1: A photo of the coral reef ecosystem at Palmyra Atoll National Wildlife Refuge, Palmyra Atoll, central equatorial Pacific Ocean.[1]

[1] `https://commons.wikimedia.org/wiki/File:Coral_reef_at_palmyra.jpg`, last access: 15.10.2019

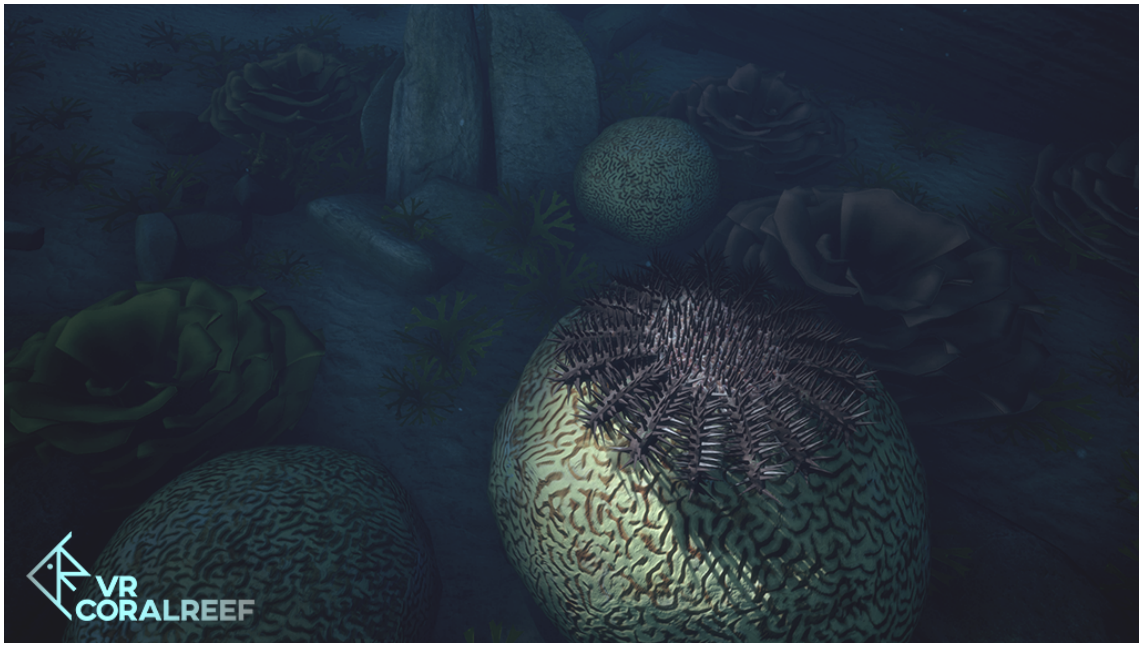

Figure 2.2: This image shows a crown of thorns starfish on a brain coral in the VR CoralReef. Implemented on the Unreal Engine version. Both, the starfish and the coral are only implemented visually, they don't interact with Siccom. Source: `https://cgvr.cs.uni-bremen.de/teaching/studentprojects/vrcoralreef2/index.shtml`

This section will provide a basic overview of what coral reefs are and what organisms are found in them. Some organisms may be described in a simplified manner or only be named, as to be suitable for the context of this thesis.

According to the World Atlas of Coral Reefs[6], coral reefs can simply be defined as the following: 'Coral reefs are shallow marine habits, defined both by a physical structure and by the organisms found on them.' The different types of organisms can be categorized into different phyla (i.e. Coral is a phylum).

The key organisms being the corals. Corals are little, usually cylindrical animals, that use tentacles to capture food from surrounding waters. Most corals have the ability to live in colonies and to build a communal skeleton made out of calcium carbonate, that grows very slowly, in a scale of a few millimeters to centimeters a year.[6] These colonies are what most people usually refer to as corals. This thesis will keep this terminology, since it is easier to understand and the corals in Siccom represent coral colonies. From the World Atlas: 'They require warm, well lit waters and a solid surface on which to settle.'[6]

Coral reefs grow, mostly barely over the vertical axis and therefore usually have a more shallow, two dimensional distribution.

Aside from corals, other well known reef organisms include algae and fish. Algae are a kind of plant, found throughout all reefs and are a critical component of reefs. They do not only add to the reef diversity but also act as a structural component in the building of reefs. Fish are marine animals, with a wide variety of size, shape and diets. (Many fish in reefs eat mostly algae or plankton.)

Another notable organism include starfish, as they are visually implemented into the VR CoralReef, as seen in Figure 2.2.

To highlight the diversity of reefs here is a list of the other inhabitants: Dinoflagellates (Dinoflagellata), Sponges, Cnidarians, Worm-like groups, Crustaceans,

Molluscs, Bryozoans, Echinoderms (include starfish), Tunicates, Reptiles, Seabirds and a very few marine mammals.[6]

Corals have a very narrow requirement for the water temperature, therefore they are very vulnerable to climate change. Other dangers for coral reefs include human induced destruction, i.e. dynamite fishing, tourism, cast anchors and more.

2.2 Computer Simulations

(Computer-)Simulations cover a wide field of sciences, with various goals, most commonly understanding certain relations in the different research fields, verifying theories and maybe the most valuable function: exploring what-if scenarios - all this in a relatively short amount of time. Relatively because, even though the goal is probably always to get the most out of the resources you have, each simulation will have a limit simply due to technical limitations and the sheer amount of data processed. So for example creating a world-simulation with 7.7 billion people as agents, with all body functions, interactions and detailed behavior, would simply be impossible.

The previously mentioned goals are certainly not the only possible ones and while simulations might be intuitively more associated with natural sciences, they certainly aren't limited to it. Just as seen in this[7] paper by Robert Axelrod from 2005, where he explores the role of simulations in social sciences, while also explaining the possibilities of simulations, adding to the list of previously mentioned goals. He also supports the idea of usability and extendibility being the main goals of the programming itself, besides validity.

Even though simulations are common in both natural and social sciences, there are obviously differences between those fields, that also affect the simulations. For example in physics the rules needed for the simulation are pretty well known in the form of formulas for the laws of nature, such as a force being applied to an object. In the social sciences on the other hand there are no clear rules that always apply, at least not as a simple formula, simply because humans are so complex they their behavior can seem random, even if at its core it is not. This distinction is more fluent than it might seem, while in biology the behavior of animals or plants may be more controlled than that of humans, the rules are still not as clear as in physics and therefore can add a certain amount of randomness. And even though in theory the behavior might not be random, if we went down to a cellular or even lower level for our simulations, it would become infeasible to simulate and we would also require even more data for an accurate representation of the processes.

So for a proper simulation we need to identify which aspects we want to simulate and research and which data we actually need. If we research for example migration of an organism and then also simulate the body functions of those individuals, we might not gain any more useful data for our research, since not all body functions will affect the migration, in a more meaningful way than a simplified process. Some aspects, such as the fitness might have an effect, due to different speed in the individuals, but other such as their digestive track, might have no impact at all. These aspects obviously do add a certain amount of information, but simply not the kind, that is helpful for our research and they usually add calculations, which in turn require more processing power.

To achieve these goals there are various types of simulation-styles, most commonly used are Equation-Based-Modeling (EBM), cellular automata (CA) and Agent-Based-Modeling (ABM), all with their own pros and cons, which will be discussed in the following sections. Currently we can also see an Uprising of machine learning and neural networks, which can be used to predict seemingly unpredictable events. This field however is relatively new and requires huge amounts of data, greater than available to most research facilities, therefore I won't discuss these types in this thesis. After the discussion this chapter will finish with a little overview about aspects to consider in scientific modeling, which if not considered can lead to wrong assumptions and unusable results.

2.2.1 Equation Based Models

An equation based model (EBM) is any simulation that describes the simulated aspect using mathematical equations, usually based on empirical data. The advantage of this style is that it, even though it requires a significant amount of data for starting, the data is aggregated and therefore relatively easy to come by. Such data could for example be population amounts or sale numbers. In ecological modeling, ordinary differential equations (ODE) are mostly used. An ODE only takes a single variable as input, enabling a clear identification of cause and effect.
EBM is often considered to be the opposite of ABM, this is not necessarily the case. Even though EBMs usually work on a statistical level, whereas ABM operate on an individual-based approach, they are not mutually exclusive. The equations can not only be derived to a statistical level, but also be used to drive the behavior of individual agents in ABM.

Other styles would need more detailed data about the single individuals providing the data, to simulate their behavior. Another advantage is that this mathematical nature makes the simulation quite easy to implement and due to the large scale abstraction, the calculations can be reduced to an extremely cost-efficient level. The disadvantage however is that this style is quite inflexible and can't accommodate irregular events.

For those reason it's probably more common in commercial uses where the required data is quite easy to collect, even over many years into the past and where irregular events might be considered less important. However as a result of the disadvantages, there are people that argue that even the economy needs Agent-Based-Modeling, to be able to include the effect of financial crises and big changes in the simulations.[8]

2.2.2 Cellular Automata

Cellular automata (CA) or automaton are a simulation type which originated in the 80s as an alternative to statistical modeling, showing that simple rules can result in complex results,[9] simplicity is probably the biggest difference to ABM. This effect can be observed for example in Conway's game of life which can be written in a few lines of code and just 4 simple rules and still create complex self recreating patterns. [2]

[2] https://en.wikipedia.org/wiki/File:Gospers_glider_gun.gif

Figure 2.3: A snapshot of Bill Gosper's Glider Gun in action—a variation of Conway's Game of Life. Both simulations are cellular automata.[2]

A cellular automata consists of a fixed numbers of cells and a set of rules. The cells each have their own status represented by a number of variables, which can just be a single boolean that determines if the cells is alive or dead. The status then updates according to the set of rules, which usually considers the current status, as well as the state of neighboring cells. The updates on each cell are supposed to happen at basically the same time. For this reason the states of the cells need to be copied before applying the rules, so that the update results may not be skewed by the order of the cells. This results in needing double the memory than just the cells themselves. If we apply parallelization to the update process we can still get a usable, realistic result but it requires quite a bit of synchronization, to avoid skewed results due to race conditions.

This is one of the disadvantage of cellular automata, the other are the limitation for the status variables. A cell is of a fixed size, can't move and has a position limited to discrete coordinates, all this doesn't matter it we just take a patch of turf an divide into a grid of cells, but it is not suitable if we want an unknown amount of non-identical individuals, than can vary greatly in those regards.

The big advantage of cellular automata is that they are extremely simple to understand and implement and due to this simplicity often very fast. And despite their limitations they offer a great range of complexity as shown in 'Universality and complexity in cellular automata' by Stephen Wolfram.[10]

While the Siccom-Simulation currently does not use a cellular automata, it could be a useful addition, since different models can of course also be combined, by using the different style for the various components. In the case of Siccom a suitable application could be the turfcells, which are currently just a simple representation of the turfalgae that grows in coral reefs. At this point they are just a bunch of autonomous cells, that grow according to a simple formula, ignoring the state of their neighbors. Other than growing they can simply be overgrown by corals and algae, which causes them to have no coverage and to stop growing.
This is just a supporting aspect of the simulation, but it could be expanded upon by giving the cells a more complex set of rules and adding a sediment type to the state of the cells. With such a change it would become possible to properly consider the geography of the reef. A sediment type could for example represent stone, sand or remains of dead corals, aside from turfalgae.

2.2.3 Agent-Based-Modeling

The approach used in Siccom is called Agent-Based-Modeling (ABM). It is already quite popular and will probably stay that for a while, the reasons were perfectly summarized by Eric Bonabeau:
'The benefits of ABM over other modeling techniques can be captured in three statements: (i) ABM captures emergent phenomena; (ii) ABM provides a natural description of a system; and (iii) ABM is flexible.' [11]

ABM is also known as individual-based-modeling (IBM). An Agent is any kind of simulated object, that acts autonomously on a specific, usually somewhat customized logic. This perfectly represents an individual such as a person, animal or plant, therefore the afore cited 'natural description of a system'. While actual living individuals are perfect for this agent role, they aren't the only possible form, a hurricane or even the ocean could be represented as an agent. The important part is that the agent acts on it's own and outside influences just may modify certain variables which determine the behavior of the agent. For example, a predator or other danger close to an agents chosen food source could reduce the attractiveness of the food source, causing the agent to search for another.

Usually a simulation will have many agents of the same type, in our case corals and algae. For this case the logic will only provide a behavior pattern, which depends on various variables, whose values are tied to the individual agents not the logic. This brings us the other afore mentioned statement 'ABM is flexible'. Since we only operate on patterns we can easily create differing behavior for the same kind of individuals, which can represent different cultures, species or just personalities in the case of humans, all without any changes in the code.

The last statement 'ABM captures emergent phenomena', results simply in the IBM nature of ABM and it's ability to provide a natural description of the system. Under the condition that the behavior patterns and rule were properly identified and implemented, which is a requirement anyway, emergent events will mostly happen by themselves, just the way they do in nature, simply because we add the same input to the same processes.

All of these aspects make ABM ideal for ecological simulations, such as Siccom, which already make use of IBM.[12][13]

Despite all those positive aspects, ABM is not without flaws. There are two big issues that arise when using ABM. The first is gathering the data. Since ABM operates on an individual base most of the time, it requires understanding of all relevant components and how they function for all simulated agent types, at least an estimated base. Gathering this data can prove quite difficult as it requires intensive observation and research on the, to be, simulated individuals, which costs time and other resources.

The other big issue is due to the detailed nature of the style, processed data can get big quite fast and require extensive amounts of calculations, especially in larger scales. This makes optimization of the software more important than in the two other styles. Aside from optimization, there is also the option of combining ABMs with equation-based models.[14]

Even with those shortcomings the advantages for both the scientific and the possibilities in visualization and educational, clearly make it the ideal choice for Siccom.

2.2.4 Pitfalls and Validity

Validity describes to which degree, a simulations results correspond to the real world. When it comes to scientific simulations we always need to consider at which point we can simplify certain elements and which processes can be abstracted into a single algorithm. Since we cannot simulate every little aspect in unlimited detail, as explained previously, we need to decide which elements to simulate and how much we can abstract those elements before they lose their realistic display of nature and with that the desired impact on the simulation.
This decision making is so important to get right because sometimes things can look right, but only as a result of them being forced into the desired shape. For example let's take a look at the clustering of corals in reefs. When we look at coral reefs, we can see that corals of the same species tend to stay together and create clusters.
So how does this result come to be? One possibility could be, that maybe corals are attracted to members of their own species and try to settle closer together and away from possible dangers. After all bonding together is a common strategy to increase the survival chances of each individual and the species itself, as seen in the many fish that realize this in the form of swarms.

On second thought corals themselves are immobile and the larva for reproduction is basically dependent on the water currents for reaching new locations and are therefore quite restricted in their possibilities to control their settlement locations. In fact the clustering is rather a result of corals fragmenting and the fragments settling as new independent corals, close to their source, since the fragments are immobile too and the current can only move them a relatively small distance.
Even though this differences might seem quite obvious to some people, the results might have not been so clear, as the clustering would happen in both cases, but the impact would be quite different. In the 'attractiveness'-Approach the corals would have gained new abilities and the clustering would be equally likely for all species, which could give an advantage to certain ones, that are less likely to cluster in reality, due to a variations in grow speed and vulnerability to fragmentation. This change could cause a chain-reaction, because suddenly certain species are far more likely to take the reef over than they should be and this could impact the life of fishes (currently not part of the simulation), because they are losing hiding spots, which in turn would have it is own cause of events and so on.

