UPPSALA
UNIVERSITET

IT 13 077
Examensarbete 15 hp
November 2013

Snapshot Algorithm Animation with Erlang

Fredrik Bryntesson

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Snapshot Algorithm Animation with Erlang

Fredrik Bryntesson

Algorithms used in distributed systems for synchronization can often be hard to understand, and especially for beginners these concepts can be difficult to apprehend. Seeing an animation of these concepts could help to gain insight about how they work. The Snapshot algorithm (Chandy-Lamport) is one of these. But what is a good animation of an algorithm? What characteristics do an animation need to be considered as good?

This thesis describes an analysis of those characteristics and a development of an animation software for the Snapshot algorithm using a game engine written in Erlang.

Handledare: Justin Pearson
Ämnesgranskare: Arnold Pears
Examinator: Olle Gällmo
IT 13 077
Tryckt av: Reprocentralen ITC

Contents

Chapter 1

Introduction

This report presents ideas on how to develop a simulation program for the Snapshot algorithm (Chandy-Lamport)[3] that besides simulation of the algorithm also presents the algorithm in a graphical interface and shows every step of it, and gives explanations about what is happening in the algorithm as it executes. For this report, ways to implement this using Erlang and third-party libraries for Erlang have been researched.

1.1 Background

The Snapshot algorithm is an algorithm used for synchronization of distributed systems, though it is a rather simple algorithm many people find it hard to understand and also to explain with an easy and pedagogical method. It can be difficult to understand exactly what Snapshot does and to make a personal visualization of it. Therefore, an application to give a visual step by step simulation for the Snapshot algorithm, to ease the level of abstraction for teaching this algorithm and other synchronization algorithms, can be of great value for computer science students.

1.2 Problem Definition

While creating this application, a few problems must be addressed. An application that simulates some different scenarios of the algorithm needs to be created, then a graphical interface to support all the steps of the simulation and visualize them needs to be implemented. Lastly, it is important to establish if this visualization is pedagogical enough to create an understanding of how the Snapshot algorithm works.

Chapter 2

Snapshot

The Snapshot algorithm is also called the Chandy-Lamport algorithm after Leslie Lamport and K. Mani Chandy. They presented the algorithm in their paper Distributed Snapshots: Determining Global States of Distributed Systems [3].

2.0.1 What is a Distributed System?

When a number of computers are connected in a network and communicate through message passing, it is called a distributed system. One important characteristic is the lack of global clock for the whole system. Another characteristic is concurrent execution. When the members of the distributed system share resources in this network the system needs to synchronize time to know in which order the actions are taking place. So occasionally the system needs to synchronize the system and calculate a global state by analyzing all the local states in every connected node. All the processes need to save their state at virtually the same time to create a valid global state. But this is hard as the network do not have a global clock. If it had a global clock, it could establish that all processes save their states at a specified time. But now an algorithm of how all processes should save their state is needed. [4]

2.1 The Algorithm

The algorithm is divided into two functions, or rules, the Marker Sending Rule, and the Marker Receiving Rule.

2.1.1 The Marker Sending Rule

Algorithm 1: The Marker Sending Rule for Process m

1 m records its local state;
2 **foreach** *outgoing channel C in which a marker message has not been sent* **do** Send a marker message on C before sending further messages on channel;

2.1.2 The Marker Receiving Rule

Algorithm 2: The Marker Receiving Rule for Process n

```
1 When receiving a marker on channel C;
2 if n has not recorded its state then
3 |   Record the state of channel C as the empty list;
4 |   Do the Marker Sending Rule;
5 else
6 |   Record the state of channel C as a set of messages that was received on
  |   channel C after n recorded its local state and before n received the marker
  |   message on channel C;
7 end
```

2.1.3 The Initializer

When a Node begins a session of the algorithm and want to create a snapshot record it begins by recording its local state, and then sending a marker message to all connected outgoing channels [3, 12].

2.1.4 The Receivers

When the node receives a marker message it immediately records its local state, and the state of the channel that sent the marker message to it is recorded as empty. Then, it sends the marker message to all their connected outgoing channels and sends the local state to the initializer. If the node receives a new message which is not a marker message after it has recorded its state, it forwards this message to the process that initiated the snapshot algorithm. Because this message was sent before the Snapshot algorithm was started, and is not included in any saved local state. After local state is recorded a node can receive a marker message on a channel after it has recorded the local state, then, it records the state of that channel as containing all messages it has received since local state was recorded. [3] [12]

2.1.5 Assumptions

For the algorithm to work the following assumptions must be made:

- There is no failure in the message passing, and all messages arrive intact and they only arrive once.
- The channels for the communication are unidirectional and FIFO-ordered to prevent messages sent after the marker to be received before the marker.
- Every node in the system must have a communication path between them.
- Any node can initiate the algorithm.
- The algorithm can not interfere with the other tasks of the nodes.

- Each node record its state and the state of the incoming channels.

These assumptions are necessary for several reasons and they are all important to create a complete Snapshot of the system. It must be assured that all messages are delivered, delivered intact and only delivered once. If this is not the case then the whole algorithm fails. Even if only one marker message is dropped the whole algorithm fails because then it has not recorded the state of all channels and the Snapshot of the system is not complete. It must be FIFO-ordered to prevent that messages sent after the marker arrives before the marker, and if the channel is not unidirectional it is necessary to keep track of messages sent both ways.

It is important that all the nodes are connected in a network and that every node can imitate the algorithm and request a global Snapshot, if one node stands alone outside the network it is not possible to communicate with that one, and it can not take part in the process to create a global Snapshot of the system. When the algorithm is executing, it can not interfere with the nodes other tasks, because it should be independent and not stall the system. [3]

2.1.6 State

A global state is the state of all processes in the system and all the channels that connect them to each other. To create a global state all processes must record their state and the state of all their channels. A record of the process state and the state of the channels is called a local state or a local snapshot. [3]

2.1.7 Marker Message

A marker message is simply a message which contains a message to start the algorithm when received. When a process receives a marker message it immediately starts to follow the algorithm. [3]

2.1.8 Channel

When two nodes have a communication link between them; there is a channel connecting them. A channel from node A to node B means that node A can send messages to Node B. A channel between node B and Node A means that B can send a message to A. The state of a channel is the set of all the messages sent on that channel which has left the sender and is on the way to the receiver, but has not been received by the receiver. [3]

2.2 Example Network

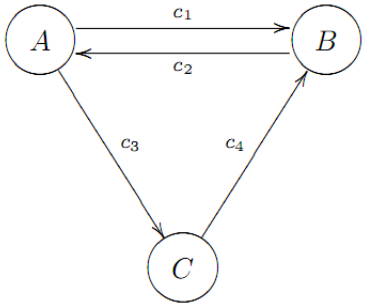

Figure 2.1:

A network containing three nodes: A, B and C. A has outgoing channels to B and C, B has an outgoing channel to A, and C has an outgoing channel to B.

