



CZECH  
TECHNICAL  
UNIVERSITY  
IN PRAGUE

**Faculty of Electrical Engineering**  
**Department of Computer Science**

**Bachelor's thesis**

# **Board Game Engine**

**Vojtěch Kaiser**

**May 2015**

**Thesis supervisor: Sporka Adam Ing., Ph.D.**

České vysoké učení technické v Praze  
Fakulta elektrotechnická

katedra počítačů

## ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: **Vojtěch Kaiser**

Studijní program: Otevřená informatika (bakalářský)  
Obor: Softwarové systémy

Název tématu: **Herní engine pro implementaci deskových her**

Pokyny pro vypracování:


Identifikujte typické požadavky autorů počítačových adaptací deskových her. Proveďte rešerši stávajících systémů pro implementaci těchto her. Navrhněte vlastní řešení herního engine. Engine podrobte testům použitelnosti programátory. Pomocí systému realizujte dvě deskové hry a otestujte s reálnými hráči.

Seznam odborné literatury:

Richard A. Bartle: Designing Virtual Worlds  
David H. Eberly: 3D Game Engine Design

Vedoucí: Ing. Adam Sporka, Ph.D.

Platnost zadání: do konce letního semestru 2014/2015

  
doc. Ing. Filip Železný, Ph.D.  
vedoucí katedry




  
prof. Ing. Pavel Ripka, CSc.  
děkan

V Praze dne 25. 2. 2014

## **Declaration**

I hereby declare that I have completed this thesis independently and that I have used only the sources (literature, software, etc.) listed in the enclosed bibliography.

Prague 22 May 2015

A handwritten signature in black ink, appearing to be 'J. K. M.', with a long horizontal flourish extending to the right.



## Thanks

I would like to express my gratitude to my supervisor Adam Sporka for his useful advices, comments and remarks. I am also grateful to all participants that sacrificed their precious time for testing of my creation. Furthermore, I would like to thank my parents for all the support, emotional and financial, they provided me with. At last, I would like to express my sincere gratitude to my dear Dominique for keeping me sane, encouraging and showing all the support possible, and in the end helping me with correction of my atrocious grammar in this thesis. (note: some places did not go through her hands so you can experience true horror).

## Abstrakt

Tato práce se zabývá tvorbou herního *engine* specificky určeného pro implementaci deskových her. Výsledek práce je funkční *framework*, ve kterém lze implementovat hry reprezentovatelné jednoduchým stavovým automatem. Pro demonstraci možností vytvořeného *engine* jsou v něm vytvořeny dvě deskové hry pro více hráčů. V závěru jsou pro ověření použitelnosti systému otestovány procesy spojené s tvorbou takových her s programátory podobných schopností jako má předpokládaná cílová skupina.

## Abstract

The aim of this thesis is to analyze the creation of game *engine* specifically designated for implementation of board games. The result of this work is a functional *framework* in which is possible to implement games that can be represented by simple state machine. The *engine* is used for two board games as a demonstration of its capabilities. Processes related to making of games in the created engine are tested with programmers of similar expertise to target group for verification of its usability.