As we see even seemingly simple things can have big effects, if not handled properly, but then how do we assure the validity of our simulation? Sadly there is no definitive solution for this problem, if we work with randomness and unpredictability. One thing we can do is simply to compare our results with data from the real world and see how the data holds up. This is however not as simple as calculating a mathematic equation and compare the results to be equal or not equal. Instead it requires interpretation of the data, that considers the input parameters, which obviously should be as similar as possible and also takes the context of the simulation into account. How this can be done for IBM was shown by A. Kubicek et. al. in 2015.[15] The context of the simulation is important as we're always only simulating certain elements and simplify others, depending on this focus different programs can give different insights, while still acting in the same field, in our case a coral reef.
Another thing for comparing the results with reality we should consider is consistency, the simulation must hold up for the same parameters and different seeds for the random number generator, so we should run multiple simulations and compare

all the results. This process also helps to identify and avoid border cases, so it should not only be used for checking the validity but also when running different input parameters. It can also be helpful to compare our simulation with others that have the same goal and use the same data, assuming the other hold up to reality and this is a possibility.

Aside from checking validity we can try to adjust the design process. For this we need to let the design of the algorithms and structure be controlled by the input parameters, not the results. The clustering case previously is a very good example for this. The first idea was based on the results, not the input. If we just take the knowledge that corals do fragment and the fragments are limited in how far they can be washed away from the coral and implement this knowledge, then we would automatically get the desired result. So in general the input data should be the driving force. If we know that the data itself was properly implemented, but fail to reproduce certain events or phenomena, we can assume that we are missing data.

2.3 Siccom

Siccom stands for Spatial Interaction in Coral Reef Communities. Siccom is a software that simulates a coral reef with focus on the competition between corals and algae, originally written, in Java, by A. Kubicek.[1] These two types of organisms are currently the only ones directly simulated as agents. In the case of corals an agent represent a whole coral colony, but as said in 2.1 this thesis and the new implementation will simply call them coral. The following sections will discuss the current implementation and limitations of Siccom.

2.3.1 Current State

Before the writing of this thesis there were two implementations of Siccom, one is the original Java implementation, the other is a plugin for the VR CoralReef written in C++. This thesis adds a third version. The plugin is simply a reimplementation of the Java implementation, specifically to enable a better communication between the VR CoralReef and Siccom, but with only very minor changes, it will be called Siccom++. At the point of writing the Java implementation is publicly available as version 2.0 on Sourceforge.[3] It will be referred to as Siccom_java.

Siccom_java is able to simulate corals and algae as agents. In addition to individual, bigger algae plants it also simulates turf algae, divided into cells, that act similar to very simple agents. In this case each cell covers a certain area, defined by a percentile value and grows according to a mathematical function and a growth rate value, that can be influenced by other phylum, such as corals already covering the area. Corals are divided into branching and massive corals, which represents their differences in topology. For example, a massive coral usually has a smoother, more even surface, whereas a the surface of a branching one covers the branches like a skeleton. All agents can interact and compete with each other, which may cause an agent to stop growing or die.

[3] `https://sourceforge.net/projects/siccom/`

Another possibility for the death of an agent is the simulated water temperature. The Temperature is simulated for the whole reef using a set of pre-recorded real world data, from a reef at Zanzibar, Tanzania. The temperature can cause a coral to bleach or completely die. It is simulated in such a way that it considers the previous simulation steps, as to create a natural temperature progression. In addition it is also possible to slightly increase the average temperature over time, to accommodate for climate change. Another simulated aspect are disturbance events, i.e. dynamite fishing. These happen on a periodic basis and cause all agents in their effective area to be killed.

Aside from getting killed by everything, agents can also reproduce. Every agent can produce a certain number of recruits after each update, which then have a chance to spawn a new agent of that type. Corals in addition can also fragment, causing themselves to shrink, while the fragment has a chance to settle as a new coral close to the fragmented one.

The world itself can be that to theoretically any dimensions, only limited by the technical limits of the programming language. It should however be noted that bigger worlds increase the calculation times significantly, which is why the default reef has a size of 40x40 meters.

The defining parameters of agents, disturbances and environments, as well as the temperature data, can be defined through external files. Additionally a starting configuration for the reef is required through external files. The simulation can also write data to the drive, about the disturbances and existing agents.

For the agent management, Siccom_java version uses a framework called MASON[3], which was substituted by the C++ plugin with the FLAME[4] framework.

Siccom_java itself has no interface for communication with an external software. The only existing possibility is a network communicator, implemented by Anthea Sander for her master thesis.[16] This network communicator is able to send the current state of the agents and temperature over a TCP connection. It is also able to receive temperature changes. The C++ plugin communicates directly through C++ and has a few more possibilities for interaction. These possibilities include the starting and stopping of the simulation, letting the simulation free or step wise and to trigger a disturbance with specific coordinates. The temperature setting is still a possibility. In addition, due to the C++ nature the data doesn't need be requested every time but can be pointed to and used directly by the VR CoralReef.

2.3.2 Limitations

The first obvious limitation is, that the only two simulated phylum are corals and algae, despite the many other organisms present in a typical coral reef. In addition the only two types of corals present are branching and massive ones despite there being many more growth forms, as seen in Figure 2.4. This means it is not possible to add coral species with other growth forms, just through the configuration files. Another result is redundancy in the different coral classes.

It is also not possible to add any new phylum to the simulation without adding it explicitly to the rest of the code base, such as the main-loop and the configuration reader.

The disturbances also can't be defined very precisely to represent different real world events. For example a disturbance can't distinguish between a mechanical or

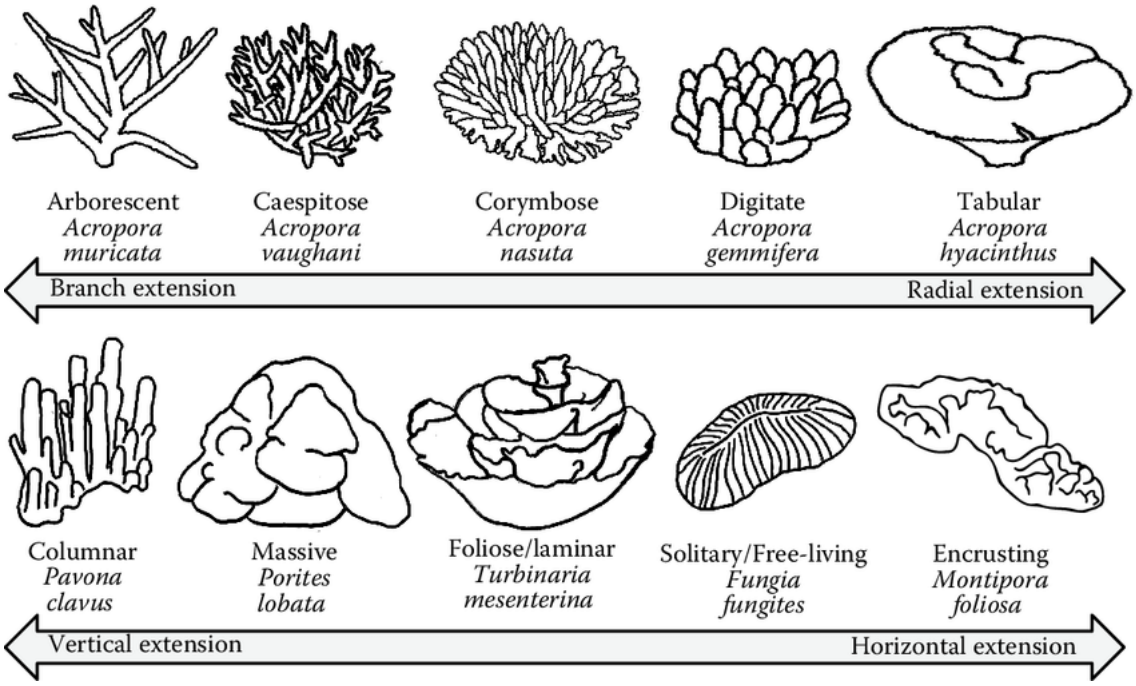

Figure 2.4: 'Major growth forms of scleractinian corals arranged according to their major growth axis. Change in area of occupation (standardized for colony size by calculating change in arithmetic mean radius) is the most suitable growth metric for corals to the right.' [17]

chemical cause of destruction, which would impact the various organisms in different ways. Further the simulation does not consider the ground of the simulated area, such as sand or stone and consider the third spatial dimension only in form of a surface factor, for each species to estimate the surface area based on the ground area of a coral. Both aspects can be important to the development of a reef, since corals require a solid surface and enough light (which can be obstructed by other phylum overgrowing the coral in the third dimension) to survive. Interaction wise it's missing various functionality, that would be desirable, especially in software like VR CoralReef. These include, the manual spawning and destruction of agents, pausing the simulation and requesting data in a more detailed way, than all or nothing. Basically recreating every process that could happen naturally during a simulation.

Aside from these aspects the simulation requires a starting configuration and real world data for the temperature. So it is not possible to just start a simulation with a random distribution of agents and it's also not possible to generate temperature from a generic function, therefore needing data for every simulated location.

2.4 Similar Software

This section will list 11 similar simulations or papers which use simulations similar to Siccom, that exist and what their focuses are. The listing will be split into to categories, the first being coral reef simulations, the second being ecological simulations in general.

2.4.1 Coral Reef Simulations

CORSET is a rather large scale, equation based model. It uses dynamic equation for simulations on a local scale, individual reefs and connects these reefs through ocean transport of larva propagules. It is designed for regional scales of $10\hat{2}$-$10\hat{3}$ km, over several decades and published by Jessica Melbourne-Thomas, in her PhD thesis in 2010.[18]

Another very similar simulation to Siccom was used in a paper from 2007, by P. J. Mumby, A. Hastings and H. J. Edwards. The model was used to identify critical thresholds for reef resilience against disturbance, in Caribbean reefs, based on grazing and alga population.[19] With a focus on climate change, a paper from 2009 used fishes as the model group in their simulation.[20]

Two papers from 1976 and 1992, use computer simulations to research the relation of light and coral growth. One being focused on growth of a specific species[21], while the other considers reefs in general.[22]

Another aspect where simulations are used is the impact of fishing on a reef, as seen by the paper 'A coral reef ecosystem-fisheries model: impacts of fishing intensity and catch selection on reef structure and processes' from T. R. McClanahan in 1995[23] and 'Modelling the effects of destructive fishing practices on tropical coral reefs' from S. B. Saila, V. L. Kocic and J. McManus in 1993.[24]

A usage of a cellular automata is demonstrated by O. Langmead and C. Sheppard, who used a cellular automaton of different disturbances in coral reefs in 2004.[25]

At last two more unique applications of computer simulation can be found in two papers from 2004 and 2006. The first, written by H. Yamano and M. Tamura uses simulations for the satellite detection capabilities of bleach reefs. [26] The second, written by C. M. Kunkel, R. W. Hallberg, and M. Oppenheimer found that coral reefs at shore sites can reduce the impact of tsunamis by up to 50 percent using simulations.[27]

2.4.2 Ecological Simulations

An example for a combination of EBM and ABM, as they are described in chapter 2.2.1 and 2.2.3, is delivered by G. V. Bobashev, D. M. Goedecke, Feng Yu and J. M. Epstein in a paper about epidemiological modeling, where they used an ABM as a starting point and switch to an EBM, when enough data is provided by the ABM.[14]

Another example from the ecological modeling workgroup from the ZMT is this swarm model, that also uses a cellular automata for the underlying food field. The simulation can be seen in Figure 2.5.[28]

At last, a very detailed simulation of parrot fish in various biomes is Kitt, shown in Figure 2.6. The parrot fish agents have their bodily functions simulated in detail, such as feeding, digestion and metabolizing. In addition each simulation step represent a second in real time.[29]

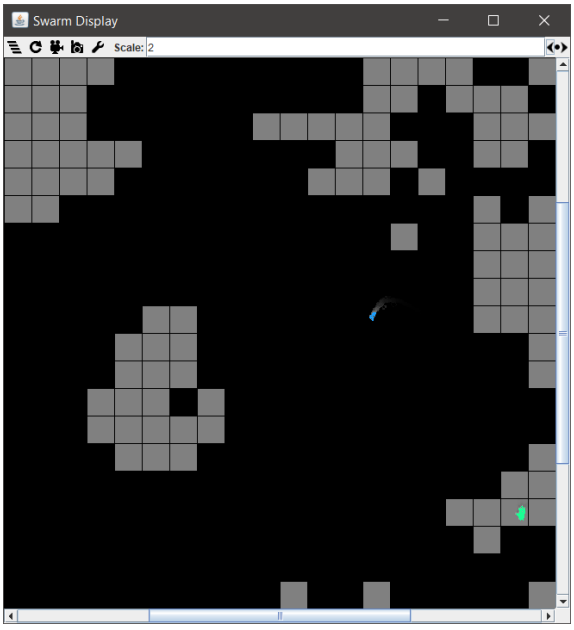

Figure 2.5: A swarm model that uses abm for the swarm (blue and green) and a cellular automata for the food field underneath (grey squares)

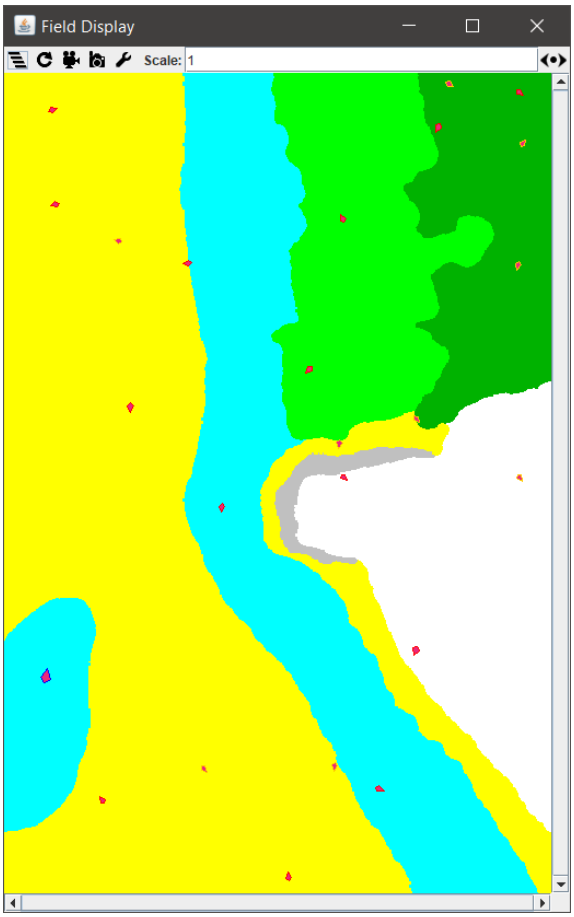

Figure 2.6: A parrot fish simulation. The fish can be seen as red diamond shapes. The different background colour show the different biomes: sand (yellow), reef (cyan), mangrove (bright green), seagrass (dark green)

2.5 DDD - Domain Driven Design

Domain Driven Design (DDD) is a architecture style that gained popularity in the last years, it is also used in the VR CoralReef 2 Masterproject. Here the basic concept of DDD will be explained and why it was chosen over Model-View-Controller (MVC) and Entity-Component-System (ECS). These two were chosen for comparison, as they both can be considered quite popular and both share a similarities with DDD.