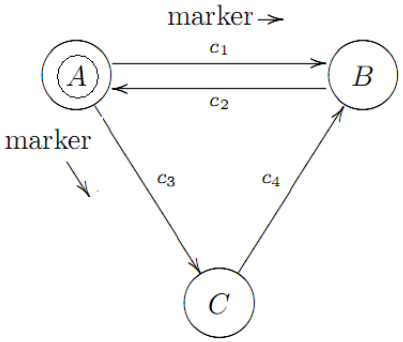

Figure 2.2:

Node A initiates the Snapshot algorithm and performs the Marker Sending Rule, it sends a marker to node B on channel c1 and to Node C on channel c3.

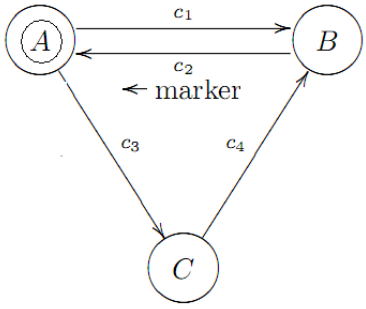

Figure 2.3:

Node B receives the marker from node A and immediately starts the Marker Receiving Rule and this is the first time it receives a marker message on the incoming channel c1. Therefore, it records its local state and the state of channel c3. It continues by performing the Marker Sending Rule and sends a marker message to all its outgoing channels, in this case it is channel c2, so Node B sends a marker message to Node A.

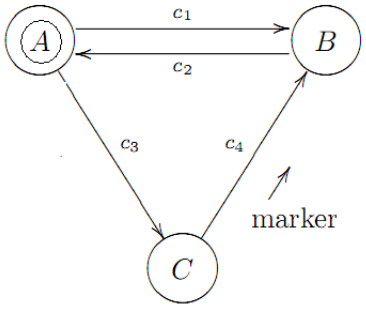

Figure 2.4:

Node C receives the marker from node A and performs the Marker Receiving Rule. Then, it records its local state and the state of channel c3, as this is the first time it has received a marker message on channel c3. Node C performs the Marker Sending Rule and sends a marker on all its outgoing channels; in this case only channel c4 to Node B. It has also received a marker message on all its incoming channels (as channel c3 was its only incoming channel) and can now terminate the Snapshot algorithm.

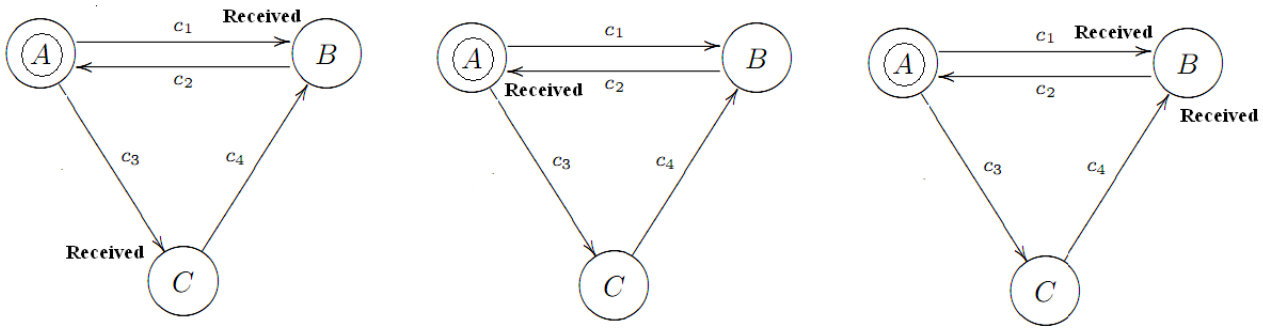

Figure 2.5:

Node A receives the marker on its incoming channel c2 and it is now finished because it has received a marker message on all its incoming channels and can terminate the Snapshot algorithm. The same thing happens to node B which receives a marker message on its incoming channel c4, and thereby it has received a marker on all its incoming channels. That terminates the Snapshot algorithm.

Chapter 3

Visualization

3.1 Algorithm Visualization as an Educational Tool

The first part of this project was to research if there was some method to visualize an algorithm like Snapshot that was superior to all other visualization methods and gave the user a larger sense of understanding the visualization than other methods did. Did it exist any research about visualization in computer science education and could this research provide some good ideas about the visualization design and explain the important characteristics of making an algorithm visualization.

3.1.1 Algorithm Visualization and Teaching

Ari Korhonen did research about visualization of algorithms in his doctoral thesis "Visual Algorithm Simulation" [10]. He describes methods of providing visualization to make abstracts concepts like complex algorithms and data structures more concrete and easier to understand. Korhonen have developed the Matrix application framework that does two things; first it visualizes fundamental computer algorithms, this is algorithm animation, and secondly, it provides a framework for exploring and understanding data structures and algorithms. He presents two different methods of visualization:

1. Static algorithm visualization - A set of images of each the steps of an algorithm execution is displayed.
2. Dynamic algorithm visualization - This is also called algorithm animation, a more living way of displaying the steps of an algorithm execution. This can be implemented in several different ways.

The key question as Korhonen phrases it is: how can methods like algorithm animation and algorithm simulation help students understand complex concepts in computer science? A major problem of this is the difficulty to capture the dynamic nature of algorithms and data structures in static teaching material such as books, lecture notes and presentation slides. A better method for teaching algorithms and data structures would be tools that support custom input data, multiple views and levels of abstraction and control, and history for execution. The easiest method to exploit these characteristics is to show a set of images that displays an execution of an algorithm, but this is still a

static method and a dynamic method would be less abstract.

Korhonen says that a dynamic method is preferred, this is because inherent parallelism of the human visual system and the importance of visualization as method of understanding a problem. So to see a visualization of an abstract problem make it easier to understand it, because a person has more inherent tools for seeing the algorithm compared to when reading a description of a problem. An advantage of making a graphical representation of an algorithm is that structural connectivity and symmetry is pointed out. It will also display concepts as links, flows, relations and directions between objects, and for most people, that is probably easier to understand than an abstract description. Korhonen presents two viewpoints of the educational purpose of algorithm animation and simulation; how the teacher can present algorithms with these tools, and how the students can work for themselves to learn and obtain a deeper understanding of the complex concepts. He states that from a pedagogical point of view it is important how the student sees the visualization, it can be fancy with many interesting special effects, but some form of interaction is necessary for the students to fully understand the algorithm. Algorithm animations are most educationally effective when they are supported by explanations in either text or audio. But they are most effective when the student can use the animation by themselves with their own code and algorithms to create their own understanding of the abstract concept through active learning. [10, 1, 8]

This application Matrix can be used to simulate an actual algorithm with several input data or simulate a manually controlled algorithm and play an animation sequence of this algorithm back and forth; it can also display the data structures of the algorithm in several different representations in different windows. It can be used in several different ways like an instruction video for the students about different algorithms, real-time visual execution for demonstrating and illustrating complex concepts, and both open and closed lab environments for the students to work with exercises. [10]

3.1.2 Design of a Visualization Program

To be able to make a program with a graphical representation that has a large educational value some things need to be kept in mind. First, a model of the concept that should be explained and displayed needs to be designed, and then this model has to be illustrated in a good and pedagogical way. For algorithm and data structures it is hard to make a model, because there is no real object to compare the design with. Scientists in fields like physic and biology have a real model to mimic when they make their model; this is not the case for an algorithm as it will always be an abstract concept. A visualization should, according to [13], either be something that guides the user, or rationalize things for them. When it is guiding the user it should help the user discover things he or she does not knows. When rationalizing, it illustrates things he or she already knows. It is important to separate these two things as they have different goals. But here it is important to think about who the user is. Is the user the teacher who wants to explain algorithms to his or her students or is it the students that should try to understand the algorithms. For the teacher, a rationalizing visualization is preferred, but perhaps for the students, a guiding visualization is to be preferred. Because instead of telling them how the concept that is displayed works they should discover it for themselves.

