Jest

Innovative Multi-touch Interaction

Jelle Dirk Licht - 4106946
Ferdy Moon Soo Beekmans - 1327755
Matthijs van Otterdijk - 1308203

TU Delft

Gemeente Delft

Preface

This is the final report of the *Innovative Multi-Touch Interaction* Bachelor Project for Delft University of Technology by Matthijs van Otterdijk, Ferdy Moon Soo Beekmans and Jelle Licht. The project was an assignment by the muncipality of Delft (Gemeente Delft) in order to explore the potential of touch tables.

The assignment was to design an application that leverages the size and multi-touch functionality of the table. We decided early on to build a collaborative game, called Jest, in which players draw roads and route trucks to transport cargo. This project started on April 2013 and ended in August 2013, and was overseen by Rafael Bidarra and Elmar Eisemann of the Computer Graphics and Visualisation Group.

This document contains the report itself, as well as appendices containing our planning, the feedback we received from the Software Improvement Group and our original orientation report.

Acknowledgements

We would like to thank the following people for their help.

Rafael Bidarra and Elmar Eisemann - our TU coaches.

Farhad Keshavarz and Dirk-Jan van Dijk - our contacts at the Gemeente Delft.

Felix Akkermans - our graphics designer.

Summary

Given the rise of touch-screen technology in recent years, it makes sense to explore new possibilities to employ them. So too thought the municipality of Delft. For their planned new office, they are currently experimenting with different ways and technologies to boost productivity.

These experiments take place in the innovation lab, or iLab. One such experiment aims to find innovative uses for touch-screens. The muncipality of Delft has outsourced part of this experiment to teams of students from the Delft University of Technology as their Bachelor project.

A collaborative multi touch game, called Jest, was chosen as the product to meet these needs. By focusing on the interaction and the user experience, familiarity with innovative techniques can be nourished. This report documents the development process of Jest.

In order to ensure a modicum of technical quality, proven techniques such as unit testing and pair programming were used. An external party also evaluated the code artifacts of the product, and commended the clear structure and simple separation of concerns, but found some amount of code duplication in the tests.

In Jest, players have to create a network of paths and instructions to guide vehicles carrying cargo to their destinations. Players face several challenges along the way, such as timing issues and interesting level properties.

Jest is developed for the JVM in the Clojure language, loosely following a system design focusing on clear module separation and scalability. Jest is deployable on both Windows and GNU/Linux.

Several hurdles had to be overcome while developing Jest, such as the fact that the JVM originally did not support the multi touch API on Windows, requiring jumping through several hoops to make this work.

The project was delayed, mostly due to developer inexperience, which led to the project lasting 18 instead of 10 weeks. Aside from some rough spots in the planning and development process, the project resulted in a working prototype which can easily be improved upon for many different situations.

Contents

1. Introduction

Developing games is a resource intensive process, even more so when experimental modes of interaction are involved. This report documents the fruits of 18 weeks of labor on the *innovative multi-touch interaction* Bachelor Project.

What follows is an attempt to create an understanding of the context in which the project took place, by introducing the client and why this project is relevant to them.

Client Gemeente Delft (GD) is currently experimenting with innovative ways of stimulating productivity and creativity. One of the major environments in which new ideas are being deployed is the Innovation Lab (iLab), a picture of which can be seen in Figure 1. GD's goal with the iLab is to provide testing grounds for experimental products in a relaxed environment which encourages iLab visitors to think outside the box. Products which are evaluated favorably can eventually be used in more mainstream offices, such as the ones that are currently being built for GD in Delft.

Relevance GD is in possession of a touch screen table, which is currently stationed in the iLab. The GD wishes to use this table to explore innovative multi-touch interaction to see if touch tables can be used by the muncipality in the future for various tasks, rather than the more common mouse and keyboard interaction.

Report overview In the upcoming sections, we will discuss the assignment, present an overview of the development process, and finally introduce the game by explaining the concepts which come together to form the experience we decided to call Jest.

The rest of this report has a focus on consecutively the system architecture and the challenges we faced when implementing the system, followed by feedback on the system by external groups and an evaluation of the product in light of the assignment.

We conclude this report with a list of future recommendations and a section where we discuss our conclusions.

Attached to this report you will the sections we deemed relevant for creating an in-depth understanding of the project. These sections will be referred to when

Figure 1: A side view of the touch screen table in the iLab.

2. Assignment

This section aims to more clearly define the assignment as it was interpreted. Most of the information presented here is a slightly formalized version of the assignment as defined in Appendix C.

2.1. Project client

As mentioned in the introduction, the client is Gemeente Delft(GD). The contact person at GD for this project are Farhad Keshavarz and Dirk-Jan van Dijk.

2.2. Project Definition

GD does not have a lot of specific demands, as this project is meant as a demonstration of the capabilities of the touch screen table in the iLab. After the orientation meeting with the GD representative, the scope of the project was narrowed down to an interactive, collaborative multi touch game, designed to encourage people to try out the touch table.

2.2.1. Requirements

The information from the GD representative, combined with some self-imposed, measurable design goals to aid in the development of Jest, led to the following set of requirements;

- The game should make extensive use of the touch screen interface
- The game should have a focus on collaborative editing of a game world
- The game should look appealing and inviting

2.2.2. Constraints

Several constraints can be identified, such as available time. The most important of these constraints are;

- The projects needs to be finished in 18 weeks[1].
- The game needs to run on the hardware as defined in Appendix C
- *Optional* The game should be able to run in modified form on the development platform of choice (GNU/Linux).

[1] Originally, the project was supposed to be finished in 10 weeks, but several factors led to an extension of the project deadline. See Section 3.

2.2.3. Risk analysis

During the orientation phase, an assessment was made of which areas of the project as a whole came with the most uncertainties was made.

- Developers had no prior experience with designing and developing games
- Developers had no extensive experience with computer graphics
- Developers had no experience with multi touch interfaces.

It was deemed essential to gain more insight in these areas during the orientation phase[2].

2.3. Miscellaneous

To complete the project overview, some issues still need to addressed.

Clojure For this project the Clojure programming language was used. The reason to use Clojure has been explained the orientation report (see Appendix C).

Copyright Copyright of each code artifact lies with the authors of that specific code artifact, unless otherwise noted. This implies that both the Brick en Jest code-bases are currently copyrighted by the developers of these projects[3]. As the graphics for the game were commissioned by the Delft University of Technology, the copyright for these lies with this organization.

License As most libraries in the Clojure ecosystem are distributed under the terms of the Eclipse Public License v1.0 (EPL-v10[4]), it was a practical decision to apply the terms of the EPL-v10 license to all created code artifacts.

[2]The orientation phase of the project consisted of the first 2 weeks of the development period.
[3]Whom coincidentally are the authors of this report.
[4]`http://www.eclipse.org/legal/epl-v10.html`

3. Process

This section describes the process followed by the team including an overview of the daily workflow, a planning and the tools used to aid in the development process. For a reflection on the process, see Section 9

3.1. Roles

In every team, members will have one or more specific roles and responsibilities. These roles can be either implicit or explicit. Three clearly separated domains have been identified with supplementary roles. Every member has been assigned a role from each domain.

- Technical roles
  - Architect, who is responsible for the architecture and it's coherence.
  - Lead testing, responsible for what is tested when by who
  - Lead quality assurance, responsible for the maintainability of the code
- Gameplay roles
  - Lead game designer, responsible for the coherence of gameplay
  - Lead user-testing, plans, guides and supervises user-tests
  - Lead production, responsible the finished product looks complete
- Administrative
  - Project lead, Keeps a good overview of project and upcoming deadlines and steers accordingly
  - Editor, makes sure all report deliverables are complete and consistent
  - Contact, manages communication with external parties

3.2. Planning

At the start of every week, a list of complementary tasks was drawn up to be completed during the rest of the week. An overview of this planning can be found in Appendix A. This way of planning gave a lot of flexibility. It was anticipated that changes in the game mechanics would occur frequently based on feedback from user tests, to accommodate this, the flexibility was a must. Sadly not everything went according to plan, for a reflection see Appendix 9.

3.3. Tools

Several tools that simplified each step of the development process have been used. This subsection aims to elaborate on the uses for each of these tools.

3.3.1. Git

Git[5] is one of the many version control systems available. Git supports a distributed workflow, which means that every local copy of a repository is itself a repository which can push and pull changes from other repositories. One big advantage is a dramatically decreased number of merges. To avoid having to push every change to every other member of the team[6], a central repository is used[7].

Github tickets Github provides an integrated ticket system[8]. This system has support for tickets that can be part of a milestones, assigned to a contributors and given a deadline. Mail notifications can also be configured.

Wiki Every repository comes with the possibility to host a Wiki[9] which can be accessed via the website and as an extra repository. The latter made it possible to edit documents in one's preferred editor.

3.3.2. Google groups

In anticipation of the assignment, Google groups[10] was used for brainstorming and discussions. Groups can be used both as a forum and as a mailing list resulting in a low barrier to post and an archive. Soon after the project started, Groups was abandoned in favor of in person discussions, documented on the Wiki.

3.3.3. Trello

Approximately half-way through the project, Github tickets have been abandoned because Trello[11] proved to be more flexible. Trello's interface provided a better overview on the overall progress. A Trello board closely resembles a bulletin board organized in columns, as seen in Figure 2. The columns are filled with cards which can have colored labels, check lists systems and can be assigned to 0 or more users.

[5] `http://git-scm.com/`
[6] Or even every computer of every member of the team.
[7] `http://www.github.com/fmsbeekmans/jest.git`
[8] `https://github.com/fmsbeekmans/jest/issues`
[9] `https://github.com/fmsbeekmans/jest/wiki/_pages`
[10] `https://groups.google.com/forum/#\!overview`
[11] `https://trello.com/`

There are many ways Trello can be used. In this case an overview of what is being worked on now, today, this week and a list of milestones was kept and updated at the beginning and end of every day.

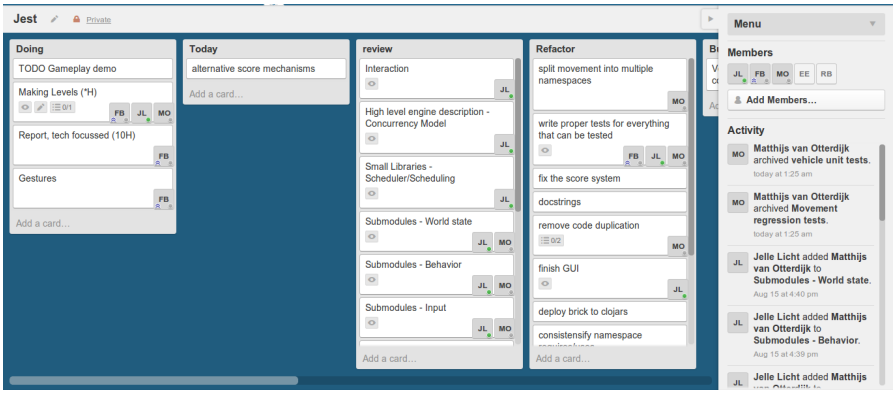


Figure 2: Overview of a Trello board

3.3.4. Travis

For continuous integration (CI), Travis-CI was used[12]. Every time new code was submitted to the central git repository, the tests were run to see if the tests are still passing. In case of a failed test or a broken build, each developer would receive an automated email from the build system, to draw attention to the fact that a problem needed fixing.

3.3.5. IRC

Most of the time, all team members worked from the same room. This was not always the case, so an IRC[13] channel was used to communicate when face-to-face communications were not practical. IRC is an old chat protocol with clients integrated in many environments.

3.4. Pair-Programming

Pair programming, or pairing, is the practice of programming in a pair behind one computer. Each programmer picks role, one behind the keyboard referred to as the driver, and the other known as the observer. The observer instantly reviews what the driver is doing. Pair programmers usually end up with better

[12] `https://travis-ci.org/fmsbeekmans/jest/builds`
[13] `www.irc.org`

and smaller designs and less bugs. On the other side, double the resources are required. Therefore it is impractical to pair-program everything.

It was decided that pair programming was to be applied mainly in 2 situations. One being during prototyping phase of a new complex component, to make sure the designs are well thought out. The other situation being error-prone code.

3.5. Work flow

When working on new code, first a new branch is created and the problem is explored and prototyped. If the component being worked on is prominent or complex, development in this phase is done in pairs when possible. When a prototype is finished, an interface is defined followed by the corresponding tests, meaning every function is implemented with its tests one by one. As a result, the developer will be aware of the semantics of what he is working on and helps bring to light possible edge cases early in the process. When both the implementation and tests of a feature-branch are finished and ready for review, the branch is rebased on top of the most recent version of the `dev` branch, after which a final run of all unit-tests determines if the rebased code will be pushed to the central repository.

3.6. Quality Control

There are several prevalent methods of maintaining a high level of quality in the development process of a potentially complex software system. No matter the amount of preparation and time spent polishing internal interfaces, eventually the time comes when a feature has to be refactored.

Static code evaluation tools are ran and their output processed at least once a week to keep track of various properties of the code-base. The tools used and the information they provide are:

- Kibit[14] - Looks where code is written in a non-idiomatic manner and suggests equivalent alternatives
- Bikeshed[15] - Checks the formatting, size of source files, lack of documentation and some code-smells
- Eastwood[16] - Checks for other code-smells, common pitfalls and spelling errors in certain parts of the code.

[14] `https://github.com/jonase/kibit`
[15] `https://github.com/dakrone/lein-bikeshed`
[16] `https://github.com/jonase/eastwood`

All code in the `dev` branch in the Github repository should have automated tests. Every namespace should contain a logic grouping of functionality and should not be too large[17].

Functional programming is oft described as programming in verbs juxtaposed against Object oriented programming as programming in nouns. In this metaphor functions are the verbs and objects are described as nouns. Programming in this manner works best when functions have a single goal. Function names such as `do-a-and-b` are a smell of a complected function that ought to be split in `do-a` and `do-b`, with higher level functions composing `do-a` and `do-b` when this is really needed.

When, despite testing, some defect in the existing code-base comes to light, regression tests will be added and the bug repaired.

Testing with Midje Midje[18] is a testing Domain specific language (DSL) (Domain Specific Language) for the Clojure programming language. Tests are writen as facts. Facts contain checks which should be true. An example can be seen in Listing 1.