The idea is to divide the Software into three distinct layers, each with a clear role insight the program and rules how the different layers communicate with each other.[30] This layer style is the most common and simple style. There are other options and more elements that can be taken into consideration.[31] However since our software contains relatively few truly different processing elements (mainly entities, disturbances and the world(reef)-state) and it is unlikely that expansion will introduce new elements, it was decided that this simple layer structure is sufficient.

In addition this is the concept for the VR CoralReef 2 and therefore supports the usability aspect, this will be elaborated at the end of the section. The Layers can have various names, in our case we call them adapter, administration and domain. One important part is that each layer may only use classes and functions from their own and lower layers, but not from layers higher in hierarchy. This ensures the modularity of the software, but can require that certain states are mirrored in the adapter and domain layers. It also requires that data may either be collected with a second iteration in a higher layer or be transmitted using return values. The layers and their roles look as follow:

- **Adapter:** This is the highest layer in the hierarchy, its role is to represent the software and its persistent objects to each other, as well as outside clients. The represented objects on this layer therefore only contain very little actual information or logic, they just serve as a proxy for identification and access to the actual logic object laying in the Domain. The communication with other programs, such as visualizations and database, happens through handler classes. These classes consist of an interface that gives access to all required functions for external sources and manage the actual processing, such as filtering the requested data or converting it to another format. There are usually two kinds of handlers, request handlers for other programs that use the software and database handlers and the like for communicating with a database or other form of saved data, which is also the reason why they are usually controlled by the software itself and not from the outside. A further explanation of the request handler will follow in chapter 3.3. Aside from representative Objects and Handlers this layer also controls the state of the whole simulation and the update processes, the responsible classes for this are therefore called controller.

- **Administration:** This is the second layer of the hierarchy. It is used by the adapter when updating objects in the domain and its only purpose is to collect all information needed by the domain-object for its update process. As a result this layer does not contains persistent objects and its logic is quite simple in general. Due to their managing nature the collecting objects are called managers.

- **Domain:** This is the lowest layer in the hierarchy and as a result can only use classes it contains. The role of this layer is the actual complex logic of the software, therefore they usually contain the biggest part code wise and also the most complex code. The classes of this layer realize mostly counterparts to the representative objects in adapter and implement their logic, they are therefore called logic. This layer can sometimes have helper objects that are not directly represented in adapter, as their existence as actual objects is irrelevant and their information can be accessed through other logics. An example of this would be the branches of a coral or the temperature contained by the world logic. For sake of modularity the logic objects should at best only act on themselves, this makes it possible to remove and add logic classes without disrupting processes in other classes to much.

Figure 2.7 visualizes the key points of this architecture.

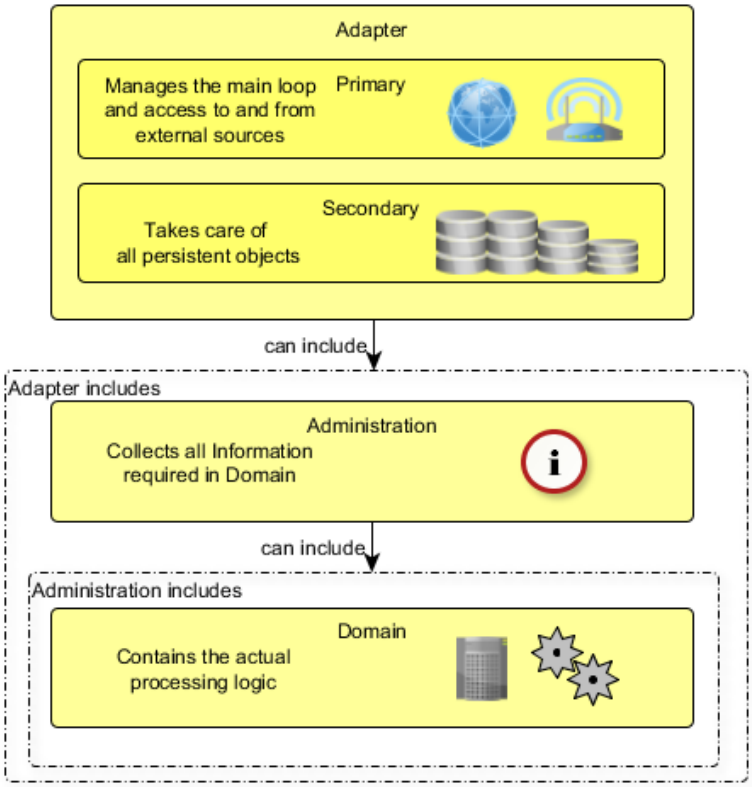

Figure 2.7: An overview of DDD, high lighting the layers and their roles, as well as the include relationship. The symbols visualize the role of each layer.

This three-way division of the software already reminds one of the MVC style. However there are quite a few differences. While the model can certainly be compared to the domain layer, view-controller don't match up with adapter and administration.

The purpose of view is usually to, well display an user interface as the name suggests. This aspect however is not even directly considered as we only have the request handlers and any kind of display is simply an external program using these handlers. Sometimes the view can also be represented by just some kind of API, in this case the request handlers can be considered to fulfill the role of view.

The role of the controller is to restrict the access to the model and ensure that every operation is valid. This is fulfilled by the handlers in DDD and the controllers. Also administration could be an element of a controller but does not have to, since MVC does not specify what kind of control is actually needed, it only specifies the process flow. Also representative objects such as in adapter, would have their place in the model, in MVC.
So we can see that while it seems similar there are quite big differences between DDD and MVC. MVC was also designed for web interfaces and other applications with many separate but simultaneous users, that might accidentally or even with malicious intend, corrupt other peoples data.[32] These are however cases that do not really apply to our simulation, as we have no shared data and access is limited anyway.

Also nearly all interactions that are possible could actually be considered useful. The only case where data could be corrupted would be scientific simulation runs, these however do not usually offer much access after start anyway and in the case of educational runs the worst case would be that the displayed state of the reef is a very unique or unlikely configuration, but no real harm would be done. So MVC is not that suited for our purposes.

ECS on the other hand is quite popular for real time applications and commonly used in game engines such as Unreal Engine 4.[4] So let's compare ECS and DDD.
The concept behind ECS is to encapsulate various elements of a simulation in components and to build entities out of these components. These elements can include for example an AI, the possibility to move or being able to interact with a physics simulation.

The great advantage of this concept is that we can create many different entity types out of reoccurring aspects (the components), even if we do not know what we might need during the implementation. This obviously is a typical case for game engines, as they are used to create a huge variety of games with unknown types of entities or agents. In our case however we know pretty good what we need, that currently being corals and algae. And not only that, but in scientific simulations we also need to consider how these components actually interact with each other, a fish for example could have a feeding, metabolism, movement and AI component, all of which require information from each other.
ECS however works so great for game engines since in a game we usually don't care how something is accomplished as long as it gets done, therefore the components can be mostly self contained and don't conflict with each other. But if the components need to exchange information, we add unwanted dependencies, which make it very hard to decide what elements should actually be represented by a component and which not.
So ECS does seem far better suited for creating generic frameworks as opposed to direct simulations and therefore is not suited for our case. In chapter 2 it was also described how important it is to identify what we want to research and therefore how detailed the various aspects need to be implemented. This creates another advantage for not using ECS, since we don't have to implement each component in X different ways for the various degree of detail, but instead can directly implement each logic object according to its requirements.

[4] `https://www.unrealengine.com`

So in direct comparison DDD is better suited for our case than MVC and ECS from a technical point of view. But that is not the only thing that supports DDD, as the goal of this reimplementation is usability and extendibility, we should take a look how these aspect fit into DDD.
The clear rules and layers, which can be summarized on a single A4 Page, make it very easy to understand the structure and where to find which kind of classes, even for people new to DDD and without computer science as their focus.

Even though it should be noted that these aspects are partly is so easy, due to the focus on the core concepts of DDD, when we add more of the deeper elements of DDD it can of course become harder to understand.
The mostly self care-taking logics also make it very easy to change the behavior of our entities, add completely new ones or remove existing. The combination of this clear structure and self care-taking make it even possible to ignore most parts of the code base during implementation. These aspects are extra important for us, since as mentioned in the previous chapters one of the most common developers for this software will include people also working on the VR CoralReef, that follow the same structure, so they won't need to wrap their head around two different architectures, which is especially important for the student projects with limited time.

Chapter 3

Software Architecture

In this chapter class- and sequence-graphs are used to explain the architecture of the software. Since this is about understanding the general architecture the class diagrams will only contain a description of the classes and not a listing of their attributes and functions. Also this chapter won't cover how exactly the classes achieve their roles, this will be explained in the next chapter about algorithms.

3.1 The general Architecture

Figure 3.1 shows the total architecture, the three layers are clearly visible. The adapter layer is split into primary and secondary. This distinction however is not a necessity, it's only another element keep order in the code. The secondary part usually contains all the classes representing or working with persistent objects, the latter being in this case the config reader, which could also be a database handler, if we would use one. The primary layer simply contains everything that is left in adapter.

Another rather obvious aspect is the color coding, classes with the same color belong to the same simulation aspect. Most of them should be self explanatory, but here is a little overview of the different colors:

- Grey: These classes represent general functionality, that is not specific to this simulation.
- Gold: This is just the representative simulation/Siccom-object.
- Orange: These classes cover everything regarding the world and environment. This includes the all present turfcell algae.
- Brown: covers the disturbances
- Light pink: These two classes cover the spawning of new corals and algae, based on manual, automatic and fragmentation spawning.
- Yellow, blue, green: These include everything regarding the different entities. Yellow represents the abstract entity classes themselves, while green covers algae and blue the corals.
- Violet: GUI.

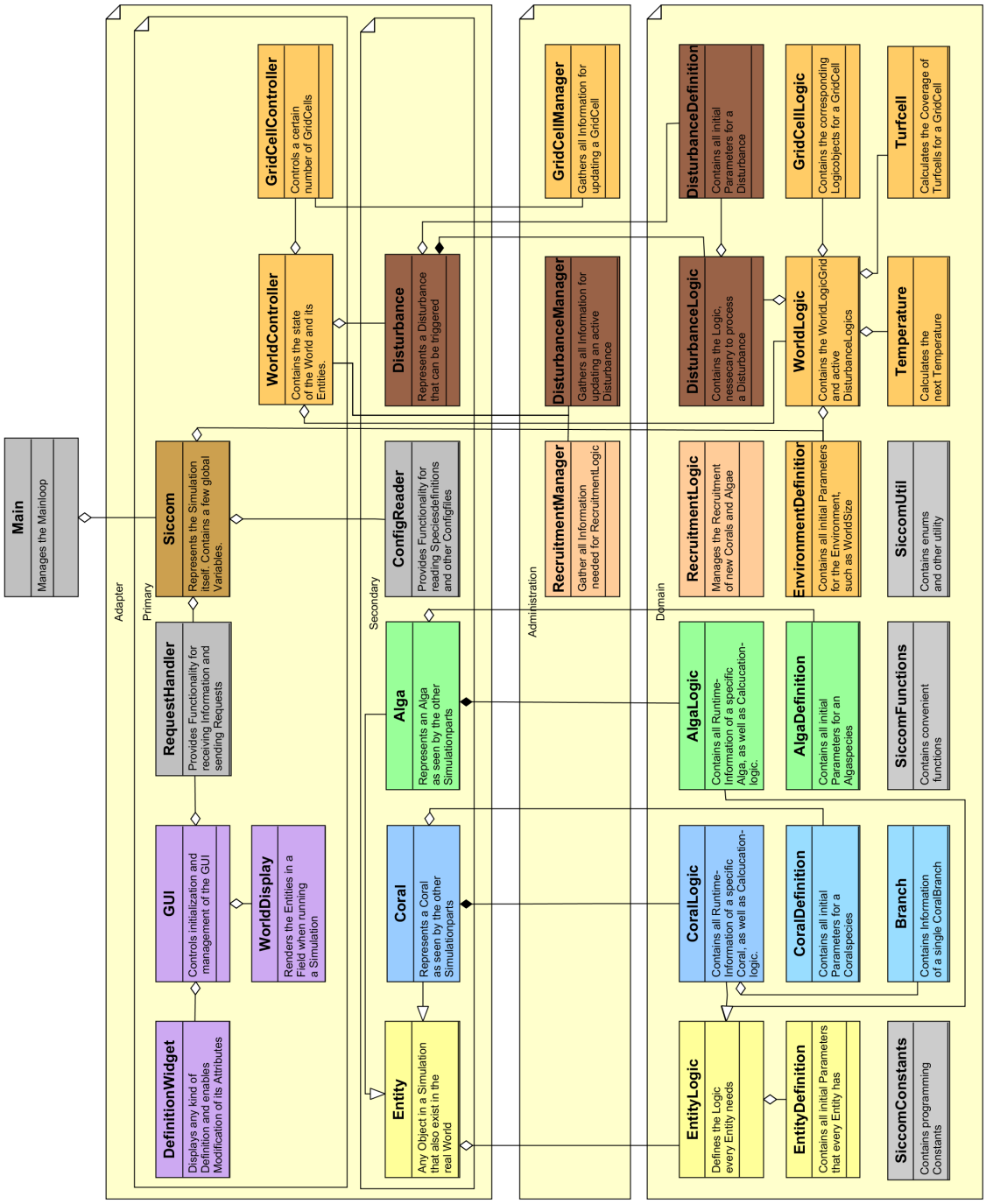

Figure 3.1: Simplified Classdiagram of Siccom, showing all classes in the new implementation and their basic relationships. Each class has a short description of its contents.

The last thing that might stand out is that the main class is out of context, this is simply to represent that the main class can be realized in different ways. While the main loop will generally be the same the simulation can be controlled by other software, such as the VR CoralReef.

3.1.1 Entity

An entity is any simulated object that represents an individual organism. Currently this covers corals and algae, turf cells are not included since they aren't simulated on a individual basis and disturbance are no entities since they are no organisms.

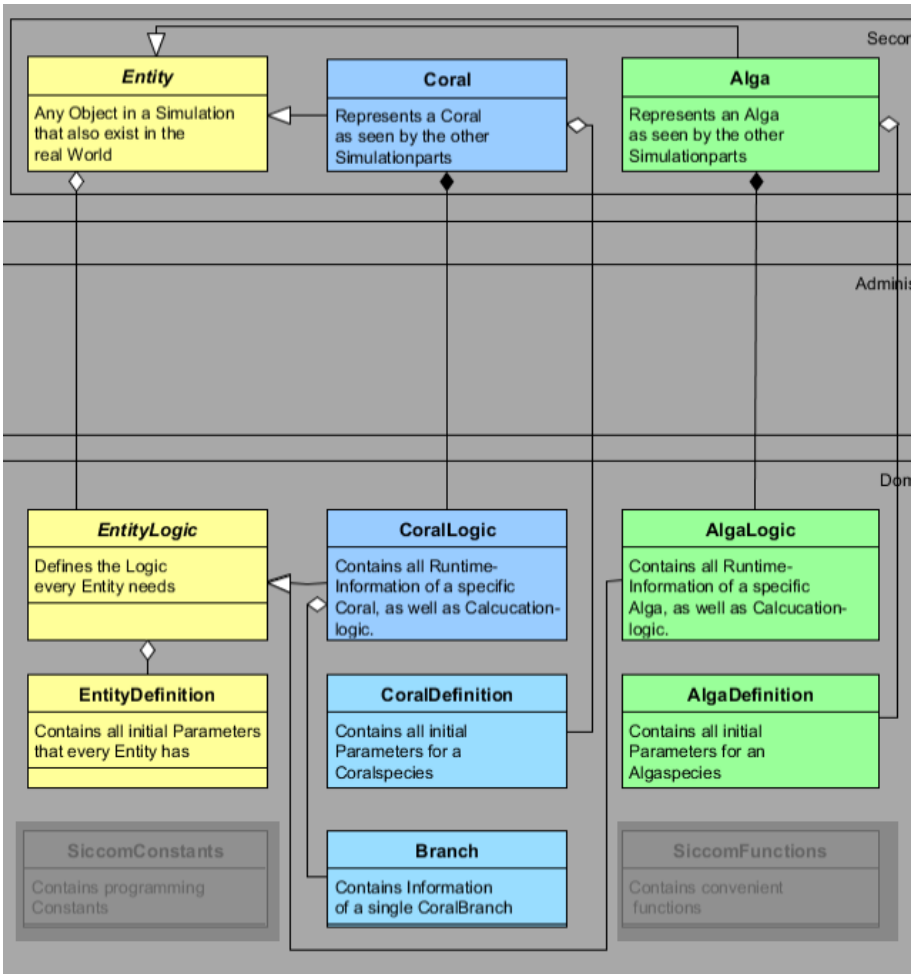

Figure 3.2: Closer view of the Entity part in the architecture, highlighting the classes belonging to entity

The entity structure, as seen in Figure 3.2 consists of three core elements: Entity, EntityDefinition and EntityLogic. Entity is simply the representation Object for all entities, it only knows the ID, the corresponding Logic and Definition of the Entity. All Entity classes are abstract, so each specific type of entity needs to implement a more detailed version.