Miller explains in [13] that the difference between the methods is the organization of

information. A visualization that displays information without a detailed control by the user is guiding the user, and a visualization that takes the user's instructions on how to organize data as parameters is rationalizing, as the user already knows how this system works.

The visualization should give labeled information so the viewer can identify the system's objects and the abstract concept is easier to follow. If it requested that the user should select a desired configuration from a large set of parameters it becomes easier to select a confusing visualization, therefore, the application should always have a default configuration which creates a suitable visualization. [13]

3.1.3 Implementation Techniques

Event Driven Approach - This means that there are key points in the program to launch animations of interesting events. For example whenever the event of a message is sent between to two processes in a message passing system this event is displayed in the graphical representation.

State Driven Approach - This means that certain data structures are monitored and an animation is launched when these data structures changes. When the program reaches a certain state in the program code it will execute a state in the visualization. [10, 9]

3.2 Important Characteristics

Some characteristics for improving the educational impact when graphically illustrating and visualizing different concepts in computer science are discussed in [14].

3.2.1 Provide Complementing Explanations

Provide resources that help learners interpret the graphical representation and complement visualizations with explanations. Do not make the visualization harder to understand than necessary; in worst case it can be as difficult to map the visualization to the algorithm as it is to understand the algorithm from the beginning. This can be avoided by creating embedded clarifications in the visualization, for example text or sound. Another method could be to do an oral explanation about how the visualization works before presenting it to the students. [14]

3.2.2 Adapt to the Knowledge Level of the User

The students should not be overwhelmed by impressions when they watch the visualization, if the students have very little knowledge about how the algorithm is being presented, or algorithms in general the visualization should be adapted to this and be as simple as possible. For a more experienced student, a larger level of details and more advanced visualization could be suitable. [14]

3.2.3 Provide Multiple Views

If the algorithm can be presented in different views, for example both displaying a visualization of the algorithm and stepping through the code, it is desirable to do so. This could facilitate a deeper understanding and create a deeper understanding for mapping the code to what is happening in the presentation of the algorithm. But if different views of the visualization are displayed then it must also be coordinated to show consistent information. [14]

3.2.4 Include Execution History

Displaying the history of a simulation is a good approach as it can be hard to remember the previous steps. If history is present it can be easier to create a global picture of what has happened. [14]

3.2.5 Support Flexible Execution Control and Custom Input Data

Being able to pause, play it slower, step back and forth and replay the visualization can improve the educational value [14].

3.2.6 Support Custom Input Data Sets

Let the user be able to insert their own parameters for the algorithm visualization [14].

3.2.7 Support Dynamic Questions

Use a pop quiz approach and make the system interact with the student and give them questions to think about or answer. This forces the student's attention to specific parts of the visualization. Examples of questions could be; what will the next frame in the simulation look like, or what is the corresponding code to it. [14]

3.2.8 State Cues

When a change in the state of the data structures which are modified by the algorithm occurs the change should be reflected by a change in the graphical representation. This could for example be that something changes shape when it receives some data, it points out that this data structure has been updated. [2]

3.2.9 Continuous versus Discrete Transitions

If an object in the visualization is moved it is better to do a smooth motion and show that this object moves between two points. If the object disappears from its original point and appears on a new point it can easily be missed. [2]

3.2.10 Color

Color in algorithm animation can create many challenges. As a graphical design, it has all the problems a textbook has, but also some of its own. Visualization of an algorithm is dynamic, it moves and images are exchanged, it can also have multiple views that must be consistent but still do not interfere with each other. A good method for designing the animation is to group things that are connected with the same color and distinguish others with different colors. If the animation has multiple views it can have the same color for all the objects that are logically related to establish their relationship. Highlighting can be used on objects that are important at the moment but they should be removed and restored quickly when another object is highlighted to eliminate confusion. [2]

3.3 Conclusion

In summary, the most important thing for an educational application is to have some aspect of user interaction. If the user can solve a problem with the application as a graphical guide, it will make it easier for him or her to understand the application. To give information in different views is also important. To have information in the form of text or audio to explain the steps of the animation, and also to maybe show the graphical animation in different views, focusing on different data structures. Finally, to make the graphical visualization of the application easy to watch and understand by not showing too many details and having a clear focus on what is happening and what is changing, and not let objects make discreet changes, making every transition visible and easy to follow.

Chapter 4

Tools

4.1 Erlang

Erlang [6] was chosen for this project because it is excellent for concurrent programming and has a large set of functions to create a system of processes that communicates with each other. The Erlang processes communicates with message passing, there is no shared memory and no need to use locks to avoid race condition. The message passing between Erlang processes is also very suitable as this is a project about nodes communicating by sending messages. Erlang also has a good support for a wide range of operating systems, the application can be compiled to run on Microsoft Windows, UNIX, Linux and MAC. [5]

4.2 Erlworld

Erlworld is a framework for creating concurrent games in 2D computer graphics with Erlang. It contains bindings for both OpenGL[15] and wxErlang[7] (which are Erlang bindings for wxWidgets[16]). Erlworld makes it possible to automatically update all the parts of an application in parallel and draw them all at the same time. It was written by Joseph Lenton as a bachelor project at the University of Kent. [11]
Erlworld[11] was chosen as the graphic engine for this application for two reasons; the first one was that it was an easier and faster method to use Erlworld rather than doing it from scratch with wxErlang[7] or some other graphic library with Erlang support. The second reason was that Erlworld could make it look somewhat more like a game than a regular simulation application. Using this technique of gamification, could give the application a more conspicuous appearance. The most interesting challenge of making the application with Erlworld was that the design needed to follow the rules of Erlworld as a game engine and not as a strict graphic library. This makes the application structure interesting because the idea of having independently working processes that performed the Snapshot algorithm together was well suited with the Erlworld design.