# Contents

|           |                                     |           |
|-----------|-------------------------------------|-----------|
| <b>1</b>  | <b>Introduction</b>                 | <b>1</b>  |
| 1         | Motivation                          | 1         |
| 2         | Used technologies                   | 1         |
| 3         | Design outline                      | 2         |
| 4         | Testing                             | 2         |
| 4.1       | Creators . . . . .                  | 3         |
| 4.2       | Players . . . . .                   | 3         |
| 5         | Goals                               | 3         |
| <b>2</b>  | <b>Background</b>                   | <b>4</b>  |
| 6         | Classification                      | 4         |
| 7         | Physical environment comparison     | 4         |
| 7.1       | Board game mechanics . . . . .      | 6         |
| 7.2       | General game engine types . . . . . | 8         |
| 7.3       | Existing solutions . . . . .        | 9         |
| <b>8</b>  | <b>General requirements</b>         | <b>10</b> |
| 8.1       | GUI . . . . .                       | 10        |
| 8.2       | Game components . . . . .           | 11        |
| 8.3       | Resources . . . . .                 | 11        |
| 8.4       | Network . . . . .                   | 12        |
| 8.5       | Game rules . . . . .                | 12        |
| 9         | Discussion                          | 12        |
| <b>3</b>  | <b>Design</b>                       | <b>13</b> |
| <b>10</b> | <b>General architecture</b>         | <b>13</b> |
| <b>11</b> | <b>Event driven applications</b>    | <b>13</b> |
| 11.1      | Advantages . . . . .                | 13        |
| 11.2      | Disadvantages . . . . .             | 14        |
| 11.3      | Application . . . . .               | 15        |
| 11.4      | Modularity . . . . .                | 16        |

|                                              |           |
|----------------------------------------------|-----------|
| <b>12 Architecture layout</b>                | <b>16</b> |
| 12.1 Scene design . . . . .                  | 18        |
| 12.2 Logic design . . . . .                  | 20        |
| 12.3 Data processing . . . . .               | 20        |
| 12.4 Network synchronization . . . . .       | 21        |
| 12.5 Security . . . . .                      | 21        |
| <b>13 Discussion</b>                         | <b>22</b> |
| <br>                                         |           |
| <b>4 Implementation</b>                      | <b>23</b> |
| <br>                                         |           |
| <b>14 Scene</b>                              | <b>23</b> |
| 14.1 Graph . . . . .                         | 23        |
| 14.2 Selection . . . . .                     | 24        |
| 14.3 Styles . . . . .                        | 25        |
| 14.4 Style properties . . . . .              | 26        |
| 14.5 Scene states . . . . .                  | 26        |
| 14.6 Locks . . . . .                         | 27        |
| <b>15 Events</b>                             | <b>28</b> |
| <b>16 Logic</b>                              | <b>28</b> |
| 16.1 Rule Handler . . . . .                  | 28        |
| 16.2 State space . . . . .                   | 29        |
| 16.3 Rulesets . . . . .                      | 29        |
| 16.4 Action vs Rule . . . . .                | 29        |
| 16.5 Evaluation element definition . . . . . | 29        |
| <b>17 Registry</b>                           | <b>31</b> |
| 17.1 Registry keychain . . . . .             | 31        |
| 17.2 Value storage . . . . .                 | 31        |
| 17.3 Map . . . . .                           | 31        |
| 17.4 List . . . . .                          | 31        |
| 17.5 Example . . . . .                       | 32        |
| 17.6 Merging . . . . .                       | 32        |
| <b>18 Rendering</b>                          | <b>33</b> |
| 18.1 2D graphics acceleration . . . . .      | 33        |
| 18.2 Updates . . . . .                       | 34        |
| 18.3 Application window . . . . .            | 34        |
| <b>19 Loading</b>                            | <b>34</b> |
| 19.1 XML . . . . .                           | 35        |
| 19.2 Styles . . . . .                        | 36        |