Due to technical limitations the abstract Entity class does only contain a getter-function for the Definition that must be overridden, but not a definition itself, since attributes cannot be overridden. The combination of the getter-Function and a missing definition attribute ensure that each specific implementation of an entity type also includes a specific custom Definition.

The Definition contains all Attributes for Entities, that are known before runtime and are initially the same for the whole species. Such attributes can contain Information like the growth speed, but also more functional things like the species name and the rendering color. Those attributes are in biology referred to as parameters. Other than that the definition usually does not contain any logic. Specific

for this implementation, it contains a function for initializing a map data type with function-pointers, which can be used to generically interpret new types while reading the configuration files or displaying them in the GUI.
The logic on the other hand contains the attributes, that are specific to each individual entity object and only exist during runtime. Such attributes include for example the position. Aside from these attributes, the logics also contain the algorithms and functions that make up the specific implementations, such as coral and alga.

Some types of entities require specific helper objects, such as the branches for the coral class. These helper objects should only interact with the entity type they belong to and be independent from the rest of the simulation.

One thing that stands out when comparing the coral structure of this implementation with the original Siccom, is that I unified the massive and branching corals into one single class. The reason for this is simply that the main difference between massive and branching corals is the topology in the different growth forms. In addition massive and branching don't even cover all the growth forms.[17] [33]
Also while the branches are obviously visible in branching corals and cause them to be more vulnerable to fragmentation, they can still serve a technical purpose in other growth forms. Independent of the form each coral can adjust to their surrounding and for example stop growing on one side but still continue on the other, therefore these branches can also be used in massive corals for describing the outline of the coral surface and to determine for example a collision.
The difference between the growth forms would then only be in the growing complexity, the branches of a massive coral might only know one direction and can only stop growing, while the ones a branching coral can change their direction and therefore adjust instead of just stop growing. It would then be left to the visualization to interpret the branch data according to the coral type.

It might even be possible to create an algorithm than generates any growth form simply by interpreting a fixed amount of variables, that may even be derived from simply looking at a real growth form, this however exceeds the scope of this thesis. Overall the unifying reduces redundancy and makes it easier to implement completely new types of corals, as they are now only dependent on their definition values and not the code base.

3.1.2 Disturbance

A disturbance is any event that causes damage to entities in the reef. Such events can represent dynamite fishing, an anchor or an oil leak, for example.

The disturbance structure is basically the same as for the entities. It is shown in Figure 3.3. A Disturbance consist of the representative disturbance object, the definition containing all information known before runtime and the logic object, containing the algorithms for executing the effects of the disturbance.

Other than the entity class however, disturbance is not abstract an the exact realization is only defined through the values of the definition, that include attributes such as size and damage potential. A disturbance can be a single or a reoccurring event, defined through the definition. A disturbance also has no consistent state that changes as it's only interaction with the other simulation components consists of it's execute function. The last difference is that the Disturbance classes have their own manager, since collecting the affected entities is a little more complex than for

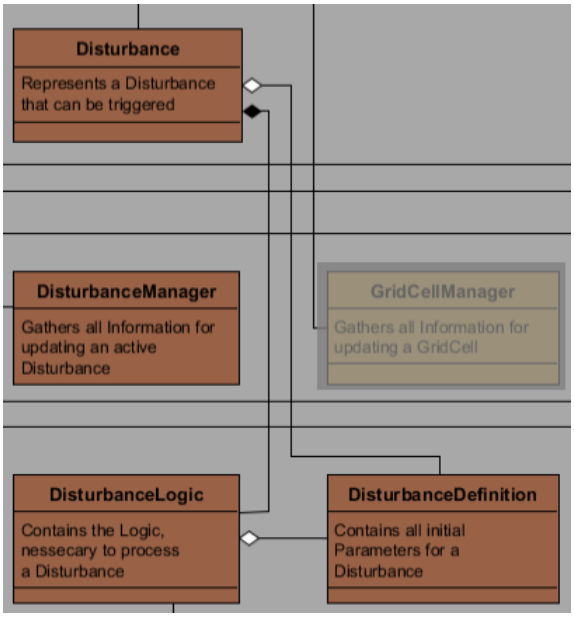

Figure 3.3: Closer view of Disturbance part in the architecture, highlighting all classes belonging to Disturbance.

entities as it depends on the shape and size of the Disturbance.

The Disturbance classes currently contain various optional attributes, which can be used for further specify the effect of the disturbance. This includes the option to specify which species or entity types are affected and even define the damage potential for each species. If these values are not set, the logic will simply use the default values. These extra options require more data than the general approach, but enable a more realistic simulation. For example in some reefs the main problems may arise from dynamite fishing or untrained divers, which both cause a mechanical disruption that could be far more damaging to corals as it is to algae, due to their difference in flexibility. In other reefs the main issues may be oil leaks or other chemical disturbances that affect the corals and algae in different ways. It can also be used to better integrate new type of events, that may for example arise from climate change.

3.1.3 World

The world part of the architecture contains most simulations elements, that are relevant at runtime. This includes the world itself that contains the various entities and disturbances, but also some global elements such as the temperature and turfcells. They can be seen in Figure 3.4.

The first part of the World consist of the World- and GridCellController. The WorldController contains a list or maps of all Entities, active Disturbances and GridCellControllers as well. It represents the World and its inhabitants during runtime and manages the update processes of all simulated objects. The world is divided into a two dimensional Grid for a more efficient data structure and to avoid unnecessary comparisons between entities.
This grid is realized through the GridCellLogic class that contains the objects in its part of the World and are managed through the GridCellController. The grid is stored only in the WorldLogic, since the entities can be accessed by their ID

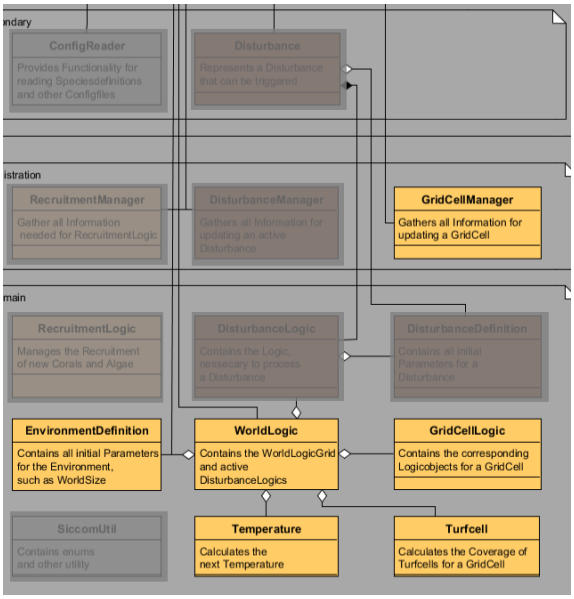

Figure 3.4: Closer view of the world part of the architecture, highlighting all classes belonging to the world

through the map in WorldController and the grid cells themselves are not relevant to other programs. Each GridCellController manages multiple GridCellLogics. These GridCellControllers enable the World to be easily managed in parallel, through multiple threads.
Other than encapsulating the WorldController functions for each GridcellLogic, the GridCellControllers don't add anything else to the WorldController. When updating the GridCells, the Controllers update their GridCellLogics in a random order, to avoid favoring one cell over another. They don't need to gather the neighboring cells each update, since these are collected for each entity when it spawns.
The WorldLogic mirrors the WorldController state on the Domain Layer and enables access to the corresponding Logic objects. It also contains the temperature and turf cells. Complex Logic is currently not present, as the World has no special processes of its own.

The EnvironmentDefinition contains, similar to the other definitions, information about the world that are known before runtime. These information include the world size, the number of GridCells and growth information for the turf algae.
The Temperature is managed in the equally named class, that ensures that the temperature progresses on a realistic basis.

The last part in the world element consist of the turf algae, which are represented as cells, similar to the general grid, but independent and the cells are quite a bit smaller. Turfalgae is simply a kind of algae that grows similar to moss and serves currently only a supportive role, for determining where corals can grow. Due to this supportive nature the turf cells are currently only represented in the Domain, however when they might be expanded to sediment cells, it could make sense to also represent them on the adapter. Such an expansion could also justify the implementation of an cellular automata for the sediment cells. Currently however the complexity of the turf cells is quite limited and the individual cells do not consider their neighbor cells, therefore the work of implementing a whole Cellular Automata is not adequate to the turf cells.

3.1.4 Recruitment

The Recruitment classes take care of the spawning of new Entities. These can come from reproduction, fragmentation or manual spawning. The recruitment manager currently are only a pass through to the RecruitmentLogic.

3.1.5 Other functionality

Here a short summary of the remaining classes. The GUI class is the main class that initializes and controls the simple visualization of the software, for this it uses the RequestHandler to communicate with the simulation and will mostly be used for configuration and scientific simulations, otherwise the VR CoralReef will replace these functions. The WorldDisplay and DefinitionWidget are simply helper classes. The current GUI is implemented using the Qt framework.

The last three files SiccomConstants, SiccomFunctions and SiccomUtil are just name spaces providing helpful Functionality used in the Domain and help to keep the simulation independent from framework such as Qt and Unreal.

3.2 Update Processes

This section will explain the general update process, as well as the basic update cycle for entities themselves. Some of the smaller classes, such as temperature won't be covered here as they are purely independent from the rest and therefore have no relevance for class interaction, they may however get a little explanation in the algorithms chapter.

The disturbance will only be roughly covered since it can be summarized with the following: 1. collect all entities in the area of effect. 2. kill each entity with a certain probability, based on the damage potential. The killing consist only of setting the alive value of an entity to false, this doesn't follow the read-only policy, but is still acceptable since this specific value will always start as true and then only be set to false, therefore we do not need any synchronization, since the value can only be 'corrupted' to the same value, so not really corrupted at all. This structure enables far better management of disturbances as they can be run in parallel to the rest, of the simulation or in between steps, without affecting other processes.

Figure 3.5 shows the main process of updating the entities. As we can see the process starts in the WorldController, which uses the GridCellController to trigger the update Function on all GridCells. This happens in parallel as each GridCellController, works in its own thread, realizing the main threads of the simulation. The GridCellController then uses the GridCellManager to gather all the information for each GridCell and causes them to update. The GridCell updates happen in parallel to. The GridCell itself then just simply iterates over its entities and calling their update function. The update function returns a data package with information for spawning new entities, which gets saved in the GridCell.

After all updates have been executed, each thread that created sub-threads, iterates a second time over the objects it updated and collects the returned recruitment data. This structure ensures the read-only policy, so we don't need any synchronization between the threads. Of course this behavior could also be applied to the entity updates themselves, causing each entity to run in its own thread. But since

this implementation utilizes only parallelization on the CPU, it's quite limited in the number of threads that can be effectively used and could create an processing overhead due to thread management, that would be greater than the gained performance. For this reason it was decided to not parallelize on this level. However in an massively parallel environment that utilizes the GPU, this approach could be very well suited and even support the realistic display of natural processes, which will be elaborated at the end of the section. It is important to note that none parallel update processes need to be in a random order, as otherwise the first entities to be updated would gain an advantage over the last.

After the update processes and the data collection has been finished the WorldController transmits the recruitment data to the RecruitmentManager, which uses the RecruitmentLogic to create the entity objects according to the data and also makes sure they have a valid position to spawn in. At last the WorldController uses the return entity objects and actually spawns them, meaning adding them to the list of entities, during this process it also removes the dead ones.

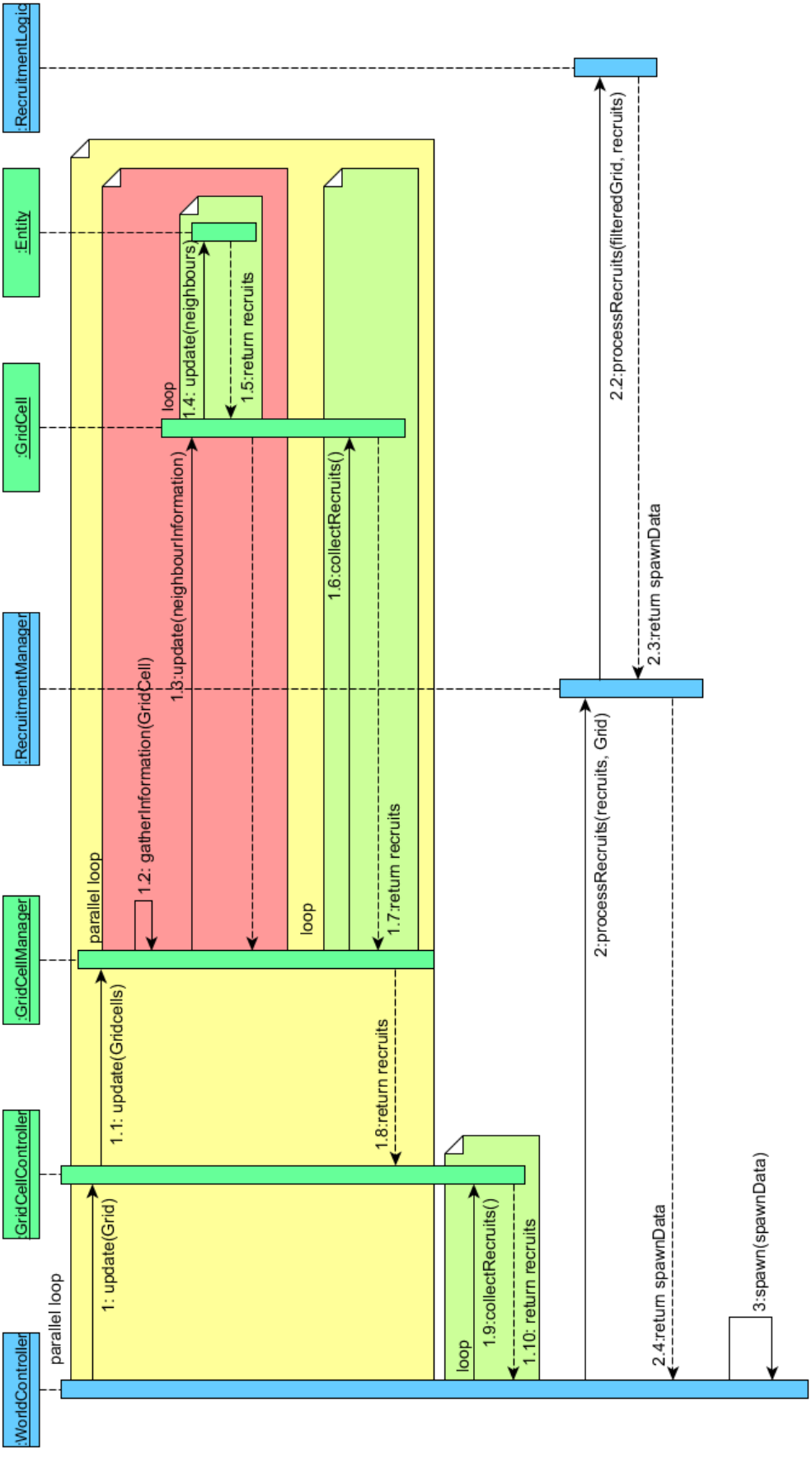

Figure 3.5: General Updateprocess of an Entity, as a sequence diagram. Each change of background color indicates a new set of working threads being introduced.