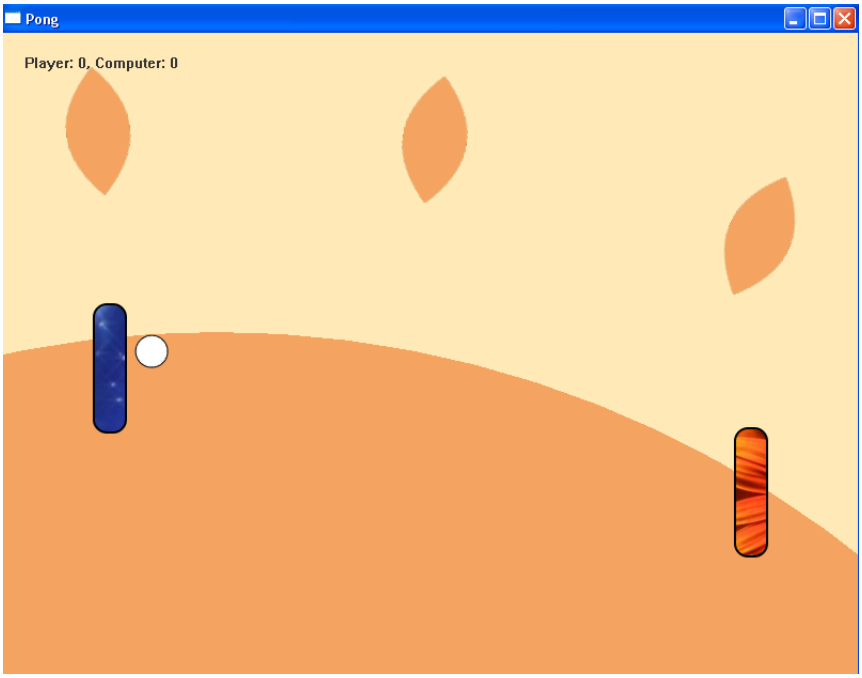

Figure 4.1: A Pong game that uses Erlworld

4.2.1 World

When an application that is created with Erlworld is running everything in the program is encapsulated in something called a world. A world is the frame that is running the execution of the program. A simple way to describing how to create a game with Erlworld is starting by creating and specifying all the components of the game. This could for example be some sort of character that the player is controlling, the enemies of the player, the game background, the score board, or pop-up messages saying that the game is over; simply, everything in the game that changes its states and updates. When all these components have been created, they are added into a world and the world executes them, the world makes the components exist in the world and behave like the source code says that they should. In other words, the world is a manager or a bootstrap for the components of the game. The World module is in some sense a sub-class of the Actor module because it works like an actor and does the three things the actors do: holds the state, updates the state with a callback function, and draws the state with a callback function. But instead of holding properties of the actor and the state of the actor, it holds actors, and when updating the state and drawing the graphics it calls the functions of every actor it holds. Everything that the world holds is in the world state which is created with a module called world state. Besides holding all the actors this state also holds the order of how the actors in the world is going to be drawn on the frame. [11]

4.2.2 Actors

The components of the game that are executed in the world are called actors, everything in the game is an actor. An actor has three main purposes: to hold the state of all characteristics that this actor owns (this is a module called actor state), to update

the actor according to the game rules with a callback function which is defined by the programmer, and to paint the actor on the interface with a callback function defined by the programmer.

```
Act = fun( AS, Parent ) ->
% do something
AS end,
Paint = fun( AS, G ) ->
 % draw something
AS end,
```

This is an example of how the callbacks can look, the Act function updates the state of the actor, and the Paint function draws the actor. As a result of the update function, Act becomes the new actor state for the next frame, and the result of the Paint function is ignored as its only purpose is to draw the graphics. The parameters of the Act function is the previous state of the actor and the PID (process identification) number to the parent of the actor which is the world process. The parameters for the Paint function is the state of the actor and a graphics object for the drawing. [11]

4.2.3 Actor_state

A module that stores all the properties that is connected to an actor, everything that is needed to create, update, draw and interact with an actor is stored in its actor state. All actors have three mandatory properties which are name, location and size. These properties must be set for the actor to exist. Except for these three mandatory properties, the actor state also contains all the other properties of the actor. In a typical game this could be for example how many points a player have scored. The properties are set with a function: `actor state:set( AS, List ) ->AS`; this function takes the actor state AS and a list of properties, which is to be added, and returns a new actor state.

To access a property of the actor the function `actor state:get( AS, Key )` is used. This function takes the actor state and the name of the property that is being accessed and returns the value of that property. [11]

4.3 Other

4.3.1 wxErlang

One tool for writing graphical user interfaces with Erlang is wxErlang, this is an application programming interface which contains Erlang bindings of wxWidgets. This is a large API, as every class in object-oriented wxWidgets API is represented in wxErlang as module. The objects created when working with wxWidgets through wxErlang is Erlang process which can change state. Since 2013 wxErlang is a part of Erlang. [16, 7]

4.3.2 wxWidgets

A widget is an object in a graphical user interface that displays some sort of graphical information that can be modified by the user, for example a textbox or a window. A

common way to make the development of the widgets faster is to use a widget toolkit, this is a set of functions to draw widgets and to create graphical interfaces without writing the same lines of codes hundreds of times. Instead a function is called, and that function creates a widget according to the preferences chosen. One of these toolkits is wxWidgets. It started as wxWindows 1992 by Julian Smart at the University of Edinburgh, and it changed name to wxWidgets on February 20, 2004, because of a dispute with Microsoft. It is currently licensed under a license called wxWindows License because the wxWidgets license has not been approved yet. The only difference between the two is the name. The wxWindows License is almost the same as a regular Library General Public License but with one exception which states that works in binary form may be distributed under the users own terms. WxWidgets are written in C++ and contain hundreds of classes for application development. For example it has classes for window layout, drawing objects such as fonts and pens, image handling, HTML rendering, and sound and video playback. [16]

4.3.3 OpenGL

OpenGL is an environment for creating portable applications with 2D and 3D graphics. It was launched in 1992. It is cross-language and multi-platform and uses the host systems own window, input and event mechanism. [15]

Chapter 5

Implementation

5.1 The Application

5.1.1 Application Structure

The application is divided into two main parts, the algorithm simulation which executes the Snapshot algorithm for a set of nodes, and the Erlworld part that starts a graphical interface which draws the simulation of the algorithm.