|                                        |           |
|----------------------------------------|-----------|
| <b>20 Network</b>                      | <b>36</b> |
| 20.1 Server . . . . .                  | 36        |
| 20.2 Client . . . . .                  | 37        |
| 20.3 Issues . . . . .                  | 37        |
| <b>21 Tic-Tac-Toe</b>                  | <b>38</b> |
| 21.1 Rules . . . . .                   | 38        |
| 21.2 State machine . . . . .           | 38        |
| 21.3 Scene . . . . .                   | 39        |
| 21.4 Registry . . . . .                | 40        |
| 21.5 Walkthrough . . . . .             | 41        |
| <b>22 Cards against humanity</b>       | <b>43</b> |
| 22.1 Rules . . . . .                   | 43        |
| 22.2 State machine . . . . .           | 43        |
| 22.3 Scene . . . . .                   | 45        |
| 22.4 Registry . . . . .                | 45        |
| 22.5 Walkthrough . . . . .             | 45        |
| <b>23 Discussion</b>                   | <b>54</b> |
| <b>5 Usability tests</b>               | <b>55</b> |
| <b>24 Testing in general</b>           | <b>55</b> |
| <b>25 Testing with programmers</b>     | <b>55</b> |
| 25.1 Method . . . . .                  | 55        |
| 25.2 Screening . . . . .               | 56        |
| 25.2.1 Participant 1 . . . . .         | 57        |
| 25.2.2 Participant 2 . . . . .         | 57        |
| 25.2.3 Participant 3 . . . . .         | 57        |
| 25.3 Tasks . . . . .                   | 58        |
| 25.4 Post test questionnaire . . . . . | 61        |
| 25.5 Found problems . . . . .          | 63        |
| 25.6 Problems analysis . . . . .       | 64        |
| 25.7 Conclusion . . . . .              | 64        |
| <b>26 Testing with players</b>         | <b>65</b> |
| 26.1 Method . . . . .                  | 65        |
| 26.2 Results . . . . .                 | 65        |
| 26.3 Found problems . . . . .          | 66        |
| 26.4 Conclusion . . . . .              | 67        |
| <b>6 Conclusion</b>                    | <b>68</b> |



|                                               |           |
|-----------------------------------------------|-----------|
| <b>27 Further development</b>                 | <b>69</b> |
| <b>7</b>                                      | <b>70</b> |
| <b>Appendices</b>                             | <b>70</b> |
| <b>A Getting started</b>                      | <b>70</b> |
| A.1 Installation . . . . .                    | 70        |
| <b>B Scene graph</b>                          | <b>71</b> |
| B.1 File specification . . . . .              | 71        |
| B.2 Available nodes . . . . .                 | 72        |
| B.2.1 SceneNode . . . . .                     | 72        |
| B.2.2 PaintableNode . . . . .                 | 72        |
| B.2.3 StyleableNode . . . . .                 | 72        |
| B.2.4 GroupNode . . . . .                     | 73        |
| B.3 Available components . . . . .            | 73        |
| B.3.1 Container . . . . .                     | 73        |
| B.3.2 Panel . . . . .                         | 73        |
| B.3.3 Text . . . . .                          | 74        |
| B.3.4 Layout . . . . .                        | 74        |
| B.4 Adding new components and nodes . . . . . | 74        |
| <b>C Events</b>                               | <b>76</b> |
| C.1 Available events . . . . .                | 76        |
| C.1.1 Event . . . . .                         | 76        |
| C.1.2 Window event . . . . .                  | 76        |
| C.1.3 Mouse event . . . . .                   | 77        |
| C.1.4 Message event . . . . .                 | 77        |
| C.1.5 Keyboard event . . . . .                | 77        |
| C.1.6 Interaction event . . . . .             | 78        |
| <b>D Scene styles</b>                         | <b>78</b> |
| D.1 File specification . . . . .              | 78        |
| D.1.1 Comments . . . . .                      | 78        |
| D.1.2 Selectors . . . . .                     | 79        |
| D.1.3 Styles . . . . .                        | 80        |
| D.1.4 Includes . . . . .                      | 80        |
| D.2 Available properties . . . . .            | 80        |
| D.2.1 Layer . . . . .                         | 81        |
| D.2.2 Visible . . . . .                       | 81        |
| D.2.3 Style ID . . . . .                      | 82        |
| D.2.4 Style class . . . . .                   | 82        |
| D.2.5 Interaction states . . . . .            | 82        |