3.2.1 Entity

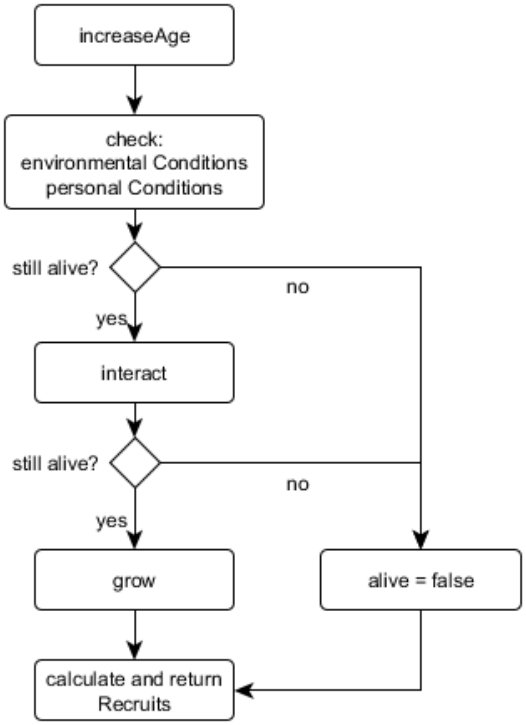

Figure 3.6: The generic update process of any entity.

As seen in Figure 3.6 the update process inside an entity is pretty simple. It starts with increasing the age and then goes over to check certain personal and environmental conditions. These currently include the temperature and age in relation to acceptable values. The next step is the interaction with neighboring entities, followed by the growing step.
If at any point during the process, the entity would die the following steps get skipped an it gets marked as dead by setting the alive boolean to false. Independent of the status the last step will always be calculating and returning the recruits, as even if an entity dies it is possible that some fragments survived or shortly before the last larvae were sent out.

This update process is just an generic example, since the entity class does not specify how the update should behave, but only that such a function is available, therefore it is up to each specific implementation to add/remove steps or simply change the order. The coral class for example has a chance to bleach after the normal condition checks.

3.2.2 Parallelization and Determinism

As mentioned beforehand the parallelization has not only an effect on the performance but also the realism and scientific value of the simulation. While parallelization is usually beneficial to performance, we lose the aspect of determinism.

Determinism can be useful for debugging purposes and recreating specific results, helpful from a scientific and technical standpoint. However as elaborated in chapter

2.2.4, a desired aspect of a scientific simulation is consistency and that border cases should not be used for representative research, at least not if they can't be justified by the configuration. For the same reasons it is common to run multiple simulations with the same configuration and different random-numbers-seeds, for creating a reliable image of the simulation results. Therefore the performance increase outweighs the loss in determinism.

In addition parallelization has an interesting aspect on displaying the natural processes. Since as we know in reality everything happens in parallel, as animals, plants and humans are not bound by the actions of their surroundings, they just react to them, if needed. So one could argue that any amount of parallelization brings the simulation closer to the natural processes it displays, with the closest possible level being, every entity running on its own and therefore parallelization actually increases the realism of the simulation.

3.3 RequestHandler and ConfigReader

The Requesthandler is an interface for the GUI or other software to the functions of the simulation. Its job is not only to provide an interface but also to format the data, depending on the requirements. Therefore any implementation of a RequestHandler is customized for a specific purpose. The current implementation only covers C++ in general, for usage from any C++ context, but there could be an implementation for a network interface or usage in a different programming language.

The ConfigReader, as the name implies, reads the configuration files and creates the definitions accordingly. It currently uses json as file format. It can also be used to read and write the current status, for processing the data afterwards or continuing a saved simulation run. An alternative implementation of this would be a database handler that does not work with json files but instead communicates with a database, such as a SQL database.

The abilities of, the RequestHandler to spawn new entities and the ability of the ConfigReader to continue a simulation also reduces the functionality loss, due to the loss of determinism, since those functions can be used to define a specific reef with special conditions that couldn't otherwise be recreated. This could also be used for ensuring that every user gets the same basic reef, every time the VR CoralReef gets started.

Chapter 4

Algorithms

This chapter will explain the algorithms used in nearly all classes, using pseudo code. It will explain how the algorithms work, key aspects that need to be considered and why changes were made, if any compared to Siccom_java. The GUI classes, GUI, DefinitionWidget and WorldDisplay won't be covered as they do not contain any logic relevant for the simulation itself. The utility classes SiccomConstants, SiccomFunctions, SiccomUtil, SiccomEnum also won't be covered as they only contain helping data types or general mathematical functions, that help reduce redundancy and make the code more readable, but nothing that is crucial to the algorithms.

4.1 Siccom

The Siccom class is the representation of the simulation as an object. It contains very little own logic. The logic consists of only four simple functions and a number of variables:

```
Variables:
  IDCounter;
  simulationSteps;
  numberOfThreads;
  configurationName;
  bool: running;
  bool: pause;
  simulationWorld;
  environmentDefinition;
  configReader;
  entityDefinitions;
  disturbanceDefinitions;
  randomNumberGenerator;
  DefinitionFunctionPointer;
```

The *run* function realizes the main-loop and simply updates the world until it is stopped from another thread, by setting the running bool to false or if the simulation has reached its target runtime, measured in simulated steps.

The *step* function is nearly identical with the difference, that it only triggers one update of the world and then returns if the simulation has reached its target runtime, this function is more useful when the simulation is controlled by another program, such as the VR CoralReef.

```
run()
{
  while(running && steps < runtime)
  {
    if(pause)
      skip;
    else
      steps++;
      world->update();
  }
  finish();
}

step()
{
  steps++;
  world->update();
  return steps >= runtime;
}
```

The third function is *finish*. It is only used to trigger the world to stop if any extra clean up is required after an update and it explicitly sets the running variable to false, to notify other threads or programs that the simulation has reached its end:

```
finish()
{
  world->stop();
  running = false;
}
```

The last function contained in Siccom is initLogging:

```
initLoggin()
{
  for(desiredOutputs)
    outputFile = createLoggingFile();
    addFilter(outputFile);
}
```

This function simply creates the different logging files for the desired output data and configures them so that they can be written to, separately.

4.2 Entity

Entity itself has no functions as it's only a representation of its logic object and definition. The original Siccom_java had no general entity classes, instead it had *Alga, MassiveCoral, MassiveGroup, BranchingCoral, BranchingGroup* and *CoralGroup* as their own classes, despite them sharing many aspects. *CoralGroup* was the only class that used polymorphism, as it was an abstract base class for the other two Group classes.
The aggregation under a generic Entity class makes it possible to write the algorithm on a generic basis and therefore adding new and removing different entity types without big changes in the code base.

The Group classes generally contained logic working on an entire species, while the Coral classes contained the logic for an individual massive or branching coral.

However since even the algorithms in the Group classes considered the individual corals, all coral logic is now contained in the class CoralLogic. On the other hand the EntityDefinition class was introduced, containing all species specific variables, as explained in chapter 3.1.1.

4.2.1 EntityDefinition

The variables in this class are all known by value before runtime. They contain general, fixed values relevant for all types of entities. This contains for example the species name. In addition each specific implementation of an entity type, such as Alga or Coral, expand these number of variables and functions if needed.
EntityDefinition has no complex logic, but it has a number of static Proxy-functions which can take one of three forms: getter, setter and constructor. They are defined as follows:

```
Getter:
  static GetSetValue getValueProxy(EntityDefinition)
  {
    return GetSetValue(value);
  }

Setter:
  static void setValueProxy(EntityDefinition, GetSetValue)
  {
    EntityDefinition->value = GetSetValue->content();
  }

Constructor:
  static EntityDefinition()
  {
    return new EntityDefinition;
  }
```

GetSetValue is simply a wrapper struct that can hold any type of data required. The only difference in between these functions and normal getter, setter and constructor's is that they are static and act as a wrapper that can act on any EntityDefinition, without an explicit call in the code. This way they can be generated and stored in a map as they are in the only function of EntityDefinition, *init*:

```
  static init()
  {
    for(allVariables)
      functionsMap->insert(valueName, getter, setter, dataType);

    accessorMap->insert(speciesName, constructor, functionsMap);
  }
```

With these static functions it is possible to create and define any kind EntityDefinition without knowing any specific variables, by just matching the valueName and speciesName to their counterparts in the configuration files. This makes it possible to add a new entity type and have its values definable from an external file without any changes to the configReader.

4.2.2 EntityLogic

Even though all EntityLogic are required to have an *update* Function and a getter for their specific EntityDefinition, they don't share any function logic aside from basic getter and setter functions. The only mentionable aspect is that each entity logic contains a list of pointer to maps of GridCell inhabitants. These maps contain all other entity logics that could be close enough to affect the list owner. This way the neighboring entities do not need to be collected for each update, but are instead available as soon as they are spawned and also not considered when they are removed again, without any extra processing.

AlgaLogic

The update process of a Alga is as follows:

```
1  update()
2  {
3    recruits = calcRecruits();
4
5    if(height > fragmentationHeigh) //too high
6      fragments = fragtate();
7
8    age++;
9    if(age > maxAge) //too old
10     die();
11
12   if(random(grazingProbability)) //getting eaten
13     die();
14
15   interact();             //interact with other entities
16   if(killedByInteraction)
17     die();
18
19   grow();
20 }
```

At first the recruits and fragments for new spawns are calculated. This happens at first since one step equals a whole month and even if the alga dies in this simulation step, it is quite possible that it cast larvae out or fragmented before it actually died.

After the recruits are calculated the alga ages and has three chances to die. The first is simply being to old, the next is getting eaten and the last is to be killed by competition. If after all three stages the alga is still alive in is allowed to grow. The order of the different functions is designed to reduce computation as much as possible.

The functions *calcRecruits* and *fragtate* are simple multiplications and therefore will be skipped. The only notable aspect is that the alga's height is reduced by the combined length of all fragments. Also the amount value is not directly set, but instead increased by the calculated recruits, since these are usually values below 1 and are summed up during recruitment, for the whole species.

The interaction looks as follows:

```
interact()
{
  for(neighbours)
  {
    if(neighbourIsTooClose && neighbourIsAlive)
      if(neighbour == alga && neighbourIsBigger)
      {
        die();
      }
      else if(neighbour == coral && neighbourIsBigger)
      {
        if(neighbour->isBleached)
          surviveWithChange(neighbour->bleachGrade);
        else
          die();
      }
  }
}
```

Basically if any other entity logic is too close and stronger than the alga, it dies. The *growth* function is similar simple:

```
grow()
{
  if(notMaxHeight)
    height += growthRateHeight;

  if(notMaxRadius)
    radius += growthRateRadius;

  calcSize();
}
```

The *calcSize* function only updates values such as the ground area, based on geometrical functions.

The last function in Alga is the *instantAge* function, which enables it to grow a certain number of steps, without competition:

```
instantAge(steps)
{
  age += steps;
  limitToMaxHeight( height += growthRateHeight * steps);
  limitToMaxRadius( radius += growthRateRadius * steps);

  calcSize();
}
```

If the new age is over the max age, the alga dies in its next update.

CoralLogic

The *update* of CoralLogic is very similar:

```
update()
{
  recruits->amount = calcRecruits();

  age++;
  if(age > maxAge) //too old
    die();

  bleachWithChance();
  if(bleached)
  {
    if(random(deathProbability))  //a bleached coral can simply die
      die();
    else
      updateGrowthRate();     //bleaching affects the growth rate
      recover();
  }

  if(canFragtate && toBig)
  {
    recruits->fragments = fragtate();
  }

  interact();
  if(killedByInteraction)
    die();

  if(alone && breakable)
  {
    //didn't fragtate earlier
    if(didNotFragmentYet && canFragte)
      recruits->fragments = fragtate();

    die();
  }

  grow();
}
```

The main differences are the possibility to bleach and more possibilities to fragment, for example some corals can completely break if they stand alone. The bleach and death probability are simply calculated by checking how far up the current temperature is, compared to the respective ranges for bleach and death temperature.

The *grow* and *interact* functions are slightly more complex than in Alga:

```
grow()
{
  for(branches)
    branch->grow(growthRateHeight, growthRateRadius);

  radius = averageBranchLengthXY;
  heigh = averageBranchLengthZ;

  calcSize();
}

interact()
{
  totalIntersectionArea = 0.0;
  totalFoliageCoverage = 0.0;
    for(neighbours)
    {
      if(neighbour == alive && neighbourIsTooClose)
      {
        if(neighbour.contains(me))
          die();
        elseif(me.contains(neighbour))
          skip;

        totalIntersectionArea += calcIntersectionArea(me,
 neighbour);

        for(branches)
          branch->reduceGrowthRate();

      }
    }

  if(bleached)
    for(turfcells)
      totalFoliageCoverage += calcIntersectionArea(me, turfcell);

  if(totalIntersectionArea > 75% || totalFoliageCoverage > 75%)
    die();
}
```

In short a coral grows by growing its branches individually and height and radius are calculated based on the branches.

During interaction a coral dies only if it's overgrown by to many other entities. The intersection is calculated for two circles with the entities position and radius. This is extremely precise and not the cheapest function, considering the formula contains two inverse cosine and one square root, for a simple comparison. Therefore one approach was to interpret the ground area of both entities as perfectly aligned squares and reduce the intersection calculation to a simple linear interpolation. This approach however delivered vastly different values and was therefore discarded.

Branch

The branch class currently has no complex logic. It only moves the coordinates of the tip location according to the growth rate.

But it has an currently, unused *calcNewDirection* function, which would enable a branch to adjust its direction. The class also has a vector for bending positions. Both were introduced for a more complex growing algorithm, that could make interaction more precise and deliver valid data for visualization in the VR CoralReef, for a more accurate representation of the simulation state. However such an algorithm would have to be newly developed and is therefore not implemented.