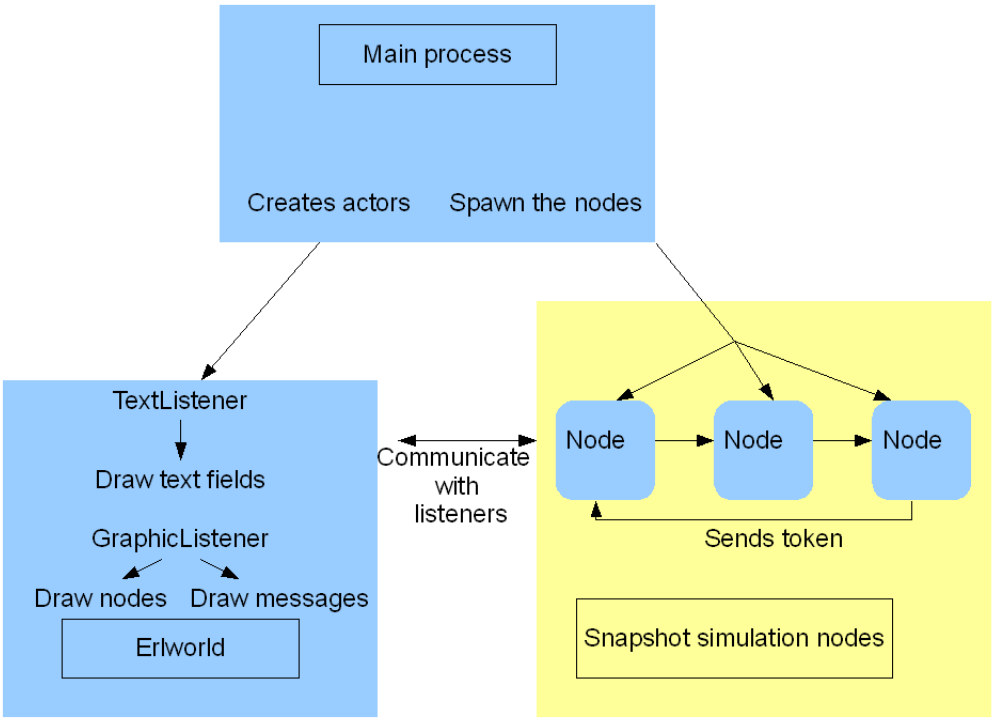

Figure 5.1: The structure of the application which is showing the relationship between Erlworld and the nodes

5.1.2 Erlworld

The simulator uses Erlworld as its graphic engine and all drawing on the screen is done by Erlworld. Erlword works with actors that exist independently in a world that encapsulates them, so every graphical object is an Erlworld actor. All the nodes that are drawn on the screen are actors that execute independently from all the other actors and different parts of the simulator. When the simulator starts the two listeners for text and graphics are created and their coordinates for the nodes are created by random number generation. All the Snapshot algorithm processes are spawned with the text listeners PID numbers as parameters so they can send data to them and also, a contact list of all the Snapshot nodes PID numbers is sent to all the Snapshot nodes. Then, the listeners are connected to an Erlworld world and the Erlworld module is started.

5.1.3 Algorithm Simulation

The simulation of the Snapshot algorithm is done by a set of nodes that all executes the same function from the beginning. The nodes are spawned by the main program and after that they execute Snapshot by their own. Before the real Snapshot simulation takes place, all the nodes prepare by receiving data that they need to perform the simulation; this is the PID number to the other nodes and their coordinate in the graphical interface. In the hibernation state they all try to initialize the Snapshot algorithm by spawning random number to get the one that moves the process into an initializing state. If they receive a marker message before they have initialized Snapshot by themselves they leave the hibernation state and continue with Snapshot only. When a process initialize the Snapshot algorithm it begins by sending a start message to the listener that controls the graphic interface to make the nodes enter, and then sending a message to the listener that controls the text messages to make it write a welcome message. The message to the text listener starts the chain of communication with that actor and the initializing node receives the identifiers to the text actors in a message from the text listener. This is because the nodes need to save these identifiers to be able to delete the actors later. Next time a node wants to update a text message on a graphical interface it sends the text identifiers to the text listener and wait for a replay with the new text identifiers.

Now the real algorithm simulation starts and the initializing node sends out markers to all the other nodes, make a call to the graphic listener to draw the messages sent between the nodes and a call to the text listener to display a text message. Before going forward and waiting for markers to get back from the nodes it sends out a token message to the next node in the contact list. This is done because there can only be one node transmitting in the graphical interface at the same time to limit the confusion and to make the graphical representation of the simulation easier to understand. When a node receives a marker and leaves the hibernation state and starts waiting for a token message to get the permission to continue with the simulation. This token message except for representing the permission for continuing with the simulation and using the listeners, also contains the text identifiers for the text fields in the graphic interface.

With the text identifiers and the permission to continue it can update the text message fields and send a graphical marker message. When it updates the text messages it also gets the new text identifiers from the text listener. When these tasks are done the process sends the token to the next process in the list together with the text identifiers so also this process can continue with the Snapshot simulation. The algorithm is finished when a node has received a marker message on all its channels and the node terminates.

5.1.4 Implementation of the Actors

```
 Act = fun( AS, Parent ) -> AS end,
Paint = fun( AS, G ) ->
graphics:draw_image( G, NodeImg, X , Y) end,
actor:new( Act, Paint ).
```

This is the structure of all the actors and how they are built. Two anonymous functions are declared. The first function has a logic task for the actor, it could for example be used to create a new actor in the world or change the current actor in some way. The function

takes an actor state for the calling actor as the first parameters and the world state as the second parameter. It returns a new actor state. The second anonymous function is the graphic function. This function does the actual drawing for the application. This function takes an actor state as the first parameter and a graphic object as the second. With these two anonymous functions the next generation of the actor is started by creating a new actor of the same function with a new actor state and a new state of the drawing.

5.2 The Nodes

5.2.1 The Node Intro

When the Snapshot algorithm has been initialized by one of the nodes, it starts the actors that draw all the nodes as they are moving from the left to the coordinate they are going to stay on during the simulation. When this happens it is actually the same function as when a node is sending marker messages to the other nodes but with a different image. The only difference between when a message is sent between two nodes and when the nodes are entering at the beginning is that when a message arrives to its destination point it the actor is removed from the world only, but when a node is entering and reaches its destination point the actor is removed and a new actor is created, and this time it is an idle node instead of a moving node.