|          |                                                |           |
|----------|------------------------------------------------|-----------|
| D.2.6    | Dimensions . . . . .                           | 83        |
| D.2.7    | Margin . . . . .                               | 83        |
| D.2.8    | Padding . . . . .                              | 84        |
| D.2.9    | Pivot . . . . .                                | 84        |
| D.2.10   | Position . . . . .                             | 84        |
| D.2.11   | Min-max dimensions . . . . .                   | 85        |
| D.2.12   | Aspect ratio . . . . .                         | 85        |
| D.2.13   | Background color . . . . .                     | 86        |
| D.2.14   | Background image . . . . .                     | 86        |
| D.2.15   | Background repeat . . . . .                    | 86        |
| D.2.16   | Background image alpha . . . . .               | 87        |
| D.2.17   | Border size . . . . .                          | 87        |
| D.2.18   | Border color . . . . .                         | 88        |
| D.2.19   | Border radius . . . . .                        | 88        |
| D.2.20   | Border position . . . . .                      | 88        |
| D.2.21   | Border mode . . . . .                          | 89        |
| D.2.22   | Border image . . . . .                         | 89        |
| D.2.23   | Cursor . . . . .                               | 89        |
| D.2.24   | Solid . . . . .                                | 90        |
| D.2.25   | Mask . . . . .                                 | 90        |
| D.2.26   | Font color . . . . .                           | 90        |
| D.2.27   | Font name . . . . .                            | 91        |
| D.2.28   | Font size . . . . .                            | 91        |
| D.2.29   | Font style . . . . .                           | 91        |
| D.2.30   | Text align . . . . .                           | 92        |
| D.2.31   | Line wrap . . . . .                            | 92        |
| D.2.32   | Line stretch . . . . .                         | 92        |
| D.2.33   | Text content . . . . .                         | 93        |
| D.2.34   | Layout type . . . . .                          | 93        |
| D.2.35   | Layout spacing . . . . .                       | 93        |
| D.3      | Adding new properties . . . . .                | 94        |
| <b>E</b> | <b>Logic</b>                                   | <b>95</b> |
| E.1      | State space file specification . . . . .       | 95        |
| E.2      | Available rules . . . . .                      | 96        |
| E.2.1    | Check message . . . . .                        | 96        |
| E.2.2    | Compare numbers . . . . .                      | 96        |
| E.2.3    | Keyboard button released . . . . .             | 97        |
| E.2.4    | Keyboard character typed . . . . .             | 97        |
| E.2.5    | Load user IP address . . . . .                 | 97        |
| E.2.6    | Load event contents . . . . .                  | 97        |
| E.2.7    | Load string length . . . . .                   | 99        |
| E.2.8    | Mouse button released over component . . . . . | 99        |
| E.2.9    | String equals . . . . .                        | 99        |

|          |                                       |            |
|----------|---------------------------------------|------------|
| E.2.10   | String regex match . . . . .          | 100        |
| E.3      | Available actions . . . . .           | 100        |
| E.3.1    | Apply stylesheet . . . . .            | 100        |
| E.3.2    | Change state in state space . . . . . | 100        |
| E.3.3    | Change state of node . . . . .        | 101        |
| E.3.4    | Change text of component . . . . .    | 101        |
| E.3.5    | Clear node . . . . .                  | 102        |
| E.3.6    | Dictionary translation . . . . .      | 102        |
| E.3.7    | Dump registry . . . . .               | 102        |
| E.3.8    | Dump states . . . . .                 | 102        |
| E.3.9    | Insert scene . . . . .                | 103        |
| E.3.10   | Load registry . . . . .               | 103        |
| E.3.11   | Log message . . . . .                 | 103        |
| E.3.12   | Modify counter . . . . .              | 103        |
| E.3.13   | Network controller command . . . . .  | 104        |
| E.3.14   | Send message . . . . .                | 104        |
| E.3.15   | Shutdown application . . . . .        | 105        |
| E.3.16   | Store data . . . . .                  | 105        |
| E.4      | Grouping . . . . .                    | 105        |
| E.5      | Adding new elements . . . . .         | 106        |
| <b>F</b> | <b>Optimization</b>                   | <b>107</b> |
| F.1      | Scene . . . . .                       | 107        |
| F.2      | Logic . . . . .                       | 108        |
| F.3      | Network . . . . .                     | 108        |
| <b>G</b> | <b>Contents of DVD</b>                | <b>110</b> |