4.3 World

4.3.1 WorldController

The WorldController is the core of each simulation, it initializes and updates the state of the simulated world. First a look at the *init* function:

```
1  init()
2  {
3    defineNumberOfThreads();
4    numberGridCellController = numberOfThreads();
5
6    generateTurfcells(turfcellsize);
7    gridCellSize = max(allSpecies->radius);
8
9    for(cells)
10     findAndAddCoveredTurfcell();
11
12   worldLogic.create();
13
14   for(species)
15   {
16     calculateCoveredArea();
17     initialRecruits = convert(coveredArea)
18   }
19
20   for(initialLogics = recruit(initialRecruits))
21   {
22     logic->instantAge(random(0-maxAge));
23     spawn(logic);
24   }
25
26   for(disturbanceDefinitions)
27     createDisturbance();
28 }
```

The most notable aspect in *init* is that the GridCells are created with the biggest radius of all species as their size. This obviously has some draw backs, as it increases the number of comparison for smaller entities, which is especially bad since the smaller ones are also greater in number for the same amount of percentage reef coverage, due to their size. So why not make the cell sizes smaller?

There was an approach to use smaller cells, but it required that the entities covering multiple cells, would be made known to those which they cover, otherwise they would be considered by the cells inhabitants, during interaction.

This introduced a huge amount of overhead for administrating the grid, as will be shown in chapter 5. The extra processing was so expensive that it took more time than anything else during the update process and was therefore discarded.

Another approach was to use different grids as layers for each species and then connect the cells between the grids. This had the advantage that each entities could precisely identify its neighbors and keep the comparison very low, it would also only need an initial setup rather than a regular update.
However at the initial setup it would require similar administration as the previous approach, but in this case the performance cost increased so dramatically that even a small 20x20m reef wouldn't initialize after 20 minutes.

A solution for this problem would require a grid structure than can work efficiently with varying sizes of its inhabitants or faster administration of the grid itself.

The other important function in WorldController is update:

```
update()
{
  updateTemperature();

  inParallel:
    for(GridCellControllers)
        updateGridCells();  //gridcells contain the entities

  disturbanceManager->update();
  removeExpiredDisturbances();


  for(entities)
  {
    collectRecruitData();
    collectDeadOnes();
  }

  inParallel:
    collectExternalRecruits();

  removeDeadEntities();

  collectRecruits(recruitData);
  spawnRecruits();

  calcRugosity();
  calcGrazingProbability();
}
```

Most of the called functions simply distribute work to other parts of the simulation. The order of the called functions is important to a certain degree. The key aspects are that, data gets removed only after it is processed, as to not loose any important data and also that the entities get updated at the beginning.

At last the temperature, rugosity and grazing probability should all be update either at first or last in the process so they affect all entities in one step in the same way. Rugosity is just a value determining of the surface roughness of a reef. The mathematical function for rugosity and grazing probability are taken from Sicom_java.

The collection of recruit data also contains summing up the fraction amounts of recruits for each species, since each entity creates recruit data, depending on its surface area, but a single entity does not produce enough recruit data for a whole new entity.

This unfortunately takes a noticeable amount of extra processing. It could maybe be solved by calculating the recruits based on the amount of entities and assuming an average surface area, it is however unclear if this would deliver the same results in a reliable way or if would be to much of an abstraction.

Another alternative could be that each entity creates enough recruit data for a whole new entity with a certain probability instead of calculating a fractional amount. This probability should of course also be dependent on the surface area, but it would still be an abstraction and first require to find a suitable probability function.

The last notable aspect of update is that some functions are run in parallel, while others are not. The decision for which to run in parallel, is simply based on if the functions writes only to the object it belongs to or not. The entities for example only read from other objects in their update process and can therefore parallelized without any synchronization. Other functions however access the same shared data, often being a map or a vector, so the processes can not be parallelized without synchronization.

Some of the processes can be written either way, for example a vector can be written to in parallel if the size and indexes are known. But in basically all processes that are currently sequential, the parallelization required either to much synchronization or memory management, that the results were the same or even slightly less efficient, than the sequential implementation.

4.3.2 EnvironmentDefinition

This class follows the same structure as EntityDefinition, the differences are only in the stored variables and that the Proxy-functions work on EnvironmentDefinition respectively, instead of EntityDefinition. For more explanation, please see chapter 4.2.1.

4.3.3 Temperature

The temperature class manages the temperature of the whole reef to provide a realistic distribution. It operates in a static manner since its has no direct link to any other simulation object and the only relevant function is *update*:

```
1  update()
2  {
3    selectMonthAndYear();
4
5    //also shifts the monthly data to consider
6    //temperature increase
7    calculateAverageTemperatureOverLongTermSummerMean();
8
9    apply yearly increase:
10     calcMonthlyMean();
11 }
```

The function can be summarized as calculating the mean over the monthly data, with the addition that a yearly increase can be integrated and that the monthly data considers the previous selected data sets. The algorithm is taken from Siccom_java.

This implementation requires a record of daily temperature data for the simulated location, over a few years.

It would be more flexible if the temperature could be represented by a generic mathematical function, that does not need so much pre recorded data, but the issue is that the temperature data does not follow a simple normal distribution and the temperature is dependent on the previous temperatures.

4.3.4 Turfcell

The Turfcell class represents patches of turf algae in the reef and has a very simple logic as follows:

```
1  update()
2  {
3    calculateAreaCoverdByCorals();
4
5    reduceGrowthRate(coveredArea);
6
7    coverage += growthRate;
8  }
```

The Turfcell simply grows according to a growth rate and that growth rate gets dampened depending on how much of the cell is covered by corals. The growth rate can be reduced to zero and the coverage itself is given as a percentage value.

4.4 GridCell

4.4.1 GridCellController and GridCellLogic

The two classes are used for encapsulation and an easier update process of entities. The *update* functions are as follows:

```
1  GridCellController:
2    update()
3    {
4      shuffleGridCells();
5      for(GridCells)
6        cell->update();
7    }
8
9  GridCellLogic:
10   update()
11   {
12     shuffleEntities();
13     for(entities)
14       entity->update();
15   }
```

The pseudo code shows that their only function is to update the underlying object in a random order. This randomness is quite important for a single or low multi threaded implementation, as it ensures that all entities are treated equally and no entity gets an advantage, because it gets always updated first and therefore is always stronger than its neighbors, for example.

These classes also make the management easier, the GridCellLogic enables the entities to only compare themselves with the inhabitants of the neighboring cells, as potential competition, which keeps the comparison low. This effect obviously would

benefit from smaller cells for a higher precision, when collecting neighbors, which in turn however requires more administration of the grid structure, which in the tested approach outweighed the performance gains, as explained in Chapter 4.3.1.

The GridCellController realizes an easy and even distribution of the GridCells to different threads, while keeping random aspect.

In case of a massively parallel environment, such as GPU multithreading, both classes could probably be ignored and the entities directly updated, since the high parallelization would introduce enough randomness for a realistic simulation. This however depends on the specific relation of entities and threads.

4.4.2 GridCellManager

The GridCellManager fulfills the role of adding entities to a specific grid cell and adding all neighboring cells to the entity. The collection of the neighbors, is the most important part, it happens only once when an entity is spawned and therefore does not require for the competition of and entity to be collected each update anew.

The neighbors will at least consist of the directly connecting cells even if the entity's radius does not reach beyond its own cell. This ensures that big neighboring entities reaching into the cell are considered during the update process.

The adding of a an entity looks as follows:

```
1  addEntity(entity)
2  {
3    gridCells[entity.xLocation][entitiy.yLocation]->addEntity(
     entity);
4
5    calcBoundaries(entity.maxRadius);
6
7    for(boundaries : cells)
8      entity->addNeighbor(cell);
9  }
```

4.5 Recruitment

RecruitmentManager

This class is currently just a pass-through to RecruitmentLogic and only extracts some values, such as world boundaries in doing so.

4.5.1 RecruitmentLogic

In RecruitmentLogic the RecruitData gets processed and formed into EntityLogics:

```
recruit()
{
  result = vector<EntityLogic>();
  recruitData.shuffle(); //ensures again that no species has an advantage over another

  for(recruitData)
  {
    for(recruitData->amount)
    {
      while(positionNotAvailable && tries < 5)
        position = randomPosition(worldBoundaries);

      if(positionNotAvailable && triesExpired)
        skip; //entity has no luck and does not get spawned
      else
      {
        entity = createEntityLogic(position, recruitData->entityDefinition);
        if(entityIsFragment)
          entity->setInitialSize(fragmentSize);

        result.insert(entity);
      }
    }
  }

  return result;
}
```

The process basically tries to find a free location in the world and if it finds one, it creates an EntityLogic at this location.

Theoretically this process could be very easily distributed over multiple threads, since the new entity logics do not depend on each other and only read from other objects. The issue here is that the result vector is shared among all new logics and can not be written to in parallel.

For the ability to fill the vector in parallel the vector would need to be pre initialized and the entities would need to know their target index, but this is not possible since each recruit data has a different amount of logics that will be created and therefore the indexes can not be reliably calculated.

RecruitData

RecruitData is a simple struct that contains an EntityDefinition, an amount of normal recruits and an amount of fragment recruits. The normal recruits are stored

as a floating point, since as mentioned in chapter 4.3.1, a single entity does not produce enough recruits to spawn a whole new entity.

4.6 Disturbance

Disturbance had no class in Siccom_java, instead it was realized as two hard coded functions that could be called.

This was changed so that a Disturbance can now be defined through its DisturbanceDefinition to better simulate different disturbance events. It also can be specified more precise.

Aside from the boundaries a DisturbanceDefinition also has information for the deadliness, a disturbance type which can currently be either be mechanical or chemical and can be specified to affect or ignore only certain species or phyla.

If these new variables are not set the disturbance will simply affect all entities in its boundaries, with its general deadliness. The deadliness is a probability factor between 0 and 1.
The Disturbance class is similar to Entity just representative and has no logic of its own.

4.6.1 DisturbanceDefinition

This class follows the same structure as EntityDefinition, the differences are only in the stored variables and that the Proxy-functions work on DisturbanceDefinition respectively, instead of EntityDefinition. For more explanation, please see chapter 4.2.1.

4.6.2 DisturbanceManager

The DisturbanceManager takes care to collect the affected cells for a disturbance and triggers it. The process:

```
1  update()
2  {
3    for(disturbances)
4    {
5      //ready is determined by the steps
6      if(disturbanceReady)
7      {
8        if(disturbance->randomize)
9        {
10         disturbance->position = randomPos();
11         disturbance->size = randomSize();
12       }
13
14       affected = gatherAffected();
15       disturbance->trigger(affected);
16
17       if(disturbance->expired)
18         expiredDisturbance.insert(disturbance);
19     }
20   }
21 }
```

```
22
23  gatherAffected()
24  {
25    affected = vector<gridcell>;
26    calcBoundaries(disturbance->shape);
27
28    for(boundaries)
29      affected.insert(cell);
30
31    return affected;
32  }
```

As seen in the update function, the manager also considers when it is time to trigger the disturbance and causes it to select a new position and size if that is to be desired. The expired disturbances are collected, but the actual removal happens in WorldController, since the manager only operates on the DisturbanceLogic objects and has no access to the representative disturbance objects.

4.6.3 DisturbanceLogic

The DisturbanceLogic simply kills all affected entities in its boundaries, based on a certain probability. Killing is simply triggering an entity to die, but with the possibility to act before the actual death, for example the entity could produce some fragments.

Aside from killing the entities the DisturbanceLogic also calculates when its next trigger step and reduce its counter for remaining triggers. When the counter reaches zero the disturbance has expired, however it can be set to -1 for the disturbance to never expire.
The process looks like this:

```
1   trigger()
2   {
3     for(affected->inhabitants)
4     {
5       if(inhabitant.isAlive && inBoundaries(inhabitant))
6       {
7         if(inhabitant.species == ignored)
8           skip;
9
10
11        if(hasCustomDeathProbability(inhabitant.phylum))
12          inhabitant->killWithChance(phylum.deathProbability)
13        elseif(disturbance->phylumSpecific)
14          skip;
15
16        if(hasCustomDeathProbability(inhabitant.species))
17          inhabitant->killWithChance(species.deathProbability)
18        elseif(disturbance->speciesSpecific)
19          skip;
20
21        inhabitant->killWithChance(disturbance->generalDeadliness);
22      }
23    }
24
25
```

```
26      //the damage depends on how much of the cell is affected
27      for(affected->turfcells)
28        if(inBoundaries(turfcell))
29          turfcell->reduceCoverage(calcDamage(distance));
30
31      nextStep += interval;
32      timeRemaining--;
33    }
```

In addition to killing entities, a disturbance also reduces coverage on turf cells. It should also be noted that even if a disturbance affects all entities, it can still have custom values for the deadliness of specific species or phyla, this way it can be represented if certain species are extra sensitive to an disturbing event.

4.7 RequestHandler

The RequestHandler serves as an access to the simulation for other programs or even just the GUI, therefore most of its functions are simply wrapper for processes that might be relevant to other parties, such as adding a new object or requesting information.

But there are four more important functions: init, startAutomated, step and stop.
The *init* function simply initializes all parts required at the start of the program, such as the proxy-functions for definitions or the config reader, therefore the logic is so simple, that it will be skipped.
The other three look as follows:

```
1   startAutomated()
2   {
3     if(worldSet = false)
4     {
5       createWorldController();
6       worldSet = true;
7     }
8
9     automated = true;
10    siccom->running = true;
11    InNewThread:
12      siccom->run();
13  }
14
15  step()
16  {
17    if(worldSet = false)
18    {
19      createWorldController();
20      worldSet = true;
21    }
22    else
23    {
24      siccom->step();
25    }
26  }
```

The function *startAutomated* and *step* functions are basically just wrapper for the *run* and *step* function in Siccom.

```
stop()
{
  if(automated)
  {
    siccom->running = false;
    automated = false;
    joinThread(siccom->run());
  }
  else
  {
    siccom->finish();
  }
  worldSet = false;
  siccom->steps = 0;
}
```

The stop functions stops a running simulation completely and resets all values to their starting values.

The important parts in these functions are the values worldSet and automated, these enable the simulation to switch between automated and step wise operation, whenever wanted and also to manage the stop function afterwards accordingly.

4.8 ConfigReader

The ConfigReader, as the name suggests, reads the definition values from external json files and creates the according objects. For this purpose it uses the boost::property_tree structure, that automatically turns a json or xml file into an iterable data structure. The reading that operates as follows:

```
readConfig()
{
  for(node : tree)
  {
    if(node.name == "Environment")
      parseEnvironmentDefinition(node.content);

    if(node.name == "Entities")
      for(node.contents) //contents are list of file names
        parseEntityDefinition(content.name);

    if(node.name == "Disturbances")
      for(node.contents) //contents are list of file names
        parseDisturbanceDefinition(content.name);
  }
}

//all parseDefinition function behave the same
parseDefinition(definitionName)
{
  definitionTree = readFile(definitionName);
  definition = getConstructor(definitionName);
  functions = getFunctions(definition);
  for(node : definitionTree)
  {
    try{
      setter = functions(node.name).setter;
      value = createGetSetValue(node.content, setter.dataType);
      call setter(definition, value);
    }
    catch(setterNotFound)
    {
      print("No function found for node.name");
    }
  }
}
```

Basically the reader compares the names from the configuration file to the keys for the proxy functions and if it finds a suitable function it gets called. The value needs to be parsed from a string to a GetSetValue, for this purpose the functions are saved together with an enum value determining which data type they use. This process can of course be reversed in the same way by iterating over the functions map and calling the getter functions to create a node for the tree, which than may be written to a file.