Code snippet 1: A simple fact

```
(fact "2 is an even number"
  2 => even?)
```

In this example, Midje validates that the number `2` is, in fact, an even number. The `2` in this example can be replaced by any valid Clojure expression. Using Midje, functions can also be mocked. Listing 2 shows how Midje's `provided` functionality can be used to abstract away underlying function implementations. Note that `..file-name..` is an example of a meta-constant; the particular value of this symbol is irrelevant, as long as several references to the same meta-constant have the same value.

Code snippet 2: An example of mocking in Midje

```
(fact "Building a level from file should behave predictable"
  (build-level-from-file ..file-name..) => ..file-level..
  (provided
    (read-file-from-disk ..file-name..) => ..file-contents..
    (level-from-contents ..file-contents..) => ..file-level..))
```

Midje also comes with an autotest mode. After a file in the project is changed, all the tests whose outcome might have changed are ran. Developing using this mode ensures that any breaking changes a developer introduces in the code base are clearly visible.

[17] As a guideline, Clojurians consider 150 lines a fairly large namespace.
[18] `https://github.com/marick/Midje`

4. Game design

In this chapter the game design will be explained. This design differs somewhat from the one that can be found in the game design in Appendix C, because during development and testing various changes were made to the design. What is presented here is the final design as it appears in the game.

4.1. Game Overview

Jest is a game about traffic control and building infrastructure. The goal is to deliver the right amount cargo to the right depots. To do this, the players build a road network to connect supply points to depots, then route trucks on this road network.

The game consists of various levels that pose different challenges to the players. The levels are won by ensuring that enough cargo is delivered at each depot. When all depots are filled, the level has been won.

Through analysis and discussion, the players collaboratively solve the levels. After solving a level, the players receive a score which represents how well they did.

4.2. A Typical Gameplay Session

When a level starts, the players are presented with a top-down view of the game world. This world contains various buildings and some initial roads. Figure 3 shows such an initial state.

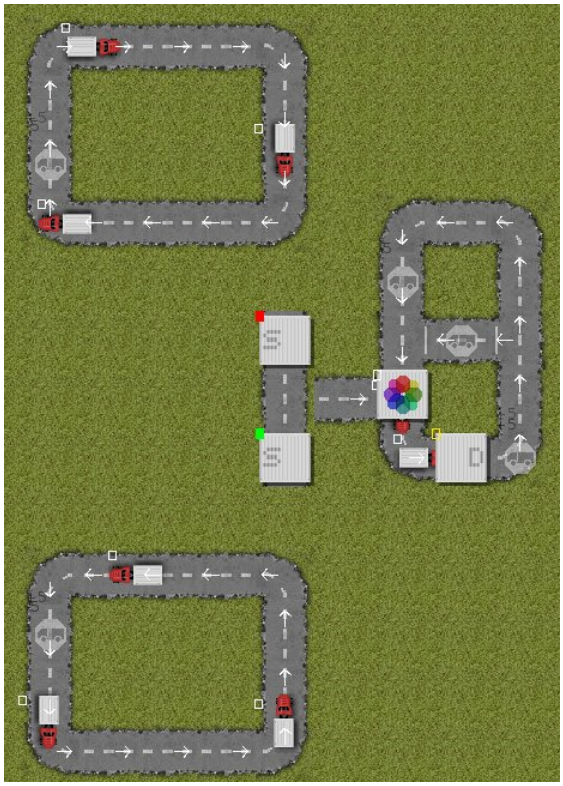


Figure 3: A level in its initial state

For all levels, the roads are built in such a way that vehicles can safely spawn without adversely affecting the game. This allows the players to calmly analyze and discuss the level.

Gameplay proceeds in roughly 3 stages.

1. Analysis
2. Build
3. Route

In the Analysis stage, the players look at what is required for this level and what the constraints and difficulties are. In the Build stage, the players draw roads connecting the various buildings correctly. In the Route stage, players route the vehicles to the right paths to solve the level.

At any one point, the players might return to an earlier stage. For example, when the road network turns out to be incorrect, it can be corrected at any time. Furthermore, some levels actually require that the road network is altered halfway into the game.

4.3. Game Elements

The game world consists of a grid of cells. Each cell can optionally contain a building and roads. These roads are traversed by vehicles that carry cargo.

4.3.1. Cargo

Cargo is represented by colors. Vehicles pick up cargo at supplies and drop them off at depots. For example, Figure 4 shows a vehicle that just picked up some red cargo at a red supply, which it is about to drop off at a red depot.

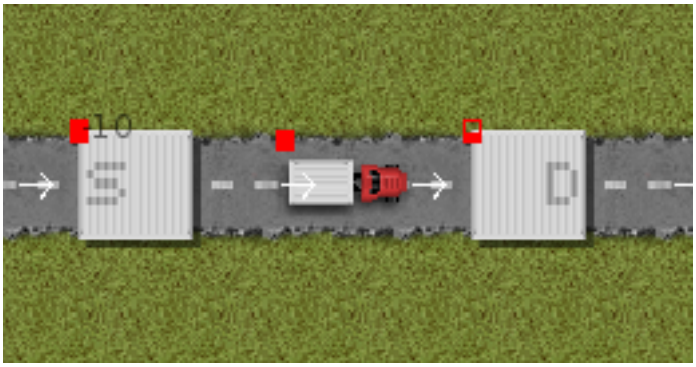


Figure 4: Red cargo has been picked up at a supply, and is about to be delivered at a depot.

4.3.2. Buildings

A building gives a cell a special function. The game has 5 building types.

Spawns A spawn is a building, from which vehicles emerge. Once they are no longer needed, it is the intention that they are parked at a spawn again. See Figure 5 for an example of a spawn.

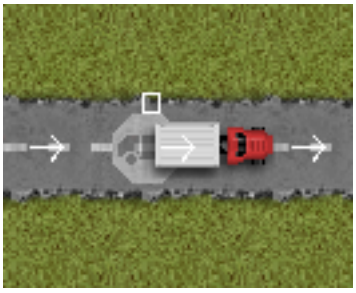


Figure 5: Vehicle leaving spawn

Supplies are buildings where vehicles pick up their cargo. Each supply provides cargo in a specific color. When a vehicle drives through a supply, it picks up this cargo, unless it is already fully loaded. Figure 6 shows a Green supply.

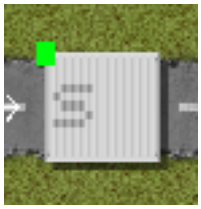


Figure 6: Green supply

Depots are buildings where vehicles drop off their cargo. All depots have a cargo quota that needs to be met to complete the level. Figure 7 shows an empty green depot. The rectangle in the top left fills up as the depot is filled.

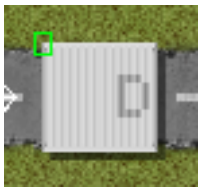
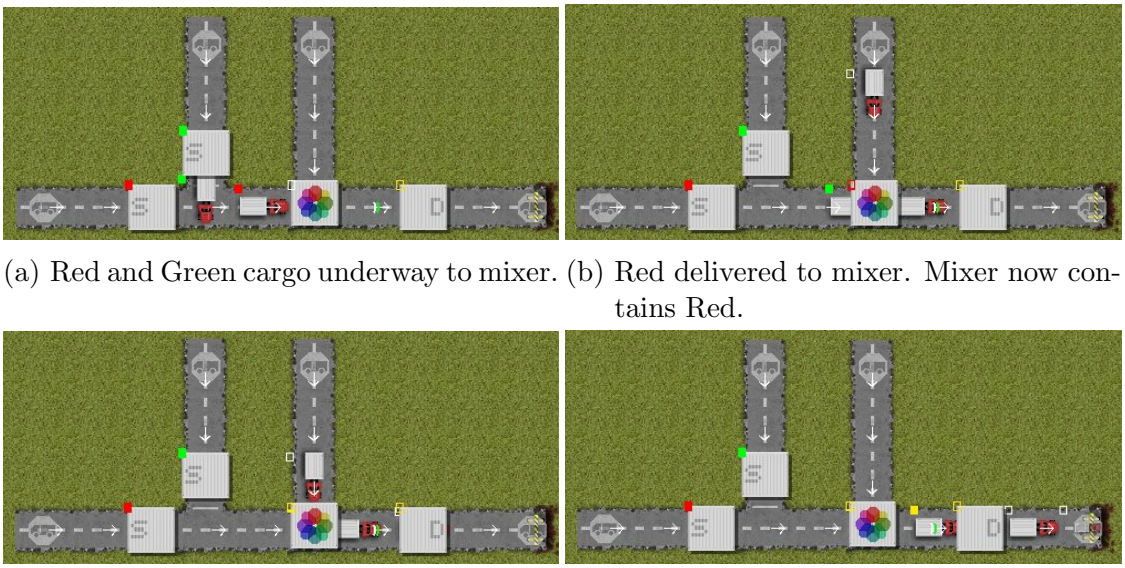


Figure 7: Empty Green depot

Mixers are buildings where cargo is mixed. All cargo dropped off at the mixers is mixed into cargo of one color. Figure 8 shows an empty mixer.

Figure 8: An empty mixer

An example of mixing is shown in Figure 9. Here, two trucks deliver Red and Green cargo. The mixer mixes this to Yellow.



(a) Red and Green cargo underway to mixer.

(b) Red delivered to mixer. Mixer now contains Red.

(c) Green delivered to mixer. Mixer now contains Yellow.

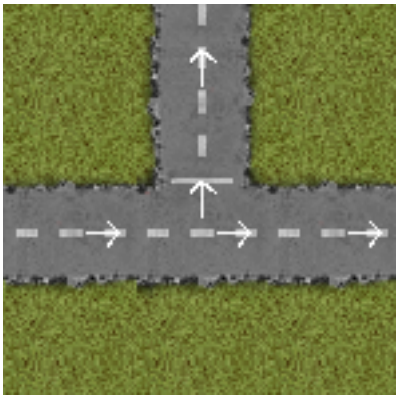
(d) Yellow is picked up from mixer.

Figure 9: Red and Green Cargo being delivered at a mixer, mixed, and picked up again as Yellow.

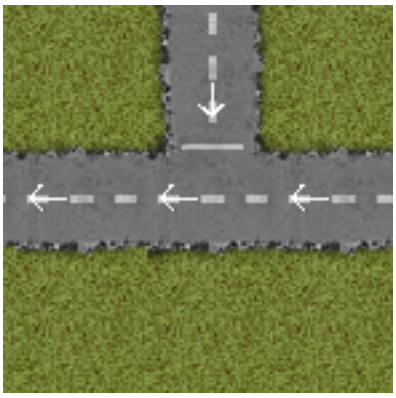
4.3.3. Roads

Roads connect adjacent cells so that vehicles can traverse between them. Roads are one-directional. This means that roads leave one cell and enter an adjacent cell. Considered from the perspective of a cell, each road is either incoming or outgoing. Figure 10 shows some examples of road configurations.

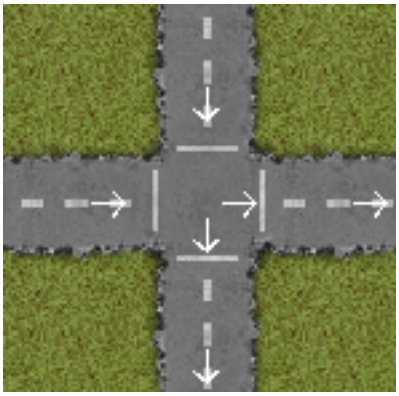
The players can build and remove roads. This is further explained in section 4.4.



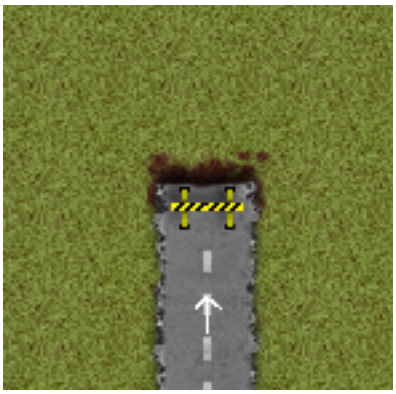
(a) incoming path from the west, outgoing paths to the north and east.



(b) Incoming paths from the north and east, outgoing path to the west.



(c) Incoming paths from the north and west, outgoing paths to the south and east.



(d) Incoming path from the south, with no outgoing paths. Dead end.

Figure 10: Road examples.

4.3.4. Restrictions

In order to prevent roads from being built everywhere, some levels also contain restricted cells. These cells are completely black, making it clear that nothing can be built there. Figure 11 shows an example of restricted cells in use.

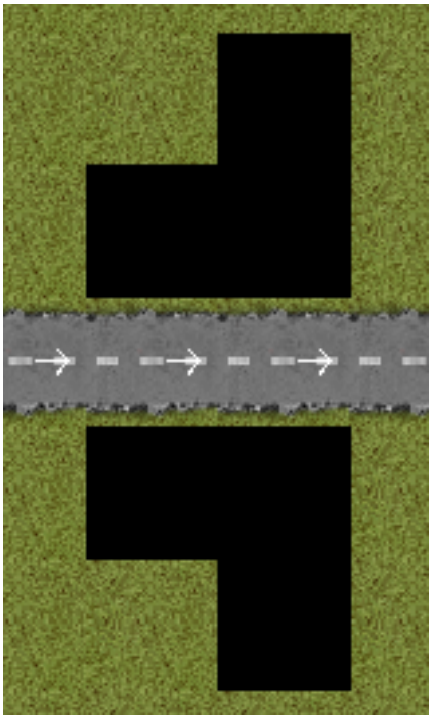


Figure 11: A road passing between some restricted cells.

4.3.5. Trucks

Trucks drive around on the road network, spawning and despawning at spawn points, and picking up and dropping off cargo.

Whenever multiple trucks enter a cell at the same time[19], both vehicles explode, as can be seen in Figure 12. When a vehicle reaches the end of a road, it also explodes.

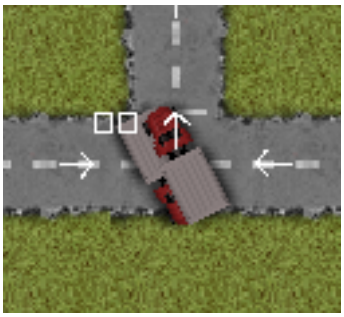


Figure 12: Two vehicles colliding

[19]in other words, when two trucks are on an incoming path for this cell at the same time

4.3.6. Routes

When there are multiple outgoing paths, trucks need to decide which one to pick. By default, trucks will pick the first path going clockwise from north. Routes are used to suppress this default behavior.