## List of Figures

|    |                                                             |    |
|----|-------------------------------------------------------------|----|
| 1  | General architecture layout . . . . .                       | 17 |
| 2  | GUI input processing . . . . .                              | 18 |
| 3  | General scene layout . . . . .                              | 19 |
| 4  | Manipulating parent without influence on children . . . . . | 19 |
| 5  | General logic layout . . . . .                              | 20 |
| 6  | General logic layout . . . . .                              | 21 |
| 7  | Inheritance schema of implemented nodes and components      | 24 |
| 8  | State machine for <i>TTT</i> . . . . .                      | 39 |
| 9  | <i>TTT</i> — New game . . . . .                             | 41 |
| 10 | <i>TTT</i> — First move . . . . .                           | 42 |
| 11 | <i>TTT</i> — End of the game . . . . .                      | 42 |
| 12 | State machine for <i>CAH</i> . . . . .                      | 44 |
| 13 | <i>CAH</i> — Connection screen . . . . .                    | 46 |
| 14 | <i>CAH</i> — Connection status . . . . .                    | 47 |
| 15 | <i>CAH</i> — Player picking name . . . . .                  | 47 |
| 16 | <i>CAH</i> — Host waiting for players to connect . . . . .  | 48 |
| 17 | <i>CAH</i> — Client being ready for game to start . . . . . | 48 |
| 18 | <i>CAH</i> — Host being ready for game to start . . . . .   | 49 |
| 19 | <i>CAH</i> — Player picking an answer card . . . . .        | 50 |
| 20 | <i>CAH</i> — Player picked an answer card . . . . .         | 50 |
| 21 | <i>CAH</i> — Czar waiting . . . . .                         | 51 |
| 22 | <i>CAH</i> — Czar picked winner . . . . .                   | 51 |
| 23 | <i>CAH</i> — Player observed what czar picked . . . . .     | 52 |
| 24 | <i>CAH</i> — Player has gambling enabled . . . . .          | 52 |
| 25 | <i>CAH</i> — Player gambled extra card . . . . .            | 53 |
| 26 | <i>CAH</i> — Increased number of cards for czar . . . . .   | 53 |

## Part 1

# Introduction

This thesis focuses on creation of 2D game engine purposed for prototyping and implementation of board games of varying complexity. First is reader introduced to problematic of board games in digital environment, then follows general engine design outline which is in implementation part demonstrated and at last tested with programmers and players.

## 1 Motivation

Board games have always had a place in casual gaming on computers but so far only as time filler on office computers as a form of distraction (solitaire as example for all). They experienced a rise in recent years, together with many attempts, successful and unsuccessful, to transform classic or brand new board games into PC games.

Common practice when implementing a PC game is to use a multi-purpose engine designed by professionals. However, these engines are usually overly complicated for as simple a thing as board game, and platform dependent, as they strive to squeeze every bit of performance out of computer they are running on. Certain groups of *indie*<sup>1</sup> board games creators might find complexity of those engines and their creation process quite intimidating, especially for those without programming background.

My motivation here is to provide an easy to use and understand engine that will require little to no programming knowledge on its user, which will remove all platform dependent issues and be fast enough to handle basic 2D games.

## 2 Used technologies

Programming language I decided to use is java because it is easy to understand. Also, it is a high level OOP language that is restrictive just enough to prevent the most tedious errors one might encounter in other low level languages. Java is renown for its not-so-impressive speed but I came to conclusion that with advantage of *Just In Time*<sup>2</sup> compiler (*JIT*) and accelerated image processing it will be fast enough for the task lying ahead.