Chapter 5

Performance Comparison

In this chapter the performance of the new Siccom++ and Siccom_java in terms of execution time, memory usage and complexity levels will be shown. Additionally the Siccom++ performance will be further analyzed to gain insight on which parts require the most processing time and would need the most optimization for further performance improvements. The data was gathered through internal time measurement and logging. The memory data was only collected by monitoring the processes memory at runtime and only peak values were collected. The data will be interpreted in Chapter 6.
All configurations were run on a computer with the following specifications:
Intel Core i7 7700K 4 cores(8 threads) @4.20 GHz
16 GB DDR4 Ram dual channel @2400MHz
500GB SSD, 545MB/s read, 525MB/s write

The configuration were all square sized reefs with side length of 25, 50, 100, 150, 200, 300 and 500 meters. The 500m variant was only run in Siccom++ as in Siccom_java the 300m variant already required so many resources that the 500m variant would have most likely surpassed the limits of the computer, without giving any new significant information. All multi threading configurations used 8 threads.

5.1 Siccom++ vs Siccom_java

This section will compare the performance of Siccom++ with the performance of Siccom_java.

The total execution time for all configurations:

Figure 5.1 shows that the execution time for Siccom_java grows faster and is in general longer than for Siccom++, even though it should be noted that on smaller configurations, Siccom_java performed faster. Additionally Siccom++ performed consistently better with multiple threads.

Configuration	Siccom_java	Siccom++ single thread	Siccom++ 8 threads
$625m^2$	2 sec	11 sec	7 sec
$2500m^2$	20 sec	1 min 9 sec	38 sec
$10000m^2$	4 min 59 sec	6 min 58 sec	3 min 49 sec
$22500m^2$	26 min 34 sec	18 min 28 sec	9 min 56 sec
$40000m^2$	1 h 45 min 3 sec	33 min 55 sec	18 min 10sec
$90000m^2$	9 h 57 min 54 sec	1 h 21 min 22 sec	44 min 2 sec
$250000m^2$	n/a	4 h 29 min 52 sec	2 h 26 min 42 sec

Table 5.1: The total execution for each configuration in the different implementations. Given as their exact values. Siccom_java is missing the $250000m^2$ configurations, since no tests were done for that specific setup.

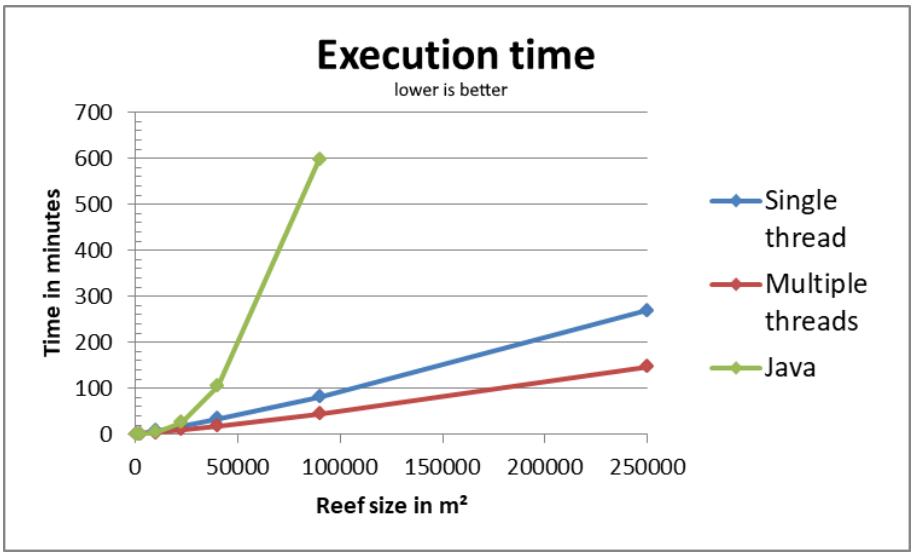

Figure 5.1: The total execution time for all configurations, given in minutes. The connections between the points, don't represent actual results and exist only for a better visualization of the data.

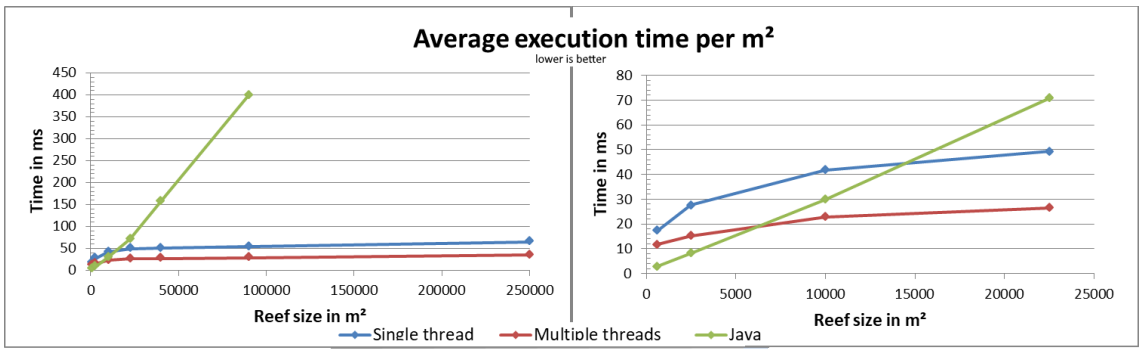

Figure 5.2: The execution time per m^2, given in ms. On the left an overview of all configurations, on the right a close up of the three smallest configurations.The connections between the points, don't represent actual results and exist only for a better visualization of the data.

For a better visualization of execution time growth and actual performance Figure 5.2 shows the execution time per square meter and also highlights the smaller configurations. It shows that Siccom_java becomes at a reef size of about $5625m^2$ slower than the multi threaded Siccom++ and at about $14400m^2$ slower than the

single threaded Siccom++.

The data shows that the execution time for Siccom_java follows a complexity of $O(n^2)$, while for Siccom++ it follows $O(n * log(n))$. However when viewed in relation to reef size, complexity for Siccom++ follows $O(n)$, while the complexity of Siccom_java is still $O(n^2)$.

Figure 5.3 shows the percentile increase in execution speed for Siccom++ in relation to Siccom_java. It shows that for smaller reef sizes Siccom++ was actually slower and that the performance gain increases in relation to reef size.

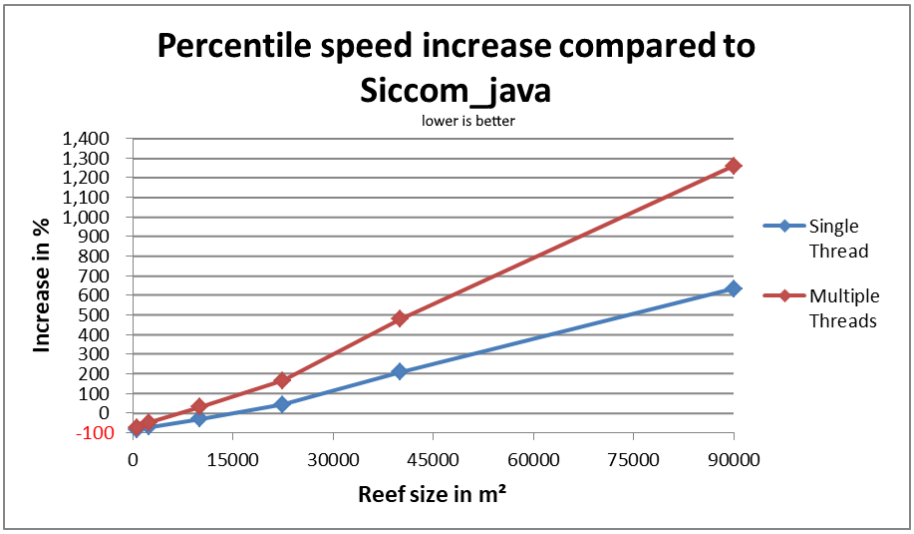

Figure 5.3: The percentile increase in execution speed, compared to Siccom_java.The connections between the points, don't represent actual results and exist only for a better visualization of the data.

The memory usage shows a slightly different relation, where Siccom_java consistently has higher requirements. For memory usage Siccom++ was not differentiated between single and multiple threads because the number of threads has no relevant impact on memory usage.

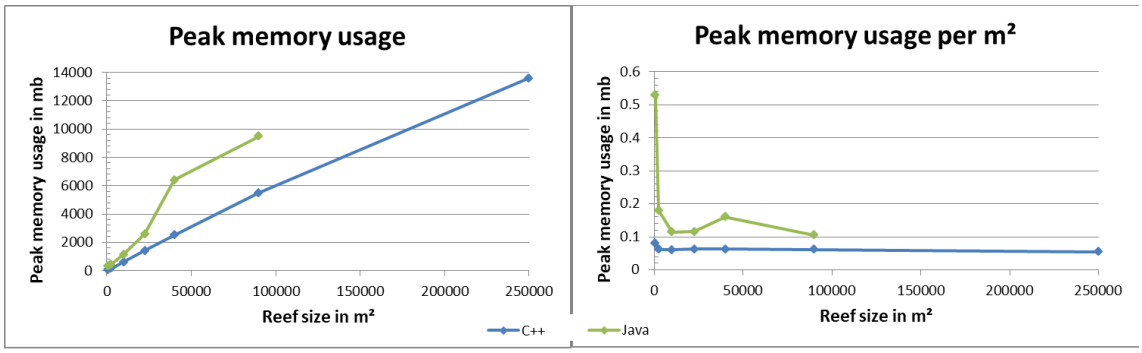

Figure 5.4: The peak memory usage of both implementations, given in MB. On the left is the total memory usage, while on the right is the memory usage in relation to reef size.The connections between the points, don't represent actual results and exist only for a better visualization of the data.

Figure 5.4 shows the peak memory usage of both implementations in total and per square meter. It is visible that the memory usage per square meter is far higher

Configuration	Siccom_java[MB]	Siccom++[MB]
$625m^2$	330	50
$2500m^2$	450	155
$10000m^2$	1150	600
$22500m^2$	2600	1400
$40000m^2$	6400	2500
$90000m^2$	10400	5500
$250000m^2$	n/a	13600

Table 5.2: The memory usage of Siccom_java and Siccom++, for each configuration, given in MB. Again Siccom_java has no values for the configuration $250000m^2$, due to missing tests.

for the smallest configuration, than any other value. For additional information the exact values are shown in the following table:

The memory consumption for both implementations follows a complexity of $O(n)$, while the memory usage per square meter is mostly $O(1)$, with the exception of Siccom_java having a higher minimum consumption for the smallest configuration.

5.2 Single-thread vs multi-threading

Figure 5.1, 5.2 and 5.3 have already shown that the multi-threaded Siccom++ performs better than the single-threaded. Here will be shown how exactly it affects the simulation.

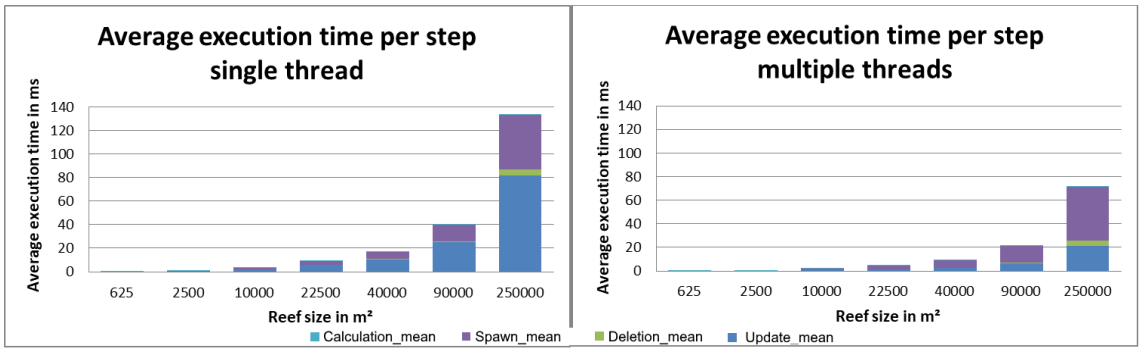

Figure 5.5: The execution time per step in ms, for each configuration. It also shows the different parts of a single step and their execution times. Light blue shows the calculation mean, purple the spawn mean, green the deletion mean and dark blue the update mean. Calculation refers to global calculations, such as the rugosity value. Update refers to the explicit updating of entities.

Figure 5.5 shows the total average execution time per time step and also displays the share of each part of the update process. It becomes quite clear that the multi threading not only shortens the execution time, but also that the spawning part takes an increasingly bigger share, with increasing reef size.

For a clearer visualization Figure 5.6 shows the shareholders of the update process summed up to 100%.

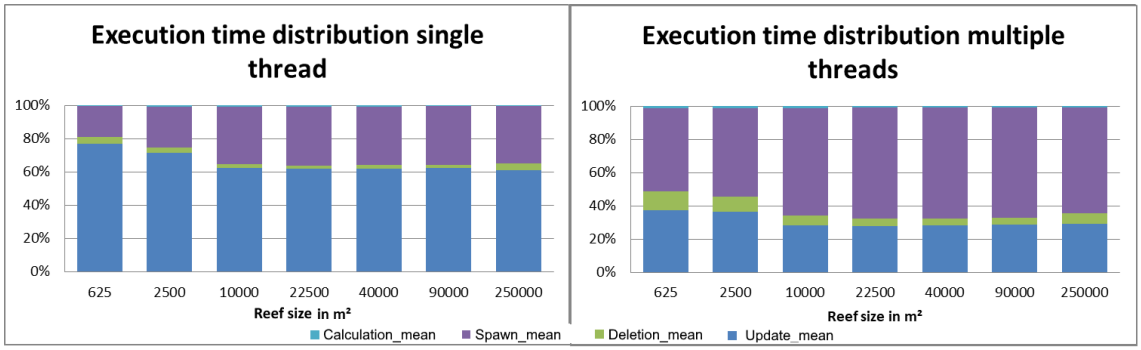

Figure 5.6: The relative share of each component of the update process per time step. Light blue shows the calculation mean, purple the spawn mean, green the deletion mean and dark blue the update mean.

Configuration	Total(ms)	Update(ms)	Spawning(ms)	Average comparisons
$625m^2$_big	17.6	15.6	1	190.86
$625m^2$_small	8.7	6.2	1.1	5.42
$2500m^2$_big	84	73.2	6	225.28
$2500m^2$_small	47.6	33.2	7	5.47
$10000m^2$_big	397.9	322.8	49.6	139.64
$10000m^2$_small	228	157.3	35.0	5.80

Table 5.3: The execution times for different reef sizes, with two different cell sizes. The suffix determines if the cell sizes were big or small. It also shows the average number of comparisons between entities.

5.3 Gridperformance

As already mentioned in Chapter 4.3.1 the simulation can benefit from smaller cell sizes. To highlight the impact of the cell sizes on execution time the smallest three configuration where run an additional time with a slight modification. In the new configuration the two massive coral species with an maximum radius of 300cm were removed and the initial coverage was replaced by the two branching corals with a maximum radius of 50cm and 30cm. This effectively reduced the cell size from $90000cm^2$ to $2500cm^2$. All configurations were run in single thread.