A route is a color assigned to an outgoing path. Vehicles that carry cargo with this color will pick this outgoing path, rather than the default,

Figure 13 demonstrates a route, and section 4.4 explains how the players specify routes.

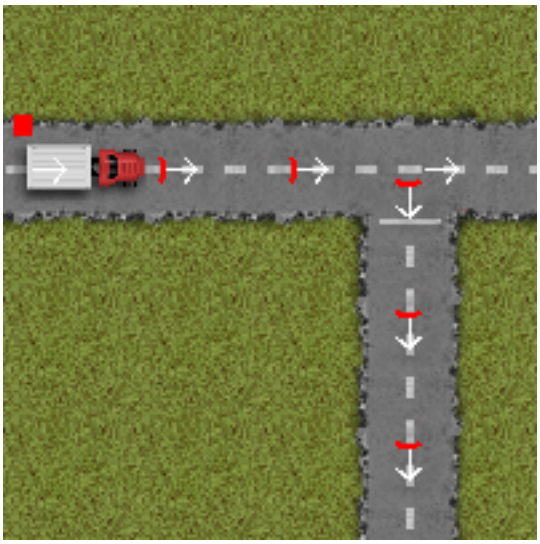
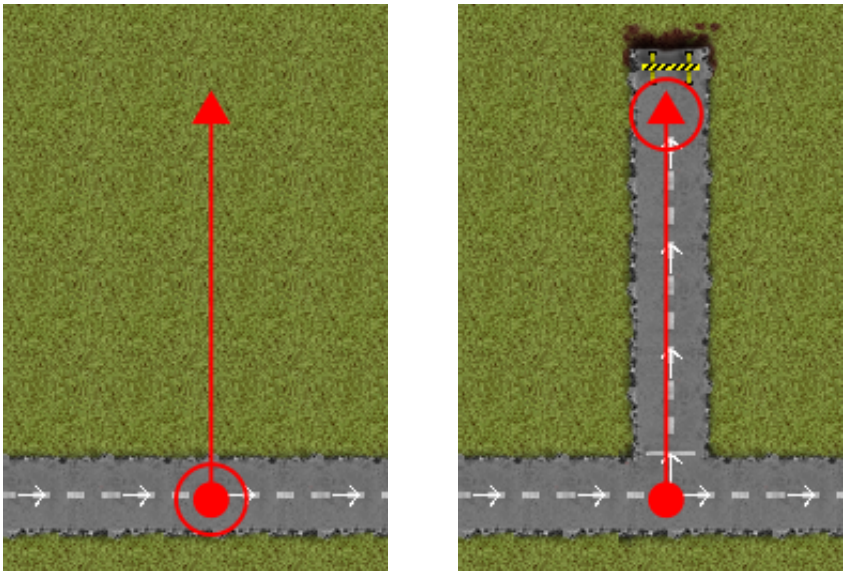


Figure 13: A Red truck following a route. This truck will go south.

4.4. Interaction

The players interact with the game by building and removing roads and specifying routes.

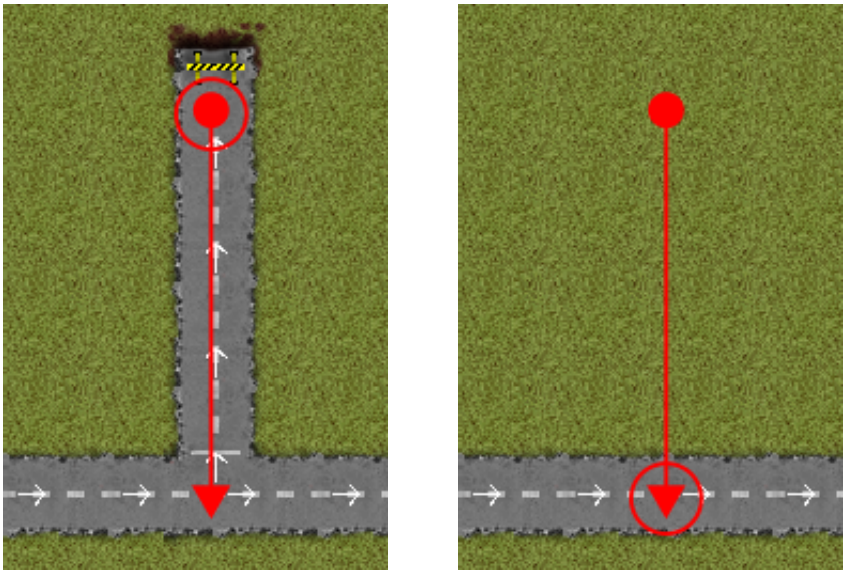
Building Roads A road is built by dragging from an already existing road towards a point where there is no road, as demonstrated in Figure 14.



(a) start of gesture (b) end of gesture

Figure 14: Road building gesture

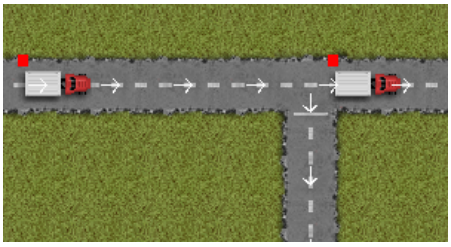
Removing Roads A road is removed by dragging over a road in the reverse direction, as demonstrated in Figure 15.



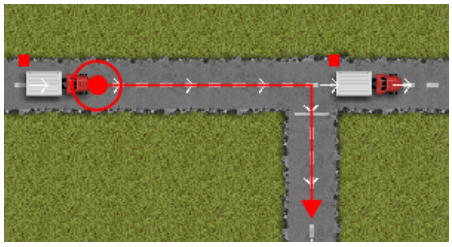
(a) start of gesture (b) end of gesture

Figure 15: Road removal gesture

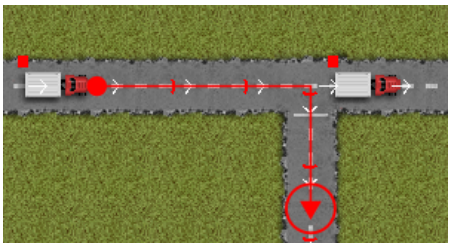
Specifying Routes A route is specified by dragging over an existing road, starting in front ofa vehicle. The route that is drawn depends on the cargo carried by the vehicle where the gesture is started. The gesture is oemonstrated in Figure 16.



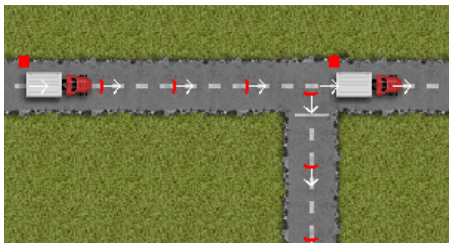
(a) Initially, Red goes east.



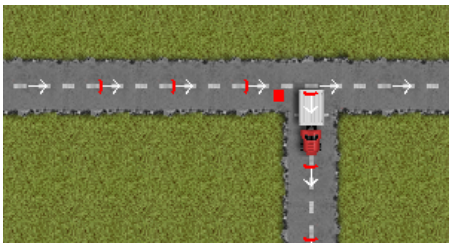
(b) Start of gesture.



(c) End of gesture.



(d) Red route displayed on the road.



(e) Red follows the route south.

Figure 16: Route drawing gesture

4.5. Level Design

The game elements described above are combined into levels of varying difficulty. Each level is 21 cells wide and 17 cells high. This size works well with the touch table, resulting in a cell size that is easy to unambiguously touch. It also ensures levels have a middle row and column, which is convenient for building symmetrical levels. Because the touch table itself is also symmetrical, both (long) sides are equal. Keeping both sides of the level equally important helps in giving every user an equal opportunity to contribute.

Timing Sometimes events need to happen close after one another, for example, to make sure the correct color is picked up from a mixer. Another timing issue is collision avoidance.
Routes from spawners with differing spawn rates might need regular attention if they need to interact.

Small Spaces Restricting the area on which roads can be build, can force the team to use every tile to the best of their ability. If multiple routes need to be taken through this area, it is likely to introduce timing issues.

Multiple feasible options When presented with a level where there are a multiple possible solution ideas, of which only 1 is valid, users may continuously second guess if whether the chosen strategy is the correct one to pursuit. Especially when faced with difficulty.

Resource sharing Sometimes roads, supplies and mixers might need to fulfill 2 or more roles in the final solution. This makes planning out a level harder.

Multiple phases Some levels might not be solvable with a single configuration. The difficulty here comes from deciding what should be done in what order, and what can be done in the same phase.

4.6. Score

In order to give some measure of how well the player did, a score mechanism is included. This mechanism works as follows:

- When a vehicle spawns, deduct 5 points.
- When a vehicle despawns by entering a spawn, add 5 points.
- When a vehicle picks up cargo at a supply, deduct 10 points.
- When a vehicle drops off cargo at a depot, add 100 points.

These scoring rules reward successful deliveries, and punish crashed vehicles and having a lot of vehicles on the map when all depots are filled and the level ends.

4.7. Art

A game is nothing without some artwork to set the right mood, and this is the same for Jest. Early on in the development process, it was decided that the visualization of Jest should invoke a feeling of familiarity in people that have ever played with the typical toy cars and a car city carpet, such as the one shown in Figure 17

Figure 17: A typical car city carpet

After instructing the artist[20] about the desired look and feel of the game, he came up with the graphics as currently implemented in the game. As can be seen in Figure 18, graphics for rails and trains were also available, but sadly the game infrastructure was not in a state where these graphics could be integrated.

[20]The hired artist for this project was Felix Akkermans.

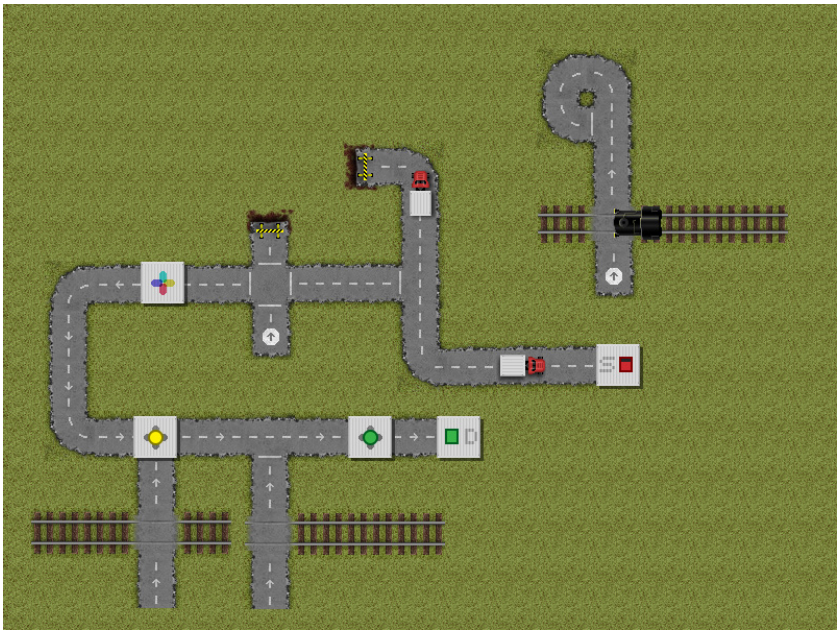


Figure 18: Collection of conceptual art

The artist also took took the concept of the game and added his own creative spins to it, creating concepts such as day and night. Figure 19 shows a visualization of a potential “night”-level.

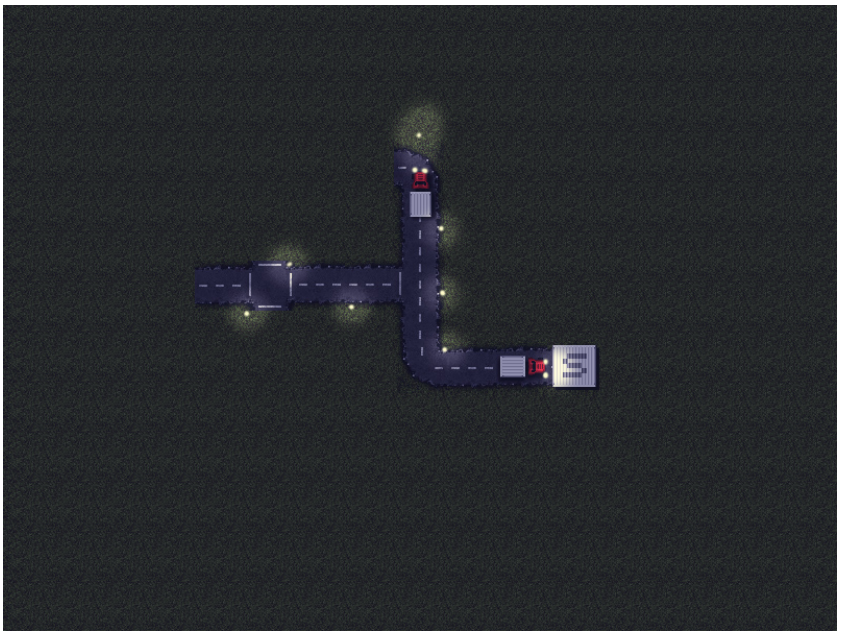


Figure 19: A conceptual night version of the game

5. System Architecture

As a software system grows in *inherent complexity*, care should be taken to ensure the *incidental complexity* and amount of dependency relations don't grow rampant. This section focuses on the decisions and corresponding rationales that have been made during the development of Jest.

To keep the system as maintainable as possible, tasks which are independent of one another have been singled out as an external library or module, as shown in Figure 20. Each component dependency relation is visualized as an arrow pointer going from the dependent to the dependee. The remainder of this section will elaborate on the high-level concepts that are involved with each part of Jest. See the orientation report in Appendix C for the intended goals of the system behind Jest.

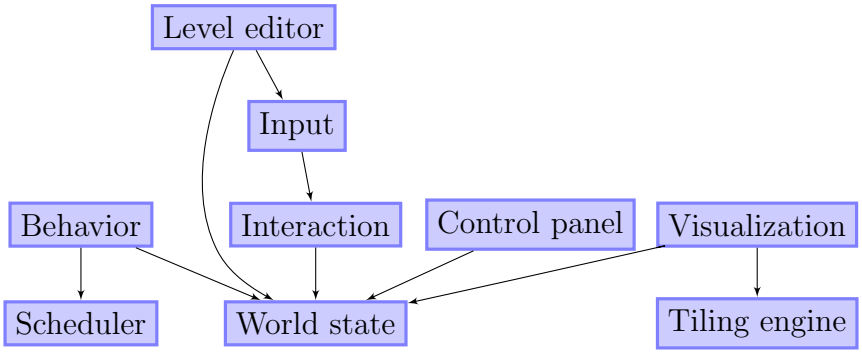


Figure 20: System components

5.1. High level engine description

This subsection focuses on the concepts on which the game engine behind Jest was based, while also explaining the rationale behind each concept as clearly as possible.