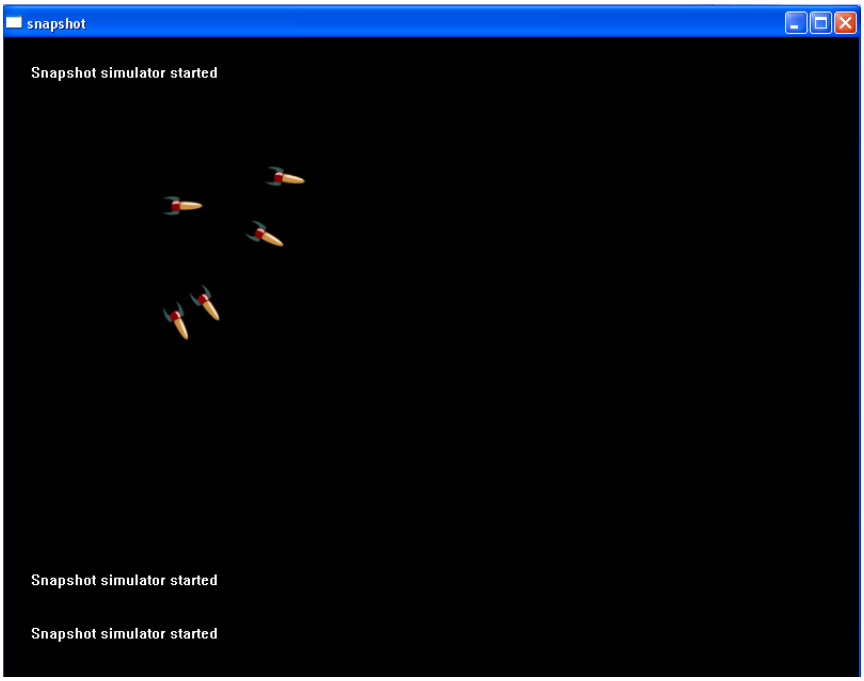

Figure 5.2: The nodes is moving to their end destinations

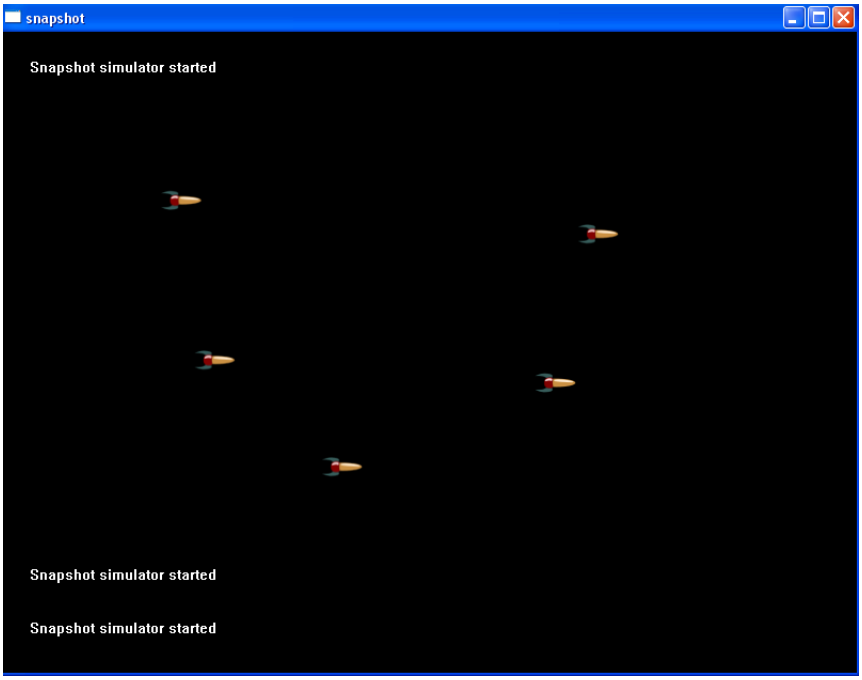

Figure 5.3: The nodes have arrived

5.2.2 Drawing Nodes

The graphic representation of the nodes in the simulation is an actor for every node. This actor holds the same structure as all the other actors with two anonymous functions, but the only thing the source code for the node actor does is to draw the node. This is done by calling the `draw_image( )` function in the graphics module of Erlworld. This function uses the OpenGL bindings in WxErlang to draw the image.

5.2.3 Changing Nodes When Message Received

To get a good view of what is happening in the simulation of the algorithm the nodes changes color when something happens in the simulation. When a node is sending out marker messages to all the other nodes it changes color to green, when it receives a message from another node it changes color to yellow, and when a node is finished with the algorithm and the process terminates it becomes red. This is done simply by adding a new actor that contains an image with the new color with a function called `add_actor()` from the world module of Erlword. To know which node are going to change a NodeNumber variable is used. This variable has been received together with the code for which color the node should change to. This number is simply nothing more than the number of the element of a list that contains all the coordinates of the nodes. So with the function `lists:nth()` it is easy to extract the correct node and change the image of the graphic objects that represents it.

Figure 5.4: Node color are switched to green and yellow

5.3 Listeners

The two listeners are in essence the core of the program. Both work in the same way and are only divided into two functions because the structure of the source code becomes easier and more readable when they are separate. They are both established as actors in the Erlworld world and works independently in the world. Their only purpose is to exist in the world and wait for orders sent to them from the nodes that simulate the Snapshot algorithm. Every time the nodes in the algorithm do something that needs to be represented on the screen they send a message with a code that explains their errand to the listener. When the listener receives this message it handles it according to the code it contains. The two listeners in the program are TextListener and GraphicListener.

5.3.1 TextListener

The text listener handles all the communication with the lines of text. When a line needs to be updated it receives a message with a code how it should be updated and with the identifiers of the old text that is held by the Snapshot nodes from the node that requested this update. The listener destroys the actors that draw the text on the screen with the identifiers it received and then spawns news actors for the new message. The identifiers for the new text actors are then sent back to the node that requested the update.

- {`start,text, Sender`} - Tells the listener to write that the simulator have started.
- {`green,RecText1,RecText2,RecText3,Sender`} - This code tells the listener to write that the green node is now sending markers to the other nodes.

- {`markerreceived,NodeNumber,Sender,RecText2,RecText3,Caller`} - This code tells the listener to write that the yellow node have now received a marker from another node.

- {`terminate, NodeNumber`} - This code tells the listener to write that the red node have now terminated because it has finished the execution of the algorithm.

The Rec1, Rec2 and Rec3 variables in the messages are identifiers for the text actors. Because the text listener actor can not carry the identifiers for the text line actors they need to be sent to the Snapshot nodes so the actors can be deleted when the text is being updated.

5.3.2 GraphicListener

The graphic listener handles the communication with the actors that draws the nodes and the messages sent between the nodes. It works in the same way as the text listener and exists as an actor which only task is waiting for messages from the Snapshot nodes containing codes about how to update the items on the screen.

The different codes for updating the screen by the graphic listener are:

- {`start`} - This message starts the whole simulation by requesting that the graphic listener draws the nodes that is going to be used in the simulation. When this message is received the graphic listener first spawns the nodes as objects moving from the left of the screen to the position which they are going to hold during the simulation. When the moving objects have reached their points the actors are destroyed and replaced by new actors that now hold node drawings that is fixed on one point.

- {`green,NodeNumber`} - This code indicates that a node is sending markers messages to the other nodes and the drawing of the node on the screen is switched to a green node.

- {`markerreceived,NodeNumber,C`} - This code indicates that a node is receiving markers messages from the other nodes and the drawing of the node on the screen is switched to a yellow node.

- {`terminate, NodeNumber`} - This code indicates that the node is now finished with the algorithm and terminates, the node on the screen becomes red.

- {`send, From, To,NodeNumber`} - This is the code to tell the graphic listener to draw the small dots that is representing the messages being sent from one node to another.

5.4 Writing/Displaying Text Comments

The method of displaying the text messages in the application is done in a similar way as the displaying of the nodes. The difference is that there is no ready image to display; such an image needs to be created every time the text messages are updated. When the

application starts an actor is created for all the three text images that are displayed when the simulator is running. This actor is created by the function `new_text()` which work almost as the draw node actor, but when it is created it starts by creating an actor state that contains an image of a text message, this is done by using the function `new()` in the text_image module of Erlworld which like the graphics module also uses WxErlang. When this actor state is created (the text object is created) it continues by using the same anonymous functions structure as always and draws the image with the `draw_image( )` function in the graphics module of Erlworld. In an earlier stage of this project a function to update the text and replace it with something else was used, this function is called `update_text()`. This function is not an actor; it is only a function to change an existing text actor. It took the text actor as a parameter and then extracted the actor state from the actor and destroyed it with the function `destroy()` from the text_image module of Erlworld. But this method did not work all the time because it did not properly remove the old text image. Therefore, a new method of simply removing the whole actor that contained the text image and create a new one on the same location that contains the new message was created. This method was much safer as it was now guaranteed that the old text message was gone.