---

<sup>1</sup>Term indie is used with relation to independent content creators or teams or companies that are not owned by a publisher.

<sup>2</sup>Just-in-time compiler for java is compiler that turns java byte code into machine native code. This compilation is done every time program is run and data specific optimizations may be done over compiled code depending on JIT being used.

## 4 TESTING

For loading general data, the XML definition seemed as the best and fastest way, since parsing of XML in java is well supported and XML structures are widely known. For parsing itself is in java the *Java Architecture for XML Binding (JAXB)* that allows generating of classes corresponding to your XML structure. All XML trees are loaded and constructed using these classes.

For the styling of scene I decided to define my own variation of *Cascading Style Sheets (CSS)*, which has several identical properties, a few with similar definition but different effect and a certain number of fully new properties. Precise differences will be elaborated in the appendix part. I decided to create my own implementation because of lack of constrictions laid on present features. This way, I can, or any user of the engine for that matter, define new properties and effectively change behaviour of existing ones in order to be make them generally more useful.

### ■ 3 Design outline

Given the turn-based nature of most board games and the fact they can be represented by a simple state machine, we can turn to architecture which is not as focused on performance. We can sacrifice a certain portion of performance in exchange for structures that are easy to maintain, change and read. That is the reason for this engine revolving around user-defined state machine controlled by an event driven core.

Scene displayed to a user is defined as *HTML* page, and behaves likewise. Moreover, it has the possibility of defining new visual states (enabled, disabled, ...) and managing the way events propagate through. Any propagation in this scene is done with synchronous calls, but once the information (event) leaves the scene, it is left to asynchronous execution by event handler.

Logic is contained solely in state machine, where each state contains rulesets consisting of actions and rules, which can fail and terminate the execution. All data used by these actions and rules are acquired from registry collection that can be pre-filled at startup, from variable space belonging to particular ruleset or as constants directly in rule definitions.

Network management is designed to be as simple as possible, or at least from user point of view. Connection is initiated by filling ports and hostname, and when that is successful, events marked for network are simply sent either to server or to clients, depending on type of currently running platform.

### ■ 4 Testing

One of the main features of this engine should be its usability. For verification of usability, we need to perform proper testing with real life users.

## ■ 4.1 Creators

Since this game engine is intended for users of varying programming abilities, it needs to be properly tested for clarity of documentation and all related processes. The proof would be for such a user to create simple a game within limited amount of time. This testing should provide enough feedback to modify the engine to improve its usability, or at least soften the learning curve.

## ■ 4.2 Players

Although the testing with players itself is not the focus of this work, the actual usability of possible resulting game needs to be tested as well. If the created game does not meet criteria set by players for pleasant game experience, and it is not possible to solve found problems using current state of the engine, more drastic changes have to be made.

## ■ 5 Goals

This engine should be capable of displaying defined interactive scene through which should player be able to affect inner state machine and registry, possibly with synchronization over network with other players. This aim should be achievable by writing and applying XML and stylesheet definitions at runtime with possibility of dynamically loading classes implementing predefined interfaces.

## Part 2

# Background

As the goal of this game engine is to implement board games, an outline of general classification of board games will be presented, together with an example of other engines focused on this problem. That will provide us with the insight of what to include and what to avoid during implementation of the engine.

## 6 Classification

In this section I shall list all the relevant aspects of the problem and highlight the ones that this work will be focused on. Although not all possible classified games will be in resulting state ready to be implemented, they should still be possible to add later on — engine should not put significant restrictions on future feature updates.

As for board game genre, even though there might be a difference between functional requirements of fantasy and scifi board game, I will not take them into account as a relevant classifier for this design.

## 7 Physical environment comparison

Bringing board games to digital environment is carrying some advantages and drawbacks. When are board games played in physical environment, there is strong social interaction aspect to it, and that is possibly one of the main reasons board games are being played at all. If we will try to digitalize them, we need to maintain as much of that interaction as possible. Without it, we are in the end just implementing another computer game and that was and will be done many times over and probably better by large teams that are specializing in the area.