5.1.1. Asynchronous game-processes

Traditionally, game programming has been facilitated by a wide variety of system types, ranging from simple Input-Update-Draw loops such as the one shown in listing 3, to intricate systems using callbacks and even elaborate schemes describing entities comprised of several components [21].

[21] `http://unity3d.com/`

Code snippet 3: Generalized game loop pseudo code

```
while game-runs?
  handle input
  update game state
  draw graphics
end while
```

Zachary Tellman headed the development efforts for the Penumbra OpenGL bindings for Clojure[22]. One of the demos he made using an early alpha of penumbra was a functional implementation of the classic game Tetris.

Instead of using one of the earlier mentioned systems, this version of tetris was implemented using a system of periodic and scheduled execution of functions. Increasing the falling speed of a tetromino required an proportional decrease in the scheduling interval of the `move-tetromino-down` function, for example.

By treating games as a distinct set of functions, operating asynchronously on a shared world representation, new functionality can be easily added, while still keeping the advantage of being able to extend and modify existing behaviors in the game. If these asynchronous functions are kept small, a game using this scheme will also be able to scale well on platforms with multiple concurrent execution threads. A potential pitfall for these kind of systems is the fact that the world state needs to be modeled in a way so that no race conditions can occur.

5.1.2. Functional programming

There is no single, correct definition of the paradigm of Functional Programming (FP), but for the scope of this document, we subscribe to the view that a programming language which facilitate FP should at least contain functions as first-class constructs. This implies that it should be possible to represent functions as values, so they can be either passed as an argument or returned as the result of a computation for further processing or execution.

Software design patterns Being able to freely pass along functions allows for the *composition* of functions, hereby creating complex systems from a manageable set of well-understood primitives. The composability of functionality *à la carte* greatly reduces the amount of boiler plate code, thus also obviating the need for many common software design patterns[23], such as those employed when designing complex software systems in traditional Object Oriented languages, such as the Observer pattern or the Memento pattern.

[22] `https://github.com/ztellman/penumbra`
[23] `http://sourcemaking.com/design_patterns`

Pure functions By eliminating state from your computations, a function can be made referentially transparent. This means that calling the same function with the same inputs will always result in the same output. Functions that that comply with this constraint are called pure functions, and they enjoy several perks;

- Computationally intensive pure functions can be memoized[24], effectively trading an decreased processing time for increased memory consumption
- Pure functions can be tested more easily, as the result of each execution depends solely on the given inputs.
- Pure functions can be reapplied as often as necessary; subsequent runs of a function will not influence each other, even from different execution threads.

Practical effects For any program to be useful, side-effects are a necessary evil [1]. By isolating functions that manipulate mutable state from pure functions, one can still end up with a testable, well-understood set of primitives which are themselves pure, which are then wired together and called from a higher level in the system. In effect, this means that systems still can do any computation at all, with one big advantage; the basic building blocks of these systems still enjoy all the benefits mentioned in the previous paragraph. A major pitfall in this scheme is the complexity required to maintain a consistent system-state between different execution threads.

5.1.3. Concurrency Model

The game uses a collection of worker threads in order to operate on all the vehicles in the game. One thread is dedicated to rendering the current world state, with the main thread holding the world state and an optional thread for handling the control panel.

Software Transactional Model In order to provide the various threads of the game with a consistent view of the game state, a Software Transactional Memory (STM[25]) system is used. The system used is the one provided by Clojure, which implements Multiversion Concurrency Control (MCC[26]). An MCC system ensures that multiple concurrent transactions always have a internally consistent view of the world state, though it is possible for different computations to see different world states.

The STM will ensure each fragment of the world state remains as it was at the start of the transaction, unless changed by code within the transaction itself.

[24] `http://clojuredocs.org/clojure_core/clojure.core/memoize`
[25] `http://clojure.org/refs`
[26] `http://web.firebirdsql.org/doc/whitepapers/fb_vs_ibm_vs_oracle.html`

It will also try to commit altered data from completed transactions, graciously resolving any conflicts occuring due to different threads mutating their version of the same data.

With this system in use, each component of the game can pretend, within a transaction, that it is the only thread operating on data. This eliminates the need for explicit locking and makes writing multithreaded code relatively simple.

5.2. External Dependencies

Using external software can take away a lot of the workload but. Each dependency influences the design. External dependencies might help to write clearer code but might come with constrains, therefor it is important to pick compatible dependencies.

5.2.1. Quil

Quil is a thin idiomatic wrapper for Processing 1.5.1[27]. Quil also provides relatively high-level functions easing development. The wrapped version of Processing does not support hardware rendering. Initial tests indicated this is not likely to be a problem for a prototype. For more information on the decision to use Quil see chapter 7 of the orientation report included as Appendix C

5.2.2. Seesaw

Seesaw[28] is an idiomatic Clojure wrapper for building graphical user interfaces, offering a small set of abstractions over the standard Java Swing interface kit.

5.2.3. Java Native Access

JNA[29] provides JVM programs the opportunity to interact with and use native libraries, using only JVM languages. While a small performance overhead is incurred by the translation from and to the native platform, this performance hit is rarely the bottleneck in a software system.

5.3. Small Libraries

Sometimes components are so generic they might be useful outside the scope of this project. These have been implemented as separate libraries on which the game is dependent.

[27] `http://www.processing.org`
[28] `https://github.com/daveray/seesaw`
[29] `https://github.com/twall/jna`

5.3.1. The Scheduler and Scheduling

In order to make the vehicles spawn and move at the right times, a scheduler has been implemented. Other components can use the scheduler to schedule tasks at a certain future *game time*, which is the internal time representation of the game. The scheduler maps this game time to a real time and runs the task at that time. The scheduler can also be paused and resumed. When pausing, all tasks are unscheduled so that none will run, and the game time will freeze. When resuming, all tasks are rescheduled using the new mapping from game time to real time. This allows the entire game to be paused transparently from all other components.

Additionally, a mock scheduler has been implemented that uses a clock which is manually forwarded. This allowed us to write tests to ensure various game events were scheduled at the right times, which would have been much harder to do with the real-time clock.

5.3.2. Tiling engine

The visualization in the game has been split in 2 layers. At the bottom there is a tiling engine providing abstractions such as tiling and layering. The latter expresses visualization of our game in terms provided by the tiling engine.

5.3.3. Interpolation

Mainly for the movement of vehicles a small interpolation framework has been made to calculate their precise location based on the time and the route they are following.

5.4. Modules

The program has been devided into modules that of which some interact with one another. This section will go into how the project has been partitioned and how these modules interact.

5.4.1. World state

The world state module defines the data structures that are used to represent a level of the game. It also defines an API to read and modify the world state. This module ensures that the world is always in a consistent and valid state.

There are 3 data structures defined:

- Cells, which represent a particular tile location in a level. These cells contains the buildings, paths and vehicles at this location.

- Paths, which represent a vehicle path on a Cell. Paths are directional, meaning they are either incoming or outgoing. Each incoming path has a matching outgoing path in an adjacent cell, and vice versa. Each path contains information about which direction it is coming from (for incoming paths) or which direction it is going to (for outgoing paths), as well as routing information (information about which cargo should traverse this path).
- Vehicles. Each vehicle contains information about its cargo, the direction it came from, the direction it is driving towards, and entry and exit times.

At any one time, only one world is active[30]. This world is essentially a mapping from coordinates to cells, with the cells containing all the world state.

5.4.2. Behavior

The behavior module is responsible for vehicle movement. A high level overview of the flow of this module goes like follows;

1. Spawn vehicles at spawn points that are set to spawn vehicles.
2. Determine the right exit path for a vehicle. The right exit path is one with routing information that matches the current cargo. If no such path can be found, the first outgoing path going clockwise from north is selected.
3. Move vehicles between cells.
4. Execute the right actions for vehicles in various situations. These situations are:
   a) When arriving at supplies, pick up cargo (when not carrying cargo)
   b) When arriving at depots, drop off cargo (when carrying cargo)
   c) When arriving at a mixer, pick up cargo if there is any (when not carrying cargo), or drop off and mix current cargo with the color already present at the mixer (when carrying cargo)
   d) When arriving at a spawn, despawn
   e) When arriving at a cell with no outgoing paths, explode
   f) When arriving at a cell where another vehicle is already incoming, make both vehicles explode

[30]For testing purposes, a facility to temporarily construct an arbitrary sand boxed world is also available.

5.4.3. Visualization

The visualization is scheduled on a different scheduler, the one internal to quil. Every time a new frame is drawn the world state is read and mapped to an image. Because the world state is always consistent this can be done naively. This mapping can be thought of as a function of settings and the world state and can be easily replaced by a web or a (multi-)command-line text interface.

5.4.4. Input

The input module takes input events from different kinds of hardware input devices and exposes these to the rest of the game in a generic format. Currently, the game supports mouse and multi-touch input (through the windows Touch API[31]). Input events are exposed to the rest of the game as either a down, up or move event. All events have a pointer identifier, so that gesture tracking becomes possible. The down and up events have a world cell coordinate specifying at which cell the event happened. The move event has two coordinates: the cell a pointer drag started in, and an adjacent cell the pointer was dragged into. These two coordinates are guaranteed to be adjacent.

5.4.5. Interaction

The interaction module takes input events from the Input module and translates these to game actions. These actions are

- Building paths
- Destroying paths
- Creating routes

This module cleanly separates the input from the game actions, meaning any changes to the interaction model only influence code in the interaction module.

5.4.6. Level Editor

The level editor module is a drop-in replacement for the interaction module. It allows for the construction of roads just like the interaction module, but additionally it lets the user place and remove buildings by tapping a location. The building to be placed or removed is set externally, either through the developer REPL or via the control panel.

[31] `http://msdn.microsoft.com/en-us/library/windows/desktop/dd317321(v=vs.85).aspx`

5.4.7. Control Panel

The control panel is a small utility component for Jest, exposing functionality normally available through the REPL in an intuitive, window-based manner.

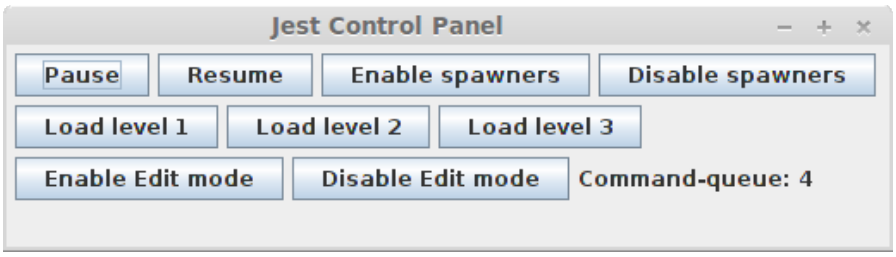


Figure 21: An example configuration of the control panel.

A non-exhaustive set of functionality available through the control panel, as shown in Figure 21;

- Loading levels from the file system.
- Pausing and resuming the game.
- Activate spawners.
- Monitor system state.
- Enable and disable the level editor.

6. Implementation

During the implementation certain things proved more complex or didn't interact as expected. Documented here are the solutions and to the encountered problems and why they work.

6.1. Input

In order to handle input in a device-agnostic way, a simple interface was designed that is able to isolate all hardware specifics to small, maintainable backends. This interface assumes that input consists of pointers that have an x and y coordinate measured in pixels, as well as a pointer ID. Three functions were defined as part of the interface that are to exposed to these backends to get input events into the game, which are;

1. `receive-down`, which takes a pointer id and the two-dimensional pixel coordinate as arguments.
2. `receive-up`, which takes a pointer id as argument.
3. `receive-move`, which takes a pointer id and the new two-dimensional pixel coordinate as arguments.

By using these primitives, the game can then operate in a device-agnostic way, and ported to a new input device in a matter of hours instead of days.

6.1.1. Mouse Input

Mouse input is the simplest form of input that the game supports. By specifying the right callbacks during the setup phase of Quil, mouse up, down and drag events are captured. Within the callbacks, the exact coordinate of the mouse is retrieved using information from the Quil window and some basic calculations. These coordinates are then passed on to the device-agnostic interface, with a fixed pointer ID of 1, since with mouse input there is only one pointer.

6.1.2. Touch Input

Unfortunately, the JVM has no native support for multi-touch. The Java Abstract Window Toolkit (AWT[32]) is the basis for all of JVM's GUI code, and therefore also the basis for Quil, only supports mouse input. If we constrained ourselves to using

[32] `http://docs.oracle.com/javase/7/docs/api/java/awt/package-summary.html`

pure JVM functionality, this would have limited the game to just one pointer.In order to make multitouch work, the native Windows API had to be used.

Windows 8 offers two different multi-touch APIs. These are WM_TOUCH and WM_POINT . Both APIs support all features required by the game (being able to track up, down and drag events for each pointer). WM_TOUCH is available from Windows 7 onwards, while WM_POINT was introduced in Windows 8, and deprecates WM_TOUCH. Since none of the developers had a Windows 8 machine available for testing, it was decided to use WM_TOUCH.

The interaction with the Windows API happens through the Java Native Access libraries (JNA[33]). Through specially-crafted classes, JNA can marshall native C structures to JVM classes, translate JVM function calls to native function calls and create JVM callback functions that can then be called from native code.

Much of the interaction with Windows happens through Window Messages. This is an inter-process communication scheme that sends a package to a message handler containing a message type and two message-specific parameters, `wParam` and `lParam`.

In order to get touch events in an AWT window, the following things have to happen:

1. Retrieve the native window handle associated with an AWT window.
2. Use the handle to register the window as a touch-enabled window.
3. Install a message hook for this window to intercept the WM_TOUCH messages.
4. Parse the message.

The Native Window Handle: Windows functions work with a native window handle. Since the game runs in the JVM inside an AWT window, this window handle is not directly available. It is hidden behind some abstractions. AWT maps each JVM component to a peer class, a Java wrapper around the native component, which can be retrieved from any AWT component using the method `getPeer()`. For Windows, each of these peers is a `sun.awt.windows.WComponentPeer`. A `WComponentPeer` has a function `getHWnd()` which returns the native pointer to the window handle as a long.