5.5 Moving Objects

5.5.1 Sending Messages

Every time a node sends a marker message to another node the graphical interface sends a small yellow dot between them. This dot is representing the marker message being sent. Every marker dot that moves on the screen is a single actor that only exists between the time it is sent from the sending node and the time it reaches the receiving node. To do this a separate a function called `mover()` which creates moving actors is used. The important data when these moving objects are created is the coordinates of the node sending them and coordinates of the node that should receive them. With this information it is easy to calculate the angle between the sending node and the receiving node which is the most important part as the moving object start its existence by moving in a specific angle. So it is important that this angle moves the object in the direction of the receiving node.

During the whole existence of the moving objects it constantly checks where it is in the coordinate system and if its current coordinate is equal too to the goal coordinate which is the coordinate of the receiving node. This is done by a simple code that checks if the current X and Y coordinate is almost equal to the receiving nodes coordinate, why it is almost and not the exact coordinate is because the moving object rarely reaches the exact coordinate so it is necessary to check a close radius instead. If the X coordinates does not have an absolute difference larger than two and the Y coordinates does not have an absolute difference larger than three then it is appropriate to say that it is a hit and the moving object have reached the receiving node. These values have been established by testing and they fit with the size of the node image that is currently used. When the coordinates of the moving object is almost the same as the receiving nodes coordinates the object kills itself by using the `remove_actor()` function in the world module of Erlworld and it is deleted from the graphical interface

5.5.2 Mover

The actual method of moving the objects is done by a small set of functions. One construction function `newMover()` that creates the actor state of the moving object with the information about where it should start, in which angle it will move forward and which image it will draw. The `act_move()` function calculates the speed of the object, the angle of it, and where the new coordinate of the moving object will be drawn next. For every 'tick' in the system the moving object will be drawn in a different coordinate so it will look like that it is actually moving forward. The actor state is then drawn on graphic interface with a function called `paintMover()`. This function simply extracts the data in the actor state and draws it with the function `draw_image_rotated()` from the Erlworld module graphics. The real drawing in done with the Erlang bindings for OpenGL.

Figure 5.5: Messages is sent with the mover functions

5.6 An Example of an Execution

There is a network of three nodes which are going to simulate a Snapshot session. Node number one has incoming and outgoing channels to both the other nodes. The other two nodes both have an incoming and an outgoing channel to node number one. The application starts by configuring and spawning the Erlworld actors, the text message field, the text listener, and the graphic listener. The Snapshot nodes are spawned as individual processes and they receive the process id numbers for the text listener. The main process sends the contact list with all the process id numbers for the Snapshot nodes to every Snapshot node. The nodes go into hibernation state where they wait for a Snapshot session to start. The main process starts the Erlworld main function and becomes an Erlworld instance.

Node number one initiates the Snapshot algorithm This node sends the message `start` to the graphic listener, this message makes the graphic listener draw the nodes on the screen. It continues with sending `start,text,Pid` where `Pid` is its own process id number to the test listener which draws a text on the screen that node one has started the Snapshot algorithm. Node one sends a message to the graphic listener telling it to mark its graphical corresponding node to green to indicate that it is sending messages to the other nodes. It then, sends messages to the graphic listener telling it which nodes it is going to send Snapshot marker messages to, so the graphic listener can create graphical messages between the nodes, and finally it sends Snapshot markers messages to its connected nodes two and three. This is the marker sending rule. After the marker sending rule, it sends a token for performing execution to the next node, number two, and start to wait for marker messages from the other nodes.

Node number two receives a marker message and also the token for performing execution It sends a message to the graphic listener telling it to change the graphical representation of it to the color yellow, to mark that it have received a marker message. It performs the marker receiving rule, and sends a message to the graphic listener that it will send marker messages to its outgoing channels so its graphical representation change color to green. Then node two sends messages to the graphical listener to launch graphical messages on its outgoing channel to node one, and then it sends a marker message to node number one. Finally it sends the execution token to the next node, node number three. As it now has sent marker messages and received markers on all its incoming channels it terminates the Snapshot algorithm and goes into a hibernation state where it only resends tokens when it receives an execution token.

Node number three receives a marker message and also the token for performing execution. Much like node number two, it sends a message to the graphic listener and tells it to change the color to yellow to mark that it has received a marker message. It performs the marker receive rule and sends messages to the graphic listener to mark it green for sending and draw the message it sends to its only outgoing channel, node one, and then it sends a marker message to node one. As with node two, it has not received a marker on all its incoming channels and sends the execution token to next node which is node one, and then it goes into hibernation.

Node number one receives a marker message from node number two and also a token, it removes node number one from the waitlist and sends token to the next node, node number two, and continues to wait for marker messages.

Node number two receives a token and is in hibernation, therefore, it sends the token to the next node, node number three.

The same thing happens with node number three, it receives a token and is in hibernation and therefore, it sends the token to the next node, node number one.

Finally, node number one receives a token and a marker message from node three, and then removes node three from the waitlist. It has now received a marker on all its incoming channels and can go into hibernation. The Snapshot algorithm is finished.

Chapter 6

Evaluation

6.1 Evaluate the Effectiveness of the Visualization

The purpose of this application is to help people understand how the Snapshot algorithm works by seeing it in action. A good question to ask is therefore: How effective is this application to create understanding of the Snapshot algorithm? There are a couple of methods for finding and answer to this question.

6.2 Compare with Established Research about Algorithm Animation

The chapter Visualization presents several characteristics that are important for creating an understanding of abstract concepts. The Snapshot animation application contains some of these characteristics and lacks others. It provides complementing explanations in the form of text strings that tells the user what is happening on the screen, and which step the algorithm performs at that stage. It is a simple animation with not too many details to follow, and therefore, it is adjusted for students who have little knowledge of the concept.

When messages are sent and received the nodes change color. This is a good pedagogical support as the user can see when a certain data structure is changed. The messages being sent are also moving in a smooth transition so the user can easily see how the object is moving from one point to another, this better than having the objects disappear at the starting point and reappear at the end point. The application also uses color as a method to group things. All the messages have the same color and the nodes change color when they change tasks. In one sense, it includes the execution history because it can replay the animation several times, but this is perhaps not the best form of providing execution history. It is also possible for the user to use custom input, but with one limitation; it is only possible to control the number of participating nodes, it does not provide multiple views of the animation, and it lacks flexible execution control as step through the animation; to go back one step and replay a specific sequence.

6.3 Input from User Interviews

During the design and implementation process, several people tried the application and looked at the animation and gave their input of when the animation was simple to follow and when it was not. These tests did not have the purpose of seeing if the users understood the Snapshot algorithm from the animation, it only gave information about if they understood what was happening on the screen. For example, if they understood that a node was sending a message to another node, and if they could see a pattern when the nodes changed color when changing tasks. This method of evaluation provided much information about the importance of keeping the animation slow, to do every step of the algorithm as a single sequence, and to never let more than one step be performed at the same time.