The results are shown in the following table, big and small refer to the cell sizes. For better readability only biggest contributors to execution time were included:

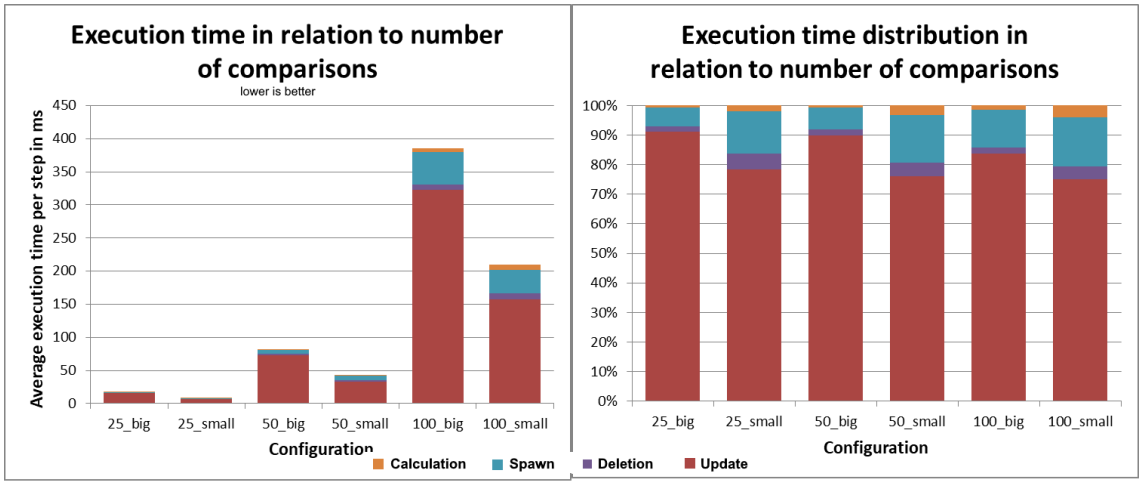

Figure 5.7: Comparison between the execution time of different cell sizes. On the left the absolute values and on the right the values summed up to 100%, showing a little more effect on the entity update part, compared to the other elements of the updateprocess.

It becomes clear that the smaller cell sizes reduced the number of average comparisons between entities and greatly reduced the execution time. It is also visible that the number of comparisons not only went don't but was also more consistent. The time data is visualized in Figure 5.7, showing also that the biggest improvements were found in the updating of entities.

Chapter 6

Evaluation

This chapter will evaluate the results of chapter 5 and also how much more flexible Siccom++ is compared to Siccom_java.

6.1 Performance

The results from chapter 5 show that Siccom++ performs better in most cases than Siccom_java. The exception to this are the smaller configurations.

One difference in Siccom++ is the introduction of recruit data. The recruit data however also added an extra amount of processing and new recruits are first created and then collectively spawned. In Siccom_java the spawning happens simply by scheduling the new entities through the MASON framework, in addition the entities are never explicitly removed, instead they simply stop re-scheduling themselves. Further profiling of Siccom_java, with the usage of VisualVM revealed that 97% of the time is spend for the updating of entities and turf cells, whereas the in Siccom++ its just about two thirds with the last third being mostly the spawning of new entities. This new introduced management and processing is probably the reason why Siccom++ performs worse at lower reef sizes, since at those sizes the gains from the grid structure can't outweigh the newly introduced processing, this changes as the reef size grows and an efficient storage structure becomes more relevant.

The memory consumption seems mostly influenced by the entities, which is why it is constant per square meter, however the total memory usage is higher in Siccom_java. The reason for this is most likely that the entity classes (Alga, BranchingCoral, BranchingGroup etc.) have many variables stored for each entity and may even store variables that are only temporarily required. Siccom++ on the other hand has most variables stored in the Definition classes which are shared, therefore each entity requires less memory than in Siccom_java. The higher memory consumption of Siccom_java at the 25x25m reef size is most likely a result of the Java VM, requiring a minimal amount of memory.

All in all Siccom++ can be considered to be performing better and the performance goal to be reached. While the smaller reefs are a little bit slower, but also more interesting to the VR CoralReef, they also require significant less memory and should still be fast enough for the VR CoralReef, therefore the gain in memory efficiency probably outweighs loss in speed.

For further performance gain the grid structure and the spawning should be the main parts to be considered for optimization. As shown in chapter 5, smaller grid

cells greatly improve the performance, so the goal should be grid that can work efficiently with entities being bigger than the cells. Potential solutions might be a grid with different layers, each with different cell sizes connected by an octree or simply a grid that works with varying cell sizes. The spawning would also benefit from a more efficient grid, as it access the grid to search for suitable locations. The other aspect that could be improved would be the calculation of new recruits, as already explained in chapter 4.3.1, which might also make it possible to properly distribute the recruitment over multiple threads, gaining additional performance.

6.2 Usability and extendability

The disturbances can now be specified to a greater degree in their own files and are also not anymore bound to two different hard coded functions. They also do not need to explicitly interact with any specific implementation of Entity, as they work on Entity itself.

The entities are now unified in a single abstract class and adding a new type requires only that a simple class structure is followed an a single line is added into the init function of EntityDefinition. While it may require the implementation of extra getter and setter functions, it is easy to follow and it's only required if the new type has any unique variables that need to be considered. In Siccom_java the adding of a new entity type would require for the newly implemented class to be explicitly called at initialization, in the disturbance functions and any calculation that needs to consider this type. In Siccom++ the new class needs only be explicitly considered if other entities interact with unique features of the new type or in a specific way. This interaction can also be further controlled with the Phylum variable, which can remove the requirement to explicitly add the new type, if its Phylum already exists. Aside from the Entity class, the differentiation between the logic and definition classes, makes it clear where which kind of values and functions can be found and written. Entities can be defined in mostly the same way from external files, the file format however was changed to json, which is widely supported and could enable an easier interpretation in other software, if that would to be desired.

Siccom++ also follows a clear defined architecture style that makes it easier to navigate the code base when searching for specific elements, compared to the structure of Siccom_java which didn't seem to follow any specific architecture or structure.

Considering all these advantages the extendability and usability has definitely been increased and the goal of this thesis was reached.

Chapter 7

Summary

The goal of this thesis was to to reimplement the coral reef simulation software Siccom.

The goals of this new implementation were to create a program with the same simulation capabilities that is easier to access from other software and easier to extend with new features, all without sacrificing realism to keep the scientific integrity. Another goal was to at least stay on the same level of performance as the original, while trying to improve it.

To reach these goals, three core elements were used. These elements consist of the Domain Driven Design, software architectural style, certain data structures and parallelization. The architecture style was the key component for the extendability and usability, since it enabled a clear and modular structure. The data structures supported both the performance and the extendability.

The extendability was supported by static function pointers stored in a map, which enabled reading and setting values from certain classes without an explicit call to any of the classes functions.
The grid data structure reduced the number of comparisons between objects and was crucial for the performance.
The parallelization directly improved the update process of certain objects.

With these elements all goals were achieved. The extendibility and usability has clearly improved, as elaborated in chapter 6.2.

While the performance has suffered in execution time for areas smaller than 75x75m^2 to 100x100m^2 (depending on the number of threads), it can also be considered to have reached its goals, when considering all aspects, especially since other aspects greatly improved, also explained in chapter 6.1. More specifically the growth of execution time per m^2 went from $O(n^2)$ to $O(n)$. The complexity of the memory usage growth function did not change, however the total memory usage went down by up to 50% in some cases. These improvements made it possible to simulate a 500x500m^2 reef in about only 25% of the time, Siccom_java needed for a 300x300m^2 reef.

All in all the re implementation of Siccom into Siccom++ can be considered a success.

Chapter 8

Future work

Possible future work includes further optimization of the grid structure and spawning logic as explained in chapter 6.

Another slight performance improvement could be implemented by providing access to the simulated entities in the form of object streams instead of vectors and maps, this could be especially helpful if the software is supposed to run on a server and distribute the data over a network connect, for example for the VR CoralReef multiplayer mode.

Beyond simple performance improvements possible future work could include, increasing the complexity of turf cells to sediment cells. This would enable to reef ground to take the form of different sediment types, which would not only add another layer of realism but could also be used to increase the visual fidelity in the VR CoralReef.

Another feature that could be implemented would be a more complex growth algorithm for corals, that simulates the exact shape of the coral species. While this could slightly increase the realism, it would be most interesting for the VR CoralReef, to enable a closer connection between the procedurally generated coral models and the actual simulated reef.

Bibliography

[1] A. Kubicek, C. Muhando, and H. Reuter, "Simulations of long-term community dynamics in coral reefs - how perturbations shape trajectories," *PLOS Computational Biology*, vol. 8, no. 11, pp. 1–16, Nov. 2012. DOI: 10.1371/journal.pcbi.1002791.

[2] A. Sander, "Virtuelles korallenriff: Ein framework zum prozeduralen wachstum von korallen auf basis von l-systemen und isoflächen," Master Thesis, Computergraphics department, University of bremen; Ecomodeling, ZMT Bremen, Sep. 8, 2015.

[3] S. Luke, G. C. Balan, K. Sullivan, and L. Panait. (). Mason, [Online]. Available: https://cs.gmu.edu/~eclab/projects/mason/.

[4] (). Flame, Software Engineering Group, Scientific Computing, STFC, [Online]. Available: http://flame.ac.uk/.

[5] T. Brandt and S. Heitmann, "Vr coralreef: Reimplementation and enhancement of thecoral reef simulation siccom into the agent-basedmodeling framework flame," Computergraphics Department, University of Bremen, paper, 2017.

[6] M. D. Spalding, C. Ravilious, and E. P. Green, *World Atlas of Coral Reefs*. UNEP World Conservation Monitoring Centre, University of California Press, Berkeley, USA, 2001.

[7] R. Axelrod, "Advancing the art of simulation in the social sciences," University of Michigan, Article, May 17, 2005.

[8] J. D. Farmer and D. Foley, "The economy needs agent-based modelling," *nature*, Aug. 5, 2009.

[9] S. Wolfram, *Cellular Automata and complexity*. Mar. 8, 2018, ISBN: 9780429494093. DOI: https://doi.org/10.1201/9780429494093.

[10] ——, "Universality and complexity in cellular automata," The Institute for Advanced Study, Princeton NJ 08540, USA, Paper, Aug. 19, 2002.

[11] E. Bonabeau, "Agent-based modeling: Methods and techniques for simulating human systems," National Academy of Sciences, Article, May 14, 2002.

[12] B. Breckling, U. Middelhoff, and H. Reuter, "Individual-based models as tools for ecological theory and application: Understanding the emergence of organisational properties in ecological systems," *Ecological Modelling*, vol. 194, pp. 102–113, Mar. 2006. DOI: 10.1016/j.ecolmodel.2005.10.005.

[13] B. Breckling, F. Müller, H. Reuter, F. Hölker, and O. Fränzle, "Emergent properties in individual-based ecological models - introducing case studies in an ecosystem research context," *Ecological Modelling*, vol. 186, pp. 376–388, Sep. 2005. DOI: 10.1016/j.ecolmodel.2005.02.008.

[14] G. V. Bobashev, D. M. Goedecke, Feng Yu, and J. M. Epstein, "A hybrid epidemic model: Combining the advantages of agent-based and equation-based approaches," in *2007 Winter Simulation Conference*, Dec. 2007, pp. 1532–1537. DOI: 10.1109/WSC.2007.4419767.

[15] A. Kubicek, F. Jopp, B. Breckling, C. Lange, and H. Reuter, "Context-oriented model validation of individual-based models in ecology: A hierarchically structured approach to validate qualitative, compositional and quantitative characteristics," *Ecological Complexity*, vol. 22, pp. 178–191, 2015, ISSN: 1476-945X. DOI: https://doi.org/10.1016/j.ecocom.2015.03.005.

[16] A. Sander, "Virtuelles korallenriff: Ein framework zum prozeduralen wachstum von korallen auf basis von l-systemen und isoflächen," master thesis, Computer graphics and virtual Reality department, University of Bremen, Germany; ZMT Bremen.

[17] M. Pratchett, K. Anderson, M. Hoogenboom, E. Widman, A. Baird, J. Pandolfi, P. Edmunds, and J. Lough, "Spatial, temporal and taxonomic variation in coral growth—implications for the structure and function of coral reef ecosystems," *Oceanography and marine biology*, vol. 53, pp. 215–295, Aug. 2015. DOI: 10.1201/b18733-7.

[18] J. Melbourne-Thomas, "Decision support tools for visualising coral reef futures at regional scales," Ph.D. University of Tasmania, Sep. 2010.

[19] P. Mumby, A. Hastings, and H. Edwards, "Thresholds and the resilience of caribbean coral reefs," *Nature*, vol. 450, pp. 98–101, Dec. 2007. DOI: 10.1038/nature06252.

[20] P. Munday, J. Leis, J. Lough, C. Paris, M. Kingsford, M. Berumen, and J. Lambrechts, "Climate change and coral reef connectivity," *Coral Reefs*, vol. 28, pp. 379–395, Jun. 2009. DOI: 10.1007/s00338-008-0461-9.

[21] R. R. Graus and I. G. Macintye, "Light control of growth form in colonial reef corals: Computer simulation," *Science*, vol. 193, no. 4256, pp. 895–897, 1976, ISSN: 0036-8075. DOI: 10.1126/science.193.4256.895.

[22] H. Bossher and W. Schlager, "Computer simulation of reef growth," *Sedimentology*, vol. 39, no. 3, pp. 503–512, 1992. DOI: 10.1111/j.1365-3091.1992.tb02130.x.

[23] T. McClanahan, "A coral reef ecosystem-fisheries model: Impacts of fishing intensity and catch selection on reef structure and processes," *Ecological Modelling*, vol. 80, no. 1, pp. 1–19, 1995, ISSN: 0304-3800. DOI: https://doi.org/10.1016/0304-3800(94)00042-G.

[24] S. Saila, V. Kocic, and J. McManus, "Modelling the effects of destructive fishing practices on tropical coral reefs," English, *Marine Ecology - Progress Series*, vol. 94, no. 1, pp. 51–60, Dec. 1993, ISSN: 0171-8630.

[25] O. Langmead and C. Sheppard, "Coral reef community dynamics and disturbance: A simulation model," *Ecological Modelling*, vol. 175, no. 3, pp. 271–290, 2004, ISSN: 0304-3800. DOI: https://doi.org/10.1016/j.ecolmodel.2003.10.019.

[26] H. Yamano and M. Tamura, "Detection limits of coral reef bleaching by satellite remote sensing: Simulation and data analysis," *Remote Sensing of Environment*, vol. 90, no. 1, pp. 86–103, 2004, ISSN: 0034-4257. DOI: https://doi.org/10.1016/j.rse.2003.12.005.

[27] C. M. Kunkel, R. W. Hallberg, and M. Oppenheimer, "Coral reefs reduce tsunami impact in model simulations," *Geophysical Research Letters*, vol. 33, no. 23, 2006. DOI: 10.1029/2006GL027892.

[28] P. Hofman, "Pplication of a fish swarm model to analyze effects of self-organization," Bachelor thesis, University Bremen, Biology Department, 2013.

[29] M.Kruse and H.Reuter, "Landscape connectivity revisited – a unifying conceptual framework for the marine and terrestrial realm," unpublished, at point of writing, 2019.

[30] E. Evans, *Domain-Driven Design - Tackling Complexity in the heart of Software*. Jun. 2011, ISBN: 0-321-12521-5.

[31] V. Vernon, *Implementing Domain-Driven Design*. Jul. 2013, ISBN: 0-321-83457-7.

[32] J. Deacon, "Model-view-controller(mvc) architecture," John Deacon Computer Systems Development, Consulting & Training, Paper, May 2009.

[33] N. N. O. Service. (Jan. 26, 2005). Coral tutorial, [Online]. Available: https://oceanservice.noaa.gov/education/tutorial_corals/welcome.html.