By calling `getPeer()` on our Quil applet (which extends an AWT component), casting the result to `WComponentPeer` and calling `getHWnd()` on the result, the window handle is retrieved.

[33] `http://jna.java.net/`

Register window as touch-enabled: By default, windows doesn't pass touch events to a window, but gesture events. In order to retrieve touch events, the native function `RegisterTouchWindow`[34] has to be called with the window handle and an option bitmap as arguments.

Unfortunately this function has to be called on the same thread that created the window. For AWT, this is a thread called 'AWT-Windows'. AWT uses this thread to create all its components, but it's inaccessible from pure Java.

Luckily there exists a way to run code on this thread. AWT creates a special toolkit window (class 'SunAwtToolkit', name 'theAwtToolkitWindow') and defines some custom message types that can be sent to this window to run various AWT actions on the AWT-Windows thread[35]. This is what AWT uses to create the components it needs. One of the message types (which seems to be unused by AWT itself) is `WM_AWT_INVOKE_VOID_METHOD`. This runs the void function at the memory location stored in `wParam`.

The game window can now be made touch-enabled by creating a callback function through JNA that runs `RegisterTouchWindow`, and sending a `WM_AWT_INVOKE_VOID_METHOD` message to the `SunAwtToolkit/theAwtToolkitWindow` window containing the callback as an argument. The procedure is shown in the sequence diagram in Figure 22.

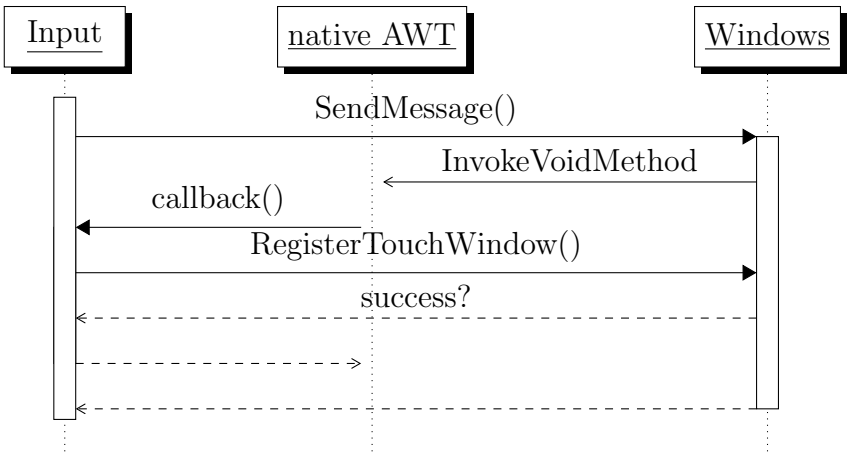


Figure 22: Register the window as touch-enabled from the AWT-Windows thread.

The Message Hook: In order to receive the touch events, a message hook needs to be installed. Hooks allow processes to intercept and handle various events

[34] `http://msdn.microsoft.com/en-us/library/windows/desktop/dd317326%28v=vs.85%29.aspx`

[35] This message handling happens in awt_Tookit.cpp, located in the openJDK7 source at jdk/src/windows/native/sun/windows/awt_Toolkit.cpp

before they reach the target window. There are various hook types, depending on the message stream that is to be intercepted and at what point in the message handling the hook should be run. In case of touch events, there seem to be two hook types that need a hook set. These are WH_GETMESSAGE and WH_CALLWND. WH_GETMESSAGE is used in Windows 7, WH_CALLWND is used in Windows 8. Therefore, to make the game work with both versions of windows, both hook types need to have a hook callback set.

Hooks can be set using the native function `SetWindowsHooksEx`[36]. This function takes a hook type, a callback function and a thread ID, which should correspond to the thread that made the AWT window. This ID is easily retrieved using the native function `GetWindowThreadProcessId`[37], which takes a window handle and returns a thread ID. Then, whenever a message arrives, the hook function will be run on the window thread.

Touch events can now be intercepted by creating a callback through JNA to handle the touch events, retrieving the thread ID of the AWT window, then setting a hook once for each hook type. This procedure is shown in the sequence diagram in Figure 23.

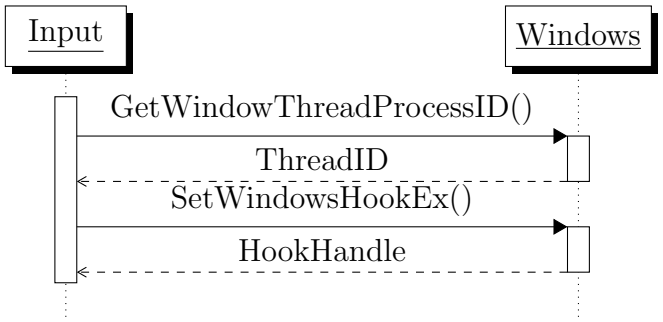


Figure 23: Setting the hook which is to receive touch events.

Parsing Touch Events: When the hooks are set up, the callback starts receiving WM_TOUCH messages. These messages contain a number specifying how many touch points are being registered, and a so-called touch input handle. By passing both the pointer count and the touch input handle to `GetTouchInputInfo`[38], a spec-

[36] `http://msdn.microsoft.com/en-us/library/windows/desktop/ms644990%28v=vs.85%29.aspx`
[37] `http://msdn.microsoft.com/en-us/library/windows/desktop/ms633522%28v=vs.85%29.aspx`
[38] `http://msdn.microsoft.com/en-us/library/windows/desktop/dd371582%28v=vs.85%29.aspx`

ified array of TOUCHINPUT[39] structures is filled with information about all the touch points. Again, JNA is used to marshall between Java classes and C structures.

Each TOUCHINPUT structure contains an x-y coordinate, a pointer ID and information about wether this event is a DOWN (someone places a finger on the table), UP (someone removes a finger) or MOVE (someone moves a finger).

The x-y coordinate is an absolute screen coordinate, rather than a coordinate relative to the window. Furthermore, it is measured in one-hundredths of a pixel. The callback converts this coordinate to a relative pixel offset by dividing both the x and y coordinate by 100, and subtracting the coordinate of the top left corner of the window.

The hook callback passes these rectified coordinates along to the rest of the game using the device-agnostic interface.

The process is shown schematically in the sequence diagram in Figure 24.

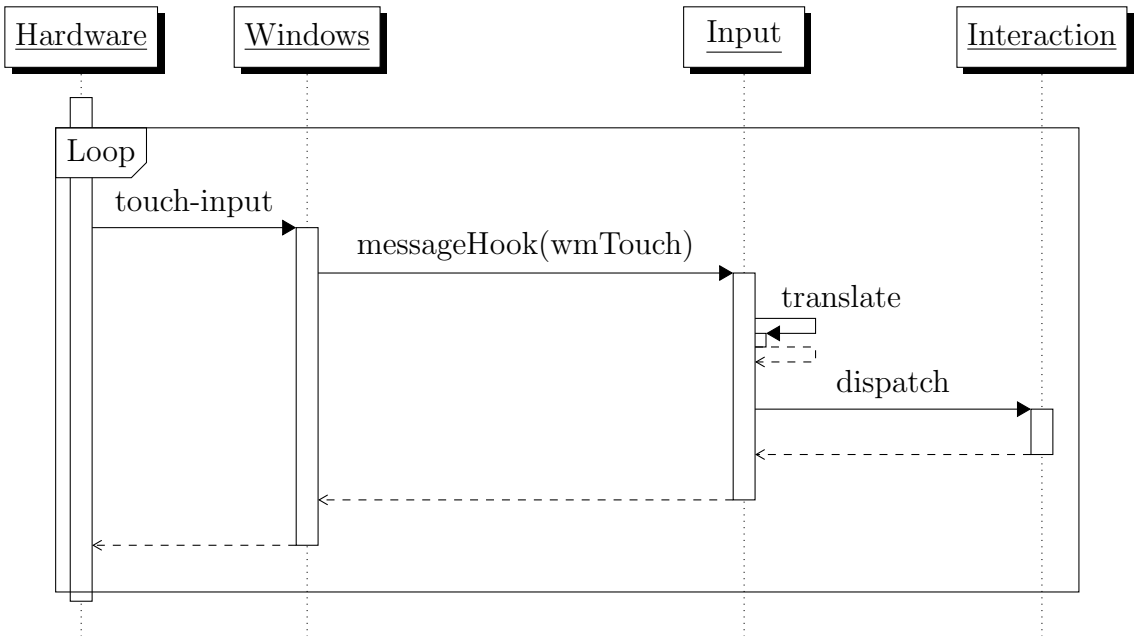


Figure 24: Touch input events are sent to the registered hook, which dispatches the touch events.

[39] http://msdn.microsoft.com/en-us/library/windows/desktop/dd317334%28v=vs.85%29.aspx

6.1.3. Processing Input

The device-agnostic interface receives input as a pair of ID and a pixel coordinate. However, for the game as a whole it is more convenient to have input in the form of an ID and a tile coordinate. Furthermore, move events are only interesting when they cross tile boundaries. The functions `receive-down`, `receive-up` and `receive-move` perform the necessary translations to in-game tiles.

Translating a pixel coordinate to a tile coordinate: The top-left and bottom-right pixels of the tile grid are known, as well as the amount of tiles in both the x and y direction. From this the width and height of individual tiles can be calculated. Therefore, given a pixel coordinate, its matching tile can be calculated by subtracting the top-left corner coordinate, then integer dividing both the x and y remainder by the tile width and height. The resulting two integers form a tile coordinate. The formula for this is displayed in Figure 25.

$$[width_{tile}, height_{tile}] = \left[\frac{width_{grid}}{amount_x}, \frac{height_{grid}}{amount_y}\right]$$

$$[x_{tile}, y_{tile}] = \left[\left\lfloor\frac{x_{pixel} - x_{corner}}{width_{tile}}\right\rfloor, \left\lfloor\frac{y_{pixel} - y_{corner}}{height_{tile}}\right\rfloor\right]$$

Figure 25: pixel to tile coordinate translation.

Coalescing movement input: By saving the tile coordinate for each pointer upon receiving an event, each down-event can check if its significant (which it is if it crosses a tile boundary) by checking if its tile coordinate is different from the tile coordinate previously recorded for this pointer. If it is, the move event can be propagated to the rest of the game.

Depending on the sampling speed of the input device, the tile coordinate registered in a move event might actually be further than one tile away from the previous tile. In order to guarantee consistent behavior to the game, it is desirable to interpolate a movement path between the previously registered tile and the current one. The input module interpolates this path by generating a sequence of artificial movement events by first simulating pointer-movements along the x-axis, after which the remaining movements on the y-axis are simulated.

Interaction callbacks: Depending on the mode the game is in, the input has to be handled in a different way. In order to facilitate various interaction procedures, a

callback system has been set up. A component that wishes to receive input has to call `set-input-handler!` to register its callbacks for up, down and move events[40]

6.2. Tiling Engine

A tiled image can be thought of as multiple layers of sprites placed in a equally sized grid. Vehicles are placed using a more manual mechanic. Note however that there are other compositions of these elements with valid use cases. It might be useful to decorate a sprite with a semi transparent layer of rust. This could be expressed as 2 layers of sprites where the rust sprite is possibly modified for added transparency.

When these 2 layers of sprites are allowed to be used as a drop-in replacement for a single sprites, using such a system would indeed be easy. Observing the following:

- All these elements, sprites, layers, grids and transparency need to be drawn
- Layering and grids are composition methods
- A sprite is a primitive
- transparency is a modifier

These facts led to the interface `Drawable` as seen in 4, making it easier to implement the aformentioned abstractions of which some are listed in Listing 5. This interface also serves as an extension point for other libraries.

Code snippet 4: The Drawable protocol

```
(defprotocol Drawable
  "Anything that can be drawn"
  (draw [this [w h]]
    "Draw the given object w times h pixels large."))
```

Code snippet 5: Headers of some of the implementations in pseudo code.

```
(defrecord Stack
  [layers]
  "Multiple drawables drawn on top of one another in the same area
     .
  This is only useful when images are (partially) transparent.")

(defrecord Grid
```

[40] A component has to revert the changes to the callbacks if for whatever reason the component is no longer activated.

```
  [w h grid]
  "Draw a w times h grid of the drawables in grid, which should be
  key-value pairs with a key [x y] drawn at position (x, y).")


(defrecord Image
  #^{:doc "A drawable wrapper for a Proccessing Image (Pimage)."}
  [img])

(defrecord Nothing
  #^{:doc "A placeholder. Doesn't draw."}
  [])
```

By composing `Drawable`s instead of sprites, `Drawable`s can be nested arbitrarily. Figure 26 is an example of such a composition. On the bottom of the stack is a

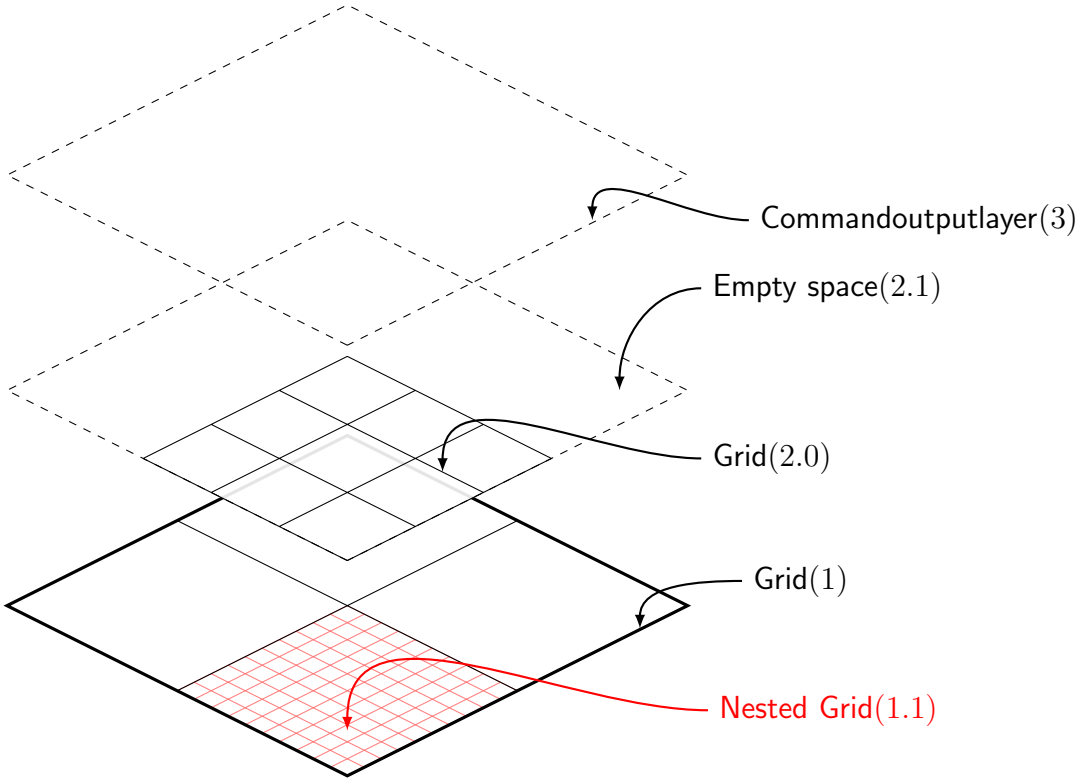


Figure 26: An example configuration of a (root-) stack

2 by 2 Grid (1), with a Grid nested in one of its cells, with a 5 by 5 grid on top of it that is only partially filled in (2.0). The rest of the cells are willed in with `Nothing` space created by filling the empty cells `Nothing`. The top layer (3) is more conceptual and will be explained later in this section.
Every `Drawable` can be fed to Quil to `drawable->sketch!` With the default settings

this will open a new JFrame and draws the passed `Drawable` repeatedly, this is called a sketch. A sketch can be sent commands, which will be ran after the next draw cycle. If this command results in graphical output, this will be done on top of the previously drawn frame which can be seen as extra layers as seen in Figure 26(3). This mechanism can also be used to query information which can only be retrieved from within a draw context.
To pass Quil more settings wrap a `Drawable` in a `Bricklet` containing the settings[41]. Settings could include size, frame-rate or which target renderer, if any, should be used.