6.4 Possible Testing on a Group of Learners

A more advanced method of evaluating the pedagogical value of the Snapshot animation could be to conduct a survey and display the animation in front of a group of students with different levels of knowledge; these could for example be computer science students with experience of computer science studies but no experience of the Snapshot algorithm, students with no knowledge of algorithms at all, students with experience of other types of synchronization algorithms but no experience of Snapshot, students with earlier experience of Snapshot, and students with a high knowledge of Snapshot. After having looked at the animation a couple of times they should answer a form with some questions about their experience and what they learned, for example: give a simple explanation of the Snapshot algorithm.

Comparing the answers to their previous experience could be a good indicator of how well the Snapshot animation provides them with knowledge about how the algorithm works. This has not been done but could be a final test of the application to see if it fills its purpose as an educational software.

Chapter 7

Conclusion

7.1 Result

This project has resulted in the creation of an application written completely in Erlang for animation of the Snapshot algorithm. The application can simulate the Snapshot algorithm in either a randomized network, or in a network chosen by the user. It performs the simulation from beginning to end, and then repeats the same simulation until the application is closed. It also provides explanations in the form of text strings on the screen as the simulation is executed. The research and development in this project created a greater understanding of the Snapshot algorithm, and knowledge about algorithm visualization and creating graphical applications with Erlang.

The major achievement of this project was to extend Snapshot so that it would work to display an animation and not be too confusing and execute too fast, and also, to find a suitable method to create the graphical animation. This task was one of the most time-consuming because it demanded a lot of testing and experimenting with different methods for writing graphical applications with Erlang. Erlworld was chosen because it had bindings to both wxErlang and OpenGL and it was interesting to use a framework created for constructing games to create something that was not a game, but could be designed with a gamification approach to make it more interesting to use. It was very interesting to learn about Erlworld and how it worked, and even more interesting to combine it with the task of visualizing an algorithm like Snapshot because to do this it was necessary to use Erlworld in a way it was not designed for. Instead of having one single Erlworld application that encapsulated everything, Erlworld needed to take the role as a graphical server that communicated with independent outside processes. Making this communication work in a satisfying way was an interesting and hard challenge. Because Erlworld is supposed to work by itself without interference from external processes, and it is not a trivial thing to communicate with internal parts of Erlworld, some of the parts of the source code became very complex. The most challenging task was ensuring that the external processes had the correct process identification numbers to the internal parts of Erlworld that they communicated with. This because the internal parts of Erlworld constantly respawns and therefore obtains a new process identification number.

7.2 Limitations

The application has several drawbacks and limitations. These can be divided into two categories, technical and pedagogical. The technical limitations manifests as two different symptoms but they both come from the same source; too many nodes participating in the Snapshot algorithm. The first symptom is that the animation becomes a little confusing when too many nodes are participating in the simulation. There is no exact number for this, it is a matter of personal preferences, but more than ten nodes would probably make the animation screen a little too crowded and create drawbacks on the usability for most people. The second symptom is that when there are too many nodes the application becomes a little slow. This is because there are too many objects to be updated and drawn on the screen. In this case it is easier to find a maximum number; ten nodes makes it become slow, but it could also be the fact that there is a large number of objects on the screen that makes it feel slower than it is as the maximum number for performance is almost the same as the number of the nodes that start create confusion.

The major pedagogical limitation of the application is that it does not contain any interaction possibility for the user, except for starting an animation and setting the number of participating nodes in the Snapshot simulation. As stated in the Visualization chapter, research about visualization has the highest pedagogical value when it has a high number of user interaction possibilities. Another pedagogical limitation is that the application only displays one view of the simulation; it is not possible to isolate a single data structure for supervision.

Chapter 8

Possible improvements

8.1 Future Design Ideas

This section describes ideas that have been considered during the design and implementation process but have been left aside, and also brief ideas about how the application could be extended or what parts a new application could contain.

8.2 User Interaction

This application is purely an animation to watch and do not contain any interaction parts from the user except for starting and setting the desired network to be animated. Therefore it lacks the higher levels of educational software that previous chapters have described. In other words, what is missing is some user controls to create full understanding for users who are not advanced. Interesting types of user controls could be to make an application that use a strong gamification view and make the user answer questions about what is going to happen next in the algorithm. Another idea is to create a mode where the user controls the algorithm and performs the steps, and if the user does not follow the algorithm a warning and a scrollback appears, and the step needs to be done again until the correct action has been performed, and then the algorithm move forward to the next step. This forces the user to perform the complete algorithm and obtain a visualization of it at the same time.

8.3 Extended Control

The application today also lacks support of stopping, play backwards, play faster or slower or step through the animation. These features could be good to extend with so the user immediately can go back, slowdown or stop if he or she needs to more time to understand a step in the algorithm animation. Especially the function of starting from a certain point or fast forward to a point would be good so that the user do not need to watch steps that he or she understands again and again to come to the point that he or she does not understand, and then the user could loop that part until he or she understands it.

8.4 Extended Text Information

Another possible improvement would be to extend the text comments so it is possible to display comments in a moving list at the edge of the screen so that everything that is happening can be commented on and the pseduocode for the steps can also be displayed. In comparison to today, it would be possible to display much more text information and keep the text longer on the screen. With more and longer text information on screen the user could reflect more on what happens. Also, another possibility would have been to display the steps of the pseducode before the animation takes place, so the user can visualize the step for him- or herself before seeing the correct visualization

Bibliography

[1] Ryan S. Baker and Michael Boilen. Testers and visualizers for teaching data structures. 1999.

[2] M.H. Brown and J. Hershberger. Color and sound in algorithm animation. 1992.

[3] K. Mani Chandy and Leslie Lamport. Disributed snapshots: Determining global states of distributed systems. 1984.

[4] George Coulouris, Jean Dollimore, and Tim Kindberg. Distributed systems: Concepts and design. 2001.

[5] Erlang. Erlang - implementation and ports of erlang. http://www.erlang.org/faq/implementations.html, September 2013.

[6] Erlang. Erlang programming language. http://www.erlang.org, September 2013.

[7] Erlang. wxerlang reference manual. http://www.erlang.org/doc/apps/wx, September 2013.

[8] Christopher D. Hundhausen, Sarah A. Douglas, and John T. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing*, 2002.

[9] Andreas Kerren and John T. Stasko. Algorithm animation - introduction. *Software Visualization State of the Art Survey*, 2002.

[10] Ari Korhonen. Visual algorithm simulation. 2003.

[11] Joseph Lenton. Erlworld - concurrent game framework for erlang. https://code.google.com/p/erlworld, September 2013.

[12] Nancy Lynch. Distributed algorhitms. 1996.

[13] B.P Miller. What to draw? when to draw? an essay on parallel program visualization. *Journal of Parallel and Distributed Computing*, 1993.

[14] Thomas L. Naps and Guido Roessling. Exploring the role of visualization and engagement in computer science education. 2002.

[15] OpenGL. Opengl - the industry standard for high performance graphics. http://www.opengl.org, September 2013.

[16] wxWidgets. wxwidgets. http://www.wxwidgets.org, September 2013.