6.3. Resource loading

Resources such as tiles should be put in the `resources/` folder, which is located in the root of the project, or in case of a redistributable binary, in the jar-file.

Tiles can be either simple single png-files, in which case one file contains exactly one tile, or multiple tiles can be concatenated in one big png-file to simplify loading big tile sets at once.

Loading the tiles is done by calling the `quil.core/load-image` function, which takes the path to valid png-file. Concatenated images are then cut up into sub images of 48 by 48 pixels and put into a dictionary to simplify access to each sub image.

6.4. Performance

In the course of developing Jest, some performance problems were encountered. Whereas most of these problems were simply bugs or resource-leaks[42], the biggest one was actually a direct consequence of using Quil.

jvisualvm To determine which optimizations would improve the performance of Jest, an objective measuring tool was required. Luckily, jvisualvm[43] fullfilled this need. By connecting jvisualvm to a running instance of Jest and checking the Sampling-option, a table with relative CPU residence time for each function could be drawn up. The latest of these hotspot charts can be seen in Figure 27, with the top four hotspots all belonging to Quil implementation functions. It can be deduced that Quil alone contributes to around 67% of processing time, thus reducing the need for further optimizations in other parts of the code. Alternative approaches to improve the performance of Jest will be explored in Section 10.

[41] The given settings will be augmented with the defaults if they are missing.
[42] Check out the repository logs for more information
[43] `http://docs.oracle.com/javase/6/docs/technotes/tools/share/jvisualvm.html`

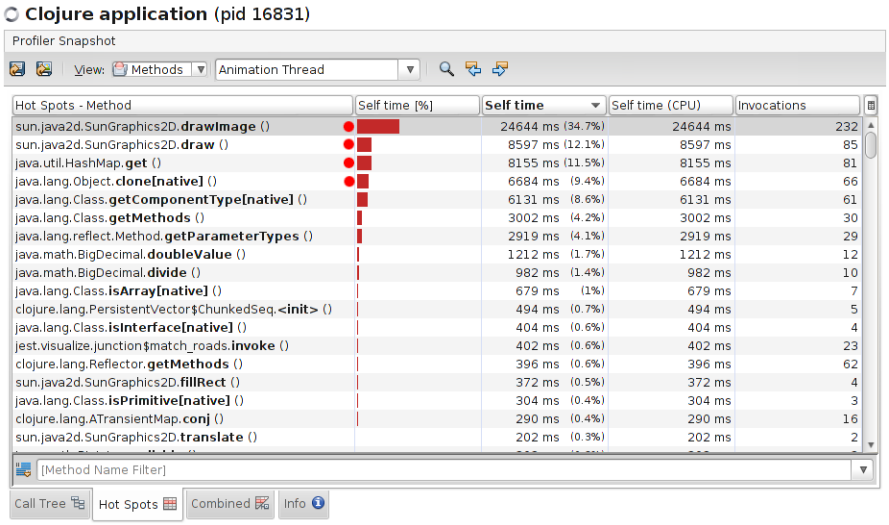


Figure 27: jvisualvm showing of the CPU hotspots of Jest. Quil related hotspots are marked with a red dot.

Recent performance To give an overview of the performance characteristics of Jest;

- On a normal level, Jest is rendered at around 15 frames per second.
- On a crowded level, either with roads or vehicles, the rendering of Jest happens at around 9 frames per second.
- The game always runs at the same speed, since input handling and game logic happen in their own threads[44]

[44]jvisualvm showed that the non-rendering threads use about 6% of their allotted CPU timeslot.

7. Feedback

In the course of developing Jest, several groups have expressed their opinions about the quality of and experience with the system. In this section, the most relevant pieces of feedback will be summarized, followed by an explanation of how feedback was processed.

7.1. Software Improvement Group

The first group that evaluated Jest would be the Software Improvement Group (SIG[45]). The source code[46] for both Jest and the custom made tiling engine were evaluated by SIG. See Appendix B for a verbatim copy of the feedback.

Processing Not a lot has been changed in the code-base in response to the SIG evaluation, because the feedback was rather concise. One of the few potential problems SIG identified was the code duplication in some of the test code. Since then, this code has not changed.

7.2. User Tests

Several user tests have been performed, each during a different stage of completion of the game. Some of these tests produced more useful feedback than others, and this subsection aims to elaborate on the most useful user tests.

7.2.1. Ut

Number of test persons	1
Type of test person(s)	First timers

This user test did not produce any particularly useful feedback. This was because the visualization component of the project was still in its infancy, which also led to the user complaining about not knowing what's going on in the game. The lesson learned would be that we needed better graphics.

7.2.2. Ut

Number of test persons	2
Type of test person(s)	First timers

This user test led to some interesting observations with regards to the quality of the game. Users commented on the fact that the gamed seemed somewhat

[45] `http://www.sig.eu/en`
[46] As it was on

sluggish in bigger levels, even slowing down to a crawl when a particularly large set of vehicles was spawned simultaneously. After a short explanation, most of the core game play elements were received with praise. At this point in time, the game did not yet have a goal, so a more thorough user test would be needed in the future. Several performance optimizations were made that partially mitigated the problems exposed in this user test.

7.2.3. Ut

Number of test persons	3
Type of test person(s)	First timers

This was the first real user test, in the sense that the game was nearing completion. Users were made to complete several of the tutorial levels already present in the game.

Users reported that the tutorial levels were well thought out, although some of intended solutions to levels were never found by users in this session. Interaction with regards to the manipulation of routing information was considered a hassle, requiring precise timing and almost no room for small mistakes. Some concerns were voiced about the placement of vehicle and the way the contents of buildings was visualized, but the choice was made to leave this piece as advice for a future enhancement to the game.

7.2.4. Ut

Number of test persons	2
Type of test person(s)	One first timer and one experienced user

This user test featured improved interaction for the manipulation of routing information. The user test showed that manipulating routes had become much easier than it was before, though an explanation of how to specify a route was still required. It was also noticed that users didn't seem to understand that vehicles need to be despawned in order to maximize the score a player achieves. Some advice was given w.r.t. the meaning of each graphic; either use whole words to spell out what the purpose of each building is, or show a legend at the side of the game explaining what each graphic represents. As an added bonus, this legend could also display the basic pictorial gesture instructions to interact with the game world.

Users confirmed that the constant buzzing activity in the game world looks inviting to someone glancing over the touch table, though the spawn rate in certain levels should be lowered to decrease the dizzying effects that can arise from lots of

vehicles looping around in a tight spot.

Some miscellaneous feedback included the fact that large swats of grass looked a tad boring. Some variation in the grass tiles, or perhaps some other sort of background would help to break up the illusion of vast expanses of grass. This was, again, sorted into the group of *nice-to-have* features, and put up as a potential future enhancement

8. Product Evaluation

This section aims to evaluate the deliverables, as defined in Appendix C, against the requirements and constraints defined in Section 2.

Clojure TUIO client As Windows 8 does not support any of the TUIO translation layers, a client library for Clojure was no longer necessary. Thus, this deliverable never made it out of the prototyping phase.

2D Graphics engine A general purpose 2d graphics engine has been created in the form of Brick. The first prototype of Brick was finished after about four weeks of work. Extra modifications and performance enhancement were added when this was deemed necessary. The overhead of using Brick instead of directly calling the renderer (Quil) is negligible. As Brick's only dependency is the renderer, Brick can run on any platform that is able to host the renderer.

Asynchronous game engine The game engine is currently comprised of two parts, one being the scheduler as presented in Section 6, the other being the concurrency features of Clojure as mentioned in Appendix C. With the scheduler being finished in week 5, and the concurrency features being available from the start, one can say the asynchronous game engine was in a functional state at the end of week 5.

Jest Jest is the aggregate of all other parts of the system. Both the game engine and 2D graphics engine were used as libraries in Jest, with Jest being finished at the end of week 18. Originally, boats and trains were also supposed to be included as transport vehicles, but due to time constraints this was deemed unfeasible in the later stages of the development process. Jest currently runs on any platform that is supported by Quil, including the GD touch screen table and generic GNU/Linux setups like those used by the Jest developers.

9. Process Reflection

This section focuses on the development process of Jest, or to be more specific; the things that went well and things that could have gone better.

Planning Planning was the biggest issue in this project. Due to a lack of experience with game development and touch screen interaction, time estimations for most tasks were grossly optimistic. When the first few tasks took longer than expected, the planning wasn't updated to reflect this fact, which in effect made the planning not that useful. A big help would have been a more predefined assignment, meaning both more strict requirements from GD and measurable criteria for the Bachelor Project.

Test Driven Development Jest was the first big project for all developers where a test driven development (TDD) workflow was used. In the early stages of the project, many of the benefits of TDD were taken for granted. Especially refactoring existing code was a breeze when an automated test suite would show any potential errors in the new code. In the later stages of development, tests were usually written after the implementation of a feature, thereby losing many of the advantages of TDD while still requiring a lot of time. For projects such as Jest, enforcing TDD would pay off in the long run.

Project Planning Tools In the course of developing Jest, several project planning tools were used. The settled upon tool was a card-and-board-based system, displaying several characteristics such as the ability to easily appoint cards to people and recategorize them when needed.

Communication Communication with both the coaches and contact persons at GD went well in general, although some events could have been handled better from both ends, such as identifying potential problems in the planning and communicating them to all parties involved.

Workflow The interface of several system components would have stabilized a lot faster if pair programming was the norm rather than an exception.

Field testing GD was very lenient with granting access to the iLab for testing and development purposes, which was very useful for finding and fixing bugs that were only observable when Jest was running on the touch screen table.

10. Future Recommendations

A project is almost never truly done, there is always something which can be polished further given more time, computational power or other resources. This section provides some suggestion as to what could be interesting improvements on Jest.

10.1. Rendering

As mentioned in Section 6.4, Quil is by far the most significant bottleneck. The use of a hardware accelerated render warrants investigation, as performance is likely to increase if rendering is no longer performed on the CPU. If tests for other possibilities turn out promising, Brick can be ported to the new renderer. Jest itself contains a few direct calls to the Quil library, but these are all contained in the visualization component, and as such can also be easily ported.

10.2. Hardware

The touch table in the iLab detects touch by means of IR light. Because the surface is both prostrate and meant to interacted with dust and other unwanted objects collect gradually around the sensors, jamming their signals. A touch screen based on capacitive technology would not suffer from this[47], just as won't react to sleeves dragging over the screen.

The iLab is equipped with both colored lighting and a sound system. Sadly, neither of these were accessable through the computer the game runs on. Background music and sound effects could make the game more immersive and inviting. The lighting system could be used in much the same way as sound effects, highlighting certain events in the game such as winning, pause or explosions.

Because the 2d graphics engine and the input module are the only hardware-dependent system components, Jest could be ported to new hardware very easily, such as the Android platform.

10.3. GUI

The current GUI has been made somewhat as an afterthought. It shows up in a separate window and is not much to look at. In a future incarnation of the interface it is worth looking into the possibility of making a GUI on top of the

[47] Dust would still gather on the edges of the screen.

rendering engine.
A separate interface for the level editor might also be useful. For example, buttons to select a building type, and if applicable, a color. The color selection could be done by having a button for every primary color and clicking them will add 1 unit thereof to the color being used. This corresponds to how colors are mixed in the game itself.

10.4. Controls/Gestures

Giving the user more feedback on what his/her gesture is going to do might improve the user experience. For example, if the user is trying to change a route, highlighting the vehicle whose color will be routed by the gesture and showing a colored pointer at the place of the finger.

10.5. Gameplay

The gameplay as described in 4 can be expanded on with a plethora of new mechanics and variations of old gameplay elements. The rest of this section will cover a number of unexplored gameplay ideas.

Vehicle Types The game was originally planned with not only trucks but also boats and trains in mind. Most of the code has been written with this in mind. Each type of vehicle would have it's own constraints. For example, boats were planned to move over rivers that the player could not add or remove but were able to carry more resources than trucks at once.

Spawner Mode Currently a spawner can either regularly spawn vehicles or not at all. Additional configurations might be spawning on demand, or the option for the user to change the spawn frequency.

Terrain Different terrain types with their own constraints can introduce an extra challenging dimension. Adding new textures will also create a more lively visualization of the world, making the game look more appealing to potential users.

Score Model The score system, as implemented at the moment, does not completely have the intended effect. The primary goal of having a score system is to push the players to find an efficient solution. By taking a good look at what optimal generally means[48] and have the score system reward actions that are in

[48] Some examples of what can be seen as virtues are short roads, little to no unwanted deliveries in depots, as few crashes as possible.

line with these ideas, players might be inclined to redesign their solutions to be closer to the maximum score possible and compete with one another. The scores can be combined with a high score list. Levels will likely have an optimal score and it might be a good idea not to show the actual score in the high score list in favor the percentage of scores are lower or equal.
Levels have to be designed with the same concept of optimal in mind for this have the wanted effect.

Local traffic As an extra challenge, extra traffic could spawn from certain areas and independently drive to it's destination. The players will have to guide the trucks around this local traffic.

Limited Resources Restricting the resources for a level can add some extra difficulty to the game. Examples of resources that can be limited are roads, vehicles, resources from supplies or perhaps a time limit.

11. Conclusion

The goal of the project was to make an innovative multi-touch application for the touch screen table at the iLab. We decided to do so by building a collaborative game controlled through touch events, Jest

In the process of developing the game, we ran into various challenges. Our initial approach for getting multi-touch to work, TUIO, did not work, so we had to work around several issues to get the JVM to play nicely with the Windows multi touch API. What follows is an evaluation of each deliverable.

The Clojure TUIO client was deemed unnecessary after the orientation phase and thus never finished.

The 2D graphics engine was created by piling a tiling abstraction layer on top of a renderer and is considered a success, as it is currently in use in the Jest application.

The asynchronous game engine was a lot simpler to implement than originally planned, while still fulfilling all of our needs, making this deliverable a success too.

Jest is the sum of every other deliverable and the art assets as delivered by the hired artist. The user experience was tweaked with feedback collected from several user tests. In the end, we'd like to go out on a limb and state the end product was a success.

Given the opportunity to improve our game, we'd switch to another rendering engine, as this is the major bottleneck of Jest. We would also improve on several gameplay aspects by changing the scoring mechanism, provide visual feedback cues when interacting with the game and give the GUI the final finish it deserves.

Both this report and the source code are a valuable asset for developers interested in enhancing Jest, creating multi touch games for Windows or thinking about game development in new ways. We think our work paves the way for interesting future developments.

List of Figures

A. Planning

week	focus	tasks
1	• Orientation game design • Orientation game programming • Orientation game concept	• Work 2 on game concept • Research existing game engines • Research game architectures • Research the use of multi-touch or single screen multi-player in games • Research multi-touch on Windows 8 and gestures • Prepare for hardware tests • Plan a meeting with supervisors
2	• Orientation report • Defining game play • The engine pipeline • Scoping assignment	• Draft report • Explore JME • Explore Quil • Plan a meeting with coaches concerning gameplay • Prepare meeting with municipality • Prepare clj-tuio tests for unknown target hardware • Write report • Review report content/writing • Review report for consistency
3	• Prototype engine components	• Prototype world state • Prototype visualization • Prototype interaction
4	• Finish Engine components	• Implement and test world state • Implement and test vehicles • Implement and test Tiling engine

5	• Visualization • Implement game mechanics	• Draw the world state • Draw vehicles naively • Implement user actions as functions • Load level from Tiled format
6	• No significant activity due to health issues	
7	• No significant activity due to health issues	
8	• Interpolate vehicle locations • SIG • User-test	• Interpolate library • Visualize paths • Prepare user-test • user-test
9	• Continue interpolation library • colors	• Debug interpolation library • Replace old vehicle movement mechanism • Bikeshed • Eastwood • Kibit • Codox • Refactor
10	• New graphics • Input	• New road visualization mechanism • Input processing mechanism

11	• Performance • New road graphics • Score • Interaction	• Profiling • Optimize • Implement score system • fire score events • Balance scores • Interaction mechanism
12	• Interaction • Borders	• Fine-tune interaction • Undecorate window • Refactor colors • (De)spawn animations
13	• Completeness	• GUI • Smoothen vehicle • Game life cycle • Level editor • Balance scores
14	• Levels • Level editor • User test	• Create tutorial levels • Level editor • Auto spawn • Load and save levels • Prepare user test(reservation, invite, setup • User test
15	• Holiday	

16	• SIG • Report	• clean up code • Minor refactors • Report outline • Document Design • Document Implementation
17	• SIG • Report	• Kibit • Bikeshed • Eastwood • Docstrings • Send in code
18	• Presentation	• Prepare presentation • Practice presentation • Give presentation

B. Feedback SIG

The Software Improvement Group had this to say about the code-base on (in Dutch);

> Het eerste wat bij jullie project opvalt is de keuze voor Clojure. We hebben als SIG in de afgelopen 13 jaar ongeveer duizend systemen gezien, maar daarvan waren er precies 0 in Clojure geschreven. Natuurlijk is de taal nog vrij nieuw, dus waarschijnlijk gaan we hem in de toekomst vaker tegenkomen, maar vooralsnog is het gevolg dat we geen automatische analyse voor Clojure hebben. Ik heb jullie code daarom handmatig bekeken.
>
> De code-indeling ziet er goed uit. De scheiding tussen de twee projecten is duidelijk, en beide projecten gebruiken dezelfde duidelijke directorystructuur. De component-indeling van de code zelf is iets minder duidelijk. Bij "jest" begrijp ik niet waarom "world.clj" niet ook in de directory "world" staat. Het kan zijn dat daar een goede reden voor is, maar de naamgeving helpt in ieder geval niet om dit duidelijk te maken.
>
> De code bestaat voor het grootste gedeelte uit kleine methodes die een overzichtelijke hoeveelheid werk uitvoeren. Het enige punt van zorg is dat er wat duplicatie in zit. Deze duplicatie zit voornamelijk in de tests, bijvoorbeeld `movement_test.clj`. Dat is minder erg dan duplicatie in productiecode, maar ga hier niet te ver mee, want je moet deze code nog steeds onderhouden.
>
> Dat er uberhaupt tests zijn is natuurlijk een plus. Hopelijk lukt het jullie om de verhouding tussen productiecode en testcode vast te houden als het project straks gaat groeien.

C. Original orientation report

Traffic orientation report

Matthijs van Otterdijk
Ferdy Moon Soo Beekmans
Jelle Licht

Contents

1 Introduction

This document will describe our findings during preparations of this bachelor project. During this phase we aimed to learn how to use the technologies we will need to accomplish our goals. The big themes have been touch, game design, game and graphics engines and planning and defining the process. A large obstacle has been delayed contact with the municipality. A lot of choices, including what game to make was dependent on their feedback. Only after having had a meeting could we finalize the game concept and make some important decisions.

The most important parts of this document are Tool Selection, where we give a brief overview of our decisions, and Game Architecture, where we detail our approach towards building this game.

We have also added an appendix containing a description of the game that we'll implement. However, many of the details of this game might change later on in the process due to feedback from the client and tests.

2 Gemeente Delft

The Gemeente Delft is building a new lab, the iLab. The aim of this laboratory is to combine technology and creativity in order to improve everyday life. Plans for the lab contain a large touch-screen table. For this table, Gemeente Delft wants to have a game that stimulates cooperation and/or competition between it's players. As a side goal, the game is to be somewhat educational, but lowering the entry barrier for the iLab is the primary goal. Gemeente Delft wants a small group of students to make a functioning prototype in the course of two months.

3 Deliverables

In order to fulfill the given assignment and achieve our goals, certain artifacts need to be in a functional state at the end of this project.

To be reasonably sure an artifact is in a functional state, the artifact must be fully unit tested where possible. The goal is to deliver the following artifacts:

- *Traffic game*: The Traffic game is the main artifact. The game will consist of game logic on top of the other artifacts.
- *Asynchronous game engine*: As part of this project a game engine is needed. The engine's main goal is to keep the state of the game world and schedule asynchronous events that may alter the state of the game.
- *2d Graphics engine*: To visualize the game a 2d graphics engine is needed with layer support. This graphics engine will be built on top of quil.
- *Clojure TUIO client library*: For communication with touch screens the lightweight protocol TUIO is used. To make this accessible for other Clojure programmers a library to make TUIO clients will be released separately.
- *Report*: To document every decision that has been made, and the rationale behind each one.

4 Case Study: Polar Defense

In a study by [Finke et al., 2008] about gaming interfaces in a public space, the different interactions between passersby, spectators and active players and the game itself was studied. Over the course of four days on the campus of the University of British Columbia (UBC), data regarding these interactions was gathered.

4.1 Users

[Finke et al., 2008] describes a model for categorizing users by their relation with the game. Users can be divided into 3 groups: actors, spectators and bystanders. Users generally transition, if at all, from being a bystander to being a spectator and from spectator to actor. Different substages are described. The bystander will first enter the room and then glance at the installation. A spectator will decode and observe the game where actors are in a loop of giving input and recieving feedback. This input results in the state of the game. Guiding these transitions between the various stages will speed up the process.

4.2 Interaction

It is also pointed out that textual instructions are likely to be (partially) ignored and that information is mostly gathered from other actors. This implies that a game can easily become over- complicated. A game on a touch table will be easier to pick up since all the actors are gathered directly around the table itself and all their actions are visible to other users. Because bystanders can see both the actions and the results thereof at the same time, the decoding of actions to consequences should be slightly easier.

4.3 Location

The game that was used for the study is played in a large public space on a projector screen, visible from the door, serving as an eye-catcher. The eye-catcher serves to maximize the influx of bystanders, though this will not be a possibility for our prototype. The room and devices are a given, with the table stationed in the lounge of an office. This will make it harder to attract bystanders, but this effect can possibly be mitigated by using the available lighting and furniture in a way that focuses people's attention on the touch screen table.

5 Hardware

The Hardware made available by Gemeente Delft is a PC (HP Z400) with 12 GB memory and a Quadro video card, a large television that comes with a touch overlay panel, the Samsung TM55LBC, that detects up to 6 points at the same time.

The television is embedded in the surface of a large table, with restricted access to the screen at the short sides of the table.

Because the touchscreen is driven by infra-red (IR), the screen is also sensitive to hovering slightly above the screen. In combination with the horizontal orientation of the screen, this causes a noticeable discrepancy between where a user pushes and where the input is registered by the hardware. Since touch input is registered before the user touches the screen, the user can get visual feedback before tactile feedback. The application will have to be designed with these quirks in mind.

6 Language and Platform

Clojure is a modern lisp that was made to run on a host platform and to have built-in support for dealing with concurrency [Hickey, 2013].

6.1 A guest on the host platform

Clojure currently supports the following host platforms:

- Java Virtual Machine (JVM)
- Common Language Runtime (CLR)
- JavaScript

Clojure allows the developer to call functions from the host language as if they were Clojure functions, allowing easy interoperability. However, even though functions can be called easily, the idioms of Clojure differ significantly from the idioms of its host platforms. For this reason many Clojure libraries exist which wrap functions of the host platform.

6.2 Easy concurrency primitives and immutable state

To achieve good concurrency support, Clojure follows the functional programming paradigm where possible. Functional programming here means programming without side-effects (The result of a function directly follows from the input with no other effects). The absence of side-effects makes it easier to test functions, as the result is only influenced by it's input, therefore less situations need to be tested and there is no need for mocks (in the functional parts). To be able to work this way, most state in Clojure is immutable unless stated otherwise. Instead of changing the state, a function will often return the new state. If this state is a composite, all the values that stay the same will be memory shared with the old state.

For state and concurrency management Clojure offers 4 different data containers.

- *Vars*: comparable to pointers A var points to a piece of state and can be pointed towards another piece of state in a certain thread, changes are isolated to a single thread.
- *Refs*: can only be changed within a transaction. This ensures that multiple refs will either be changed together or not at all. Doing so guarantees that the state will always be consistent.
- *Agents*: are used to store state that can be updated asynchronously by sending the agent a function to update itself. This is a useful model for entities that operate autonomously.
- *Atoms*: offer support for independent state and synchronous changes to it.

Despite its functional character, Clojure also supports polymorphism, a feature often used in Object Oriented languages. Polymorphism is used to specialize behavior of objects at runtime. This mechanic enables the programmer to use best of both the functional programming (FP) and object oriented programming (OOP) worlds.

6.2.1 The Read Evaluate Print Loop (REPL)

Clojure development is usually done with an open REPL, which is an interactive process which accepts (reads) Clojure code forms, evaluates these forms and then prints the result. This allows the live modification of a running program. The availability of a REPL gives the developer instant feedback on small code snippets. This way of developing is called REPL-based development.

6.3 Platform

Picking a platform is a very influential decision for any project. The platform can constrain an application in certain ways as different libraries are available for different platforms. The advantages and disadvantages will here be described and compared with one another.

6.4 Considerations

The following is a detailed list of our considerations by which we compared the different environments.

- *Ease of development*: In the short development phase for this project, it is important to quickly get a working prototype up and running. The platform needs to be mature enough to comfortably develop in and have sufficient debugging options. Furthermore, building must have reliable and reproducable results.
- *Performance*: The platform must not significantly limit us in graphics and gameplay options.
- *Ecosystem*: The platform may offer or lack important dependencies that are required for the development of the traffic game. Important dependencies are support for rendering, audio and touch input.
- *Prior knowledge*: Prior experience is a big factor in accurately estimating development speed.
- *Target environment of the platform*: The target environment needs to be suitable for the game and should not limit the user experience.

6.5 The CLR

- On the CLR, Clojure does not provide dependency management tools, a build system or a test platform. This makes the CLR implementation of Clojure the least supported implementation, causing development to be relatively awkard and less maintainable.
- The CLR has good support for graphics through DirectX
- Touch event handling is natively supported in .NET, with no extra dependencies necessary.
- The team has little experience of the de facto libraries on the .NET platform.
- The platform has mature support for a wide variety of applications including games.

6.6 The Java Virtal Machine

- The JVM is the main target of Clojure. The JVM implementation of Clojure has the most mature tools and the largest community.
- The JVM is not a platform often used for games, but OpenGL libraries are readily available, and support should be strong enough for a prototype and probably more.
- Because the JVM is Clojure's main target, there is a wide variety of libraries available. This includes good rendering support for 2D and audio support.
- The JVM is somewhat lacking in its support for touch input. However, during the orientation phase the team was able to quickly develop a usable library for this purpose.
- Every member of the team has worked with Clojure on the JVM to some extend and is comfortable working with it.
- The target environment is suitable for the performance needed for rendering and event handling.

6.7 JavaScript (ClojureScript)

Clojure has an implementation in JavaScript, called ClojureScript. * ClojureScript has a working build system but only minimal support for unit testing. * JavaScript is inherently single-threaded. On a multi-core system this leaves much of the CPU unused. * OpenGL is supported both in the browser (through WebGL) and in NodeJS (a non-browser JavaScript environment) through C bindings. * Prior knowledge in the team is limited. * ClojureScript can be used in the browser with limited support and in NodeJS with unidiomatic support.

7 Game Engines

7.1 JMonkeyEngine

The JMonkeyEngine(JME) is a popular game engine written in java. It offers support for features including but not limited to 3d-graphics, physics, collision detection, input handling and audio.

7.2 First impressions

The JMonkey Engine is designed as a set of individual libraries, which makes it easy to only use the features you need. The website offers high-quality documentation for all the libraries and has tutorials on some basic use cases showing how one can combine some of the libraries, mainly focussed on the graphics library. Some small demo applications showing the capabilities of the graphics library have been made available, with all of these applications being subclasses of SimpleApplication. Overriding the SimpleInitApp and SimpleUpdate is the way games can be rendered.

7.3 Advantages

- Libraries for 3d graphics and physics
- Clear documentation
- Tutorials
- Active community

7.4 Disadvantages

- Input handling framework is intrusive
- No need for 3d support and graphics
- Quite heavily coupled together despite being seperate libraries
- Big time investment

7.5 Multitouch for Java

MultiTouch 4 Java (MT4J) is a Java framework for developing visually rich, multi-touch-enabled applications and games.

7.6 First impressions

MT4J development has halted somewhere in 2011, with the most recent source package left in an uncompilable state.

7.7 Advantages

- Input methods: MT4J already supports multiple input methods, from TUIO to the Windows Touch API, which are both possible input methods for the target hardware platform.
- Loadable resources: There are several resource loaders available in MT4J. The list of supported formats include SVG, .3DS and .obj files.
- Gesture recognition: Standard gestures are included in MT4J, with the added possibility of adding custom gestures to the embedded gesture recognition framework.
- Flexible rendering system: MT4J support rendering schemes built on top of Processing, OpenGL and even Java2D. Depending on our needs, we could go with the least complicated abstraction level of rendering while still retaining the required level of control over the rendering process.

7.8 Disadvantages

- Doesn't compile: *If* we make the decision to use MT4J, a significant portion of available development time has to be invested in making sure MT4J compiles, is tested and is suited for our particular purposes. This development time could potentially be better spent designing and implementing an interesting and fun game.
- Nonexistent support: Because development has halted, there is currently no active MT4J community to speak of. If unexpected problems arise in the course of this project, we are left to our own devices to solve or mitigate these problems.

7.9 Penumbra

Penumbra is an idiomatic OpenGL wrapper for Clojure, suitable for use in interactive development of graphic-intensive applications.

7.10 First impressions

Penumbra is currently not actively developed, although bug fixes will be applied when appropriate. At first glance, Penumbra seems to only work with older, deprecated versions of Clojure.

7.11 Advantages

- OpenGL for Clojure: Penumbra offers the possibility to control the complete power of OpenGL in a way that plays nicely with a REPL-based development style.

7.12 Disadvantages

- Current Clojure version: Penumbra doesn't seem to work out of the box with a recent version of Clojure. Porting Penumbra would be a separate project on itself, seeing as there are currently no unit tests or specifications whatsoever.
- Non-deterministic run-time behavior: Programs utilizing Penumbra only seem to work occasionally. Performing an in-depth investigation to find the root cause for this problem is most certainly out of scope for this project.

7.13 Quil

Quil is an idiomatic wrapper for Processing, a graphics rendering library for Java. Quil is based on Processing 1.5.1.

7.14 First impressions

Quil works with sketches, which are simplified representations of visual applications. Each sketch has a draw function, which is called repeatedly to render to a window. A major part of Quil is devoted on extending this functionality in an easy-to-use manner.

7.15 Advantages

- Online documentation: Quil has a documentation system integrated in the clojure repl. Calling (show-cats) gives a category overview of all functions in clojure, and (show-fns *category-id*) shows all functions in a category. Finally, (doc *function-name*) shows the documentaiton of particular functions as usual.
- Redefinable draw function: The draw function can be redefined without stopping the sketch. Though the draw callback can't be changed while a sketch is running, the callback can be a simple proxy function to the real draw function, which can be changed. This allows for a edit-compile-run cycle of seconds instead of minutes, as any change of code immediately propagates to the rendering process.

7.16 Disadvantages

- No collision detection: Quil only draws, but doesn't retain knowledge about the objects it has drawn. Therefore it can't do collision detection. Any collision detection will have to be implemented outside of quil.
- Limited 3D: While Quil offers 3d primitives, there is no support for programmable pipeline constructs (shaders). There seem to be processing libraries that support this though, and the newer processing (2.0b8) also supports this, but it'll take some effort to integrate this into Quil.

8 TUIO

TUIO [Kaltenbrunner et al., 2005] is comprised of a protocol and API for abstract multi-touch interfaces. It includes a way to separate the mapping of input to the TUIO protocol, after which all TUIO-enabled clients can interpret the translated input in a correct manner.

8.1 First impressions

TUIO seems a reasonably thought-through framework, with existing implementations of both TUIO-servers and clients for most of today's programming environments being available on the TUIO website.

8.2 Advantages

- Decoupled Input Handling: By implementing TUIO, an application is no longer dependent on a particular user input interface. As long as there is a TUIO server available that maps input to the TUIO protocol, a single unmodified version of an application can be deployed on multiple platforms.
- Easy to Implement: TUIO offers a modularized protocol, which implies that creating partial TUIO support for your platform of choice is rather easy. TUIO currently supports 2d, 2.5d and 3d cursors, objects and blobs.

8.3 Disadvantages

- Relatively Unreliable: TUIO uses UDP for communications between the client and server software. If client and server programs are not running on the same device, or even the same network, package loss and latency can become a hindrance to normal operation of the application.
- Simple: Because TUIO is built on top of a generalized, abstract representation of multi-touch interfaces, some platform specific features or efficiency is lost when TUIO is used. For example, the native Windows gestures aren't supported by TUIO and have to be reimplemented on top of the interaction primitives provided by TUIO.

9 iGesture

iGesture is a gesture recognition framework for Java. It provides a library to recognize gestures, and a tool for defining gestures.

9.1 Advantages

- Easy tool for the definition of custom gesture.
- Works with a wide range of input devices.

9.2 Disadvantages

- Complexity: A gesture library is an extra layer of complexity, which might not be justified given the scope of our project.
- Delay: The library recognizes completed gestures only. Actions that need to occur while a gesture is happening can't be handled by the iGesture framework.

10 Tool Selection

10.1 Language

Our language of choice is Clojure. The reason we chose for Clojure rather than a more established language is that this project requires a rapid development cycle. REPL-based development is very suitable for this task, and since Clojure is hosted on a platform it has access to all the libraries of its host.

10.2 Platform

Given the choice for Clojure, the most suitable platform for our application is the JVM. Javascript is too limited, and Clojure on .NET misses a lot of the tools required for a succesful software development cycle.

10.3 Game Engine

Neither one of the game engines considered were suitable for our needs. MT4J is an abandoned project which was left in an uncompilable state. Getting it working would be a significant time investment. JMonkeyEngine is a complete solution but lacks multi-touch support, and is heavily focused on 3D development, while our application will be 2D. We have decided to build our own engine.

10.4 Graphics Library

For graphics, we've decided to use Quil, an idiomatic wrapper of Processing for Clojure. Quil offers rapid REPL-based development, and unlike the alternative we considered, Penumbra, it is still actively maintained and works with the latest version of Clojure.

10.5 Touch Interface

For Touch input we'll use TUIO, a generic protocol for various kinds of multi-touch devices. This allows us to test our application on more devices than just the target platform at the Gemeente Delft. The alternative, using Windows-specific libraries, did not offer us that possibility.

10.6 Gesture recognition

We briefly considered using a gesture recognition framework, and investigated using iGesture for this purpose. However, this and similar libraries have the disadvantage of only recognizing completed gestures. This, and the complexity added by a gesture recognition library far outweight the advantages of using this library. Since the gestures required for our project are not very complex, it'll be more easy to write our own gesture recognition rather than using a library.

11 Game Architecture

Game development typically happens in statically typed and Object-Oriented programming languages. In stark contrast, Clojure is a dynamically typed, multiple paradigm programming language, with different strengths and weaknesses depending on the application domain.

11.1 Object Oriented (OO) game design

A typical game loop in a typical OO-language consists of the following four steps repeated in an (endless) loop: 1. Handle user input 1. Update the game state 1. Render the game state 1. Wait until the designated time span has passed

Keeping everything sequential has some advantages for a OO-programmer, of which the major one would be that no coordination of access to mutable state is needed. A disadvantage of this approach is the fact that modern hardware has support for lots concurrently executing threads, but this capability is left unused when using this simple design.

11.2 Clojure's multiple paradigm game design

Clojure borrows heavily from functional programming languages, but still manages to combine several FP concepts with tried and tested OO techniques. As such, Clojure provides the concurrency primitives which are needed to break free from the restrictions of typical OO-languages.

Instead of viewing the game as a repeating sequence of operations, a Clojure game can be seen as a set of asynchronous processes operating on a (set of) concurrency primitives representing the game state. By leveraging closures, callbacks and higher-order functions, one can totally decouple rendering, physics, animation, input handling and even AI subsystems from each other[Tellman, 2010]. Combining these techniques with existing OO libraries leads to a maintainable, testable and above all, simple system which utilizes the capabilities of modern hardware.

11.3 Scheduling

Built into the Java platform is a task scheduler. Using this scheduler, tasks can be scheduled to either run once at some defined future time or run repeatedly until stopped. Using this scheduler all game actions that are time-based (i.e. rendering) rather than event-based (i.e handling user input). can be scheduled, ensuring that the game engine does not need to have a dedicated timed loop for them.

11.4 Thread safety

A primary reason why many games choose for a single-threaded model is thread safety. Thread safety would normally require careful locking of shared data, but due to Clojure's data immutability and shared data constructs, much of the work for reasoning about thread operation has already been done. Thread safety is no reason to avoid a multi-threaded program.

References

[Finke et al., 2008] Finke, M., Tang, A., Leung, R., and Blackstock, M. (2008). Lessons learned: Game design for large public displays. *DIMEA*.

[Hickey, 2013] Hickey, R. (2013). Clojure.

[Kaltenbrunner et al., 2005] Kaltenbrunner, M., Bovermann, T., Bencina, R., and Costanza, E. (2005). Tuio: A protocol for table-top tangible user interfaces. In *Proc. of the The 6th Intl Workshop on Gesture in Human-Computer Interaction and Simulation.*

[Tellman, 2010] Tellman, Z. (2010). Creating a simple game in clojure. `http://ideolalia.com/creating-a-simple-game-in-clojure/index.html`.

A Game Design

Traffic is a game about resource transportation. The game displays a map containing supply and drop-off points, and the goal for the players is to build an efficient transport network, moving resources from supply to drop-off points

A.1 Game Elements

A.1.1 Levels

The game is level-based. Each level is built around a map, consisting of pre-defined Supplies and Depots, a transport target (how many resources should end up in each depot) and some pre-defined zones where special rules are in effect for the grid-cells.

A.1.2 Cells

The game map is divided into equally-sized square cells. The cells contain the game elements: paths, supplies, depots,transhipment points and vehicle spawners.

Each cell has a zone type which specifies what rules are in effect for that cell. Such rules are maximum driving speed (such as in residential areas) maximum vehicle throughput (environmental concerns) and allowed path types.

Paths Paths are used to transport resources. There are 3 types: roads (used by trucks), canals (used by boats) and rails (used by trains). Each cell has 4 connection points, allowing paths to connect with adjacent cells. Paths of different types can cross eachother, and paths can split and join. The player configures the resources that can be transported over a path.

Supplies Supplies are the start point for resources. An unlimited amount of resources can be extracted from any supply. These resources will all be of the same type. No Supply supplies more than one type of resource. A Supply occupies a cell, and can supply to vehicles on any adjacent cell.

Depots Depots are the end point for resources. When resources arrive at a Depot, they're consumed. Depots occupy a cell, and consume resources from vehicles on adjacent cells. Depots are pre-set to receive only a certain kind of resource.

Transhipment Points Transhipment points are both an end point for resources and a starting point. They allow transhipment of resources from one type of path to another. When a loaded vehicle drives by on an adjacent cell, it will drop off its resource at the transhipment point. If an unloaded vehicle drives by, it picks up the resource. Transhipment points can only contain one kind of resource, so if a resource has already been dropped off, only resources of the same type will be dropped off afterwards, until the transhipment point is empty again.

Vehicle Spawners Vehicle spawners spawn vehicles. For roads, these are trucks, for canals boats and for rails trains. Each type of vehicle differs in maximum speed, maximum cargo and zone restrictions.

The vehicle spawner can spawn an unlimited amount of vehicles, but the players will be penalized if they don't return vehicles to a vehicle spawner. Each vehicle spawner has an entry and an exit point which must be connected correctly. Vehicles are supposed to either drive around in a loop back to the same spawner, or drive towards another spawner.

A.2 Game Mechanics

The players play by building paths and specifying what type of resource can travel over each path. The players have no direct control over the vehicles.

A.2.1 Pause and Resume

The game starts paused, allowing the players to specify whatever paths and routes they want. While paused, no vehicles will move. The players can resume the game in order to make the vehicles move.

A.2.2 Building Paths

Paths can be built by extending existing paths or splitting off from them. Each level will begin with some basics paths that can be extended.

A.2.3 Specifying Routes

Routes can be specified by extending from a supply or a depot. Since both supplies and depots only deliver/consume one type of resource, the kind of route this creates is always unambiguous. Routes can join together on a path and split away later when the path splits.

A.2.4 Vehicle Behavior

Vehicles follow simple rules even if it means driving off a road and crashing. The rules are as follows:

- On spawn, leave in the vehicle spawn exit direction.
- When passing by a supply or transhipment point while fully loaded, pick up cargo (if there is any.)
- When passing by a transhipment point while carrying cargo, drop the cargo if the transhipment point allows this cargo type (this is the case when the transhipment point is either empty or containing cargo of the same type).
- When passing by a depot while carrying cargo, drop cargo if it matches the depot's requirement.
- When at a split, choose the path that allows the current cargo. If multiple such paths are available, choose the left-most one.
- When a vehicle enters a vehicle spawn, it disappears.
- When a vehicle enters the end of a road, it crashes and disappears with a penalty.

References

[1] Ben Moseley and Peter Marks. Out of the tar pit. In *SOFTWARE PRACTICE ADVANCEMENT (SPA)*, 2006.