Basic instrument to keeping social experience in the game is message system in the game. That is of course not enough and hides all spontaneous reactions of players that differentiate face-to-face play sessions. As enhancement, we can implement conference call system on top of regular chat, that will make players feel more connected to each other. Of course, we could decide to go even further and transfer video, but there is beginning to show the limitation of available equipment that every player has. It will not be possible to comfortably put faces of five other player on one screen while having enough space for game components, not to mention poor quality of cheaper web cameras that will average player probably possess.



Great advantage on the other hand is that communication between players may be private, which gives to specific genre of games whole new dimension. For instance in board game *Game of Thrones* <sup>3</sup> are players encouraged to secretly cooperate against other players. This has in physical environment very limited effect because everyone can see which players are talking to each other, and even if their opponents do not hear what is the topic, it is a sign of some mischief.

Another advantage board games in digital environment may possess is learning curve softening. Big portion of board games have complicated rules to stay interesting for experienced players. This can be quite intimidating, especially when mentioned rules have more than *20 pages* of text. Players are then discouraged from sessions because between start of the session and actual gameplay may lay an hour of explanation of rules, which at least one of the players has to study beforehand. This is not only common for new players, but for those that already played the game some time ago. That not only means they have to go through that painstakingly boring process again, but now they know about it, which discourages them even more. Board games in digital environment can completely remove this issue by simply telling player what to do, highlighting possible options and not letting player to do anything else. This is of course subject to good UI design and large amount of testing.

Certain board games are designed to be not only complicated, but as well lengthy. These two go usually hand in hand, since player having all the complex options at once will take a lot of time to play his turn, and when are these options made available gradually, instead of turn length is the gradual progress stretching the session. For instance improvement of played character is unlocking new abilities for player. This process is gradual and player has enough of time to get to know gained ability before next improvement. This will make each turn faster, but every level of played character is bounded by a time spend achieving it. Implementing board games online will not directly solve the duration of the game, but unlike session state in physical environment, one in digital can be easily saved and finished at different time.

Complicated board games are often big boxes <sup>4</sup> full of cards, tokens, playing fields and other items that take not only some time to setup, but they require a lot of space on table as well. This issue was so significant it caused creation of whole new line of tables specifically designed for board games. Of course both of these problems disappear with introduction of digital tabletops.

Overall, we can say that bringing board games to digital environment has a lot of advantages, and only issue we need to tackle is correct integration of social interactions to have success and improve the quality of time spent playing board games. Naturally, it is not possible to entirely replace board games in physical environment, as they will

---

<sup>3</sup>[www.fantasyflightgames.com/en/products/a-game-of-thrones-the-board-game-second-edition/](http://www.fantasyflightgames.com/en/products/a-game-of-thrones-the-board-game-second-edition/)

<sup>4</sup>Game Tide of Iron is a box of five kilograms containing over 800 game items.

always have their place in entertainment, but there is a lot of space for improvement and change.

### 7.1 Board game mechanics

Following general mechanics <sup>5</sup> are used in different board games and their examples will be of help while describing particular functionalities needed for implementation of such games. Instances of disadvantages of these mechanics in physical environment are described along with them to highlight advantages of PC implementation.

**Acting** Player performs certain audiovisual action for other players. This kind of mechanic is often used in *party games*. Only drawing is fit for PC implementation, since any other input method relies too heavily on input devices.

**Action Programming** Player secretly plans out his whole turn and then the actions of all players are executed at once. Complex strategic games using this mechanic are often lengthy and players are discouraged from playing them, since it is a big time investment. However, in the case of PC implementation, game can be saved and finished later on. Also playing game items in secret often means playing them *face-down*, which means player has to remember all played items before finishing turn.

**Action Points** Player uses pool for his action and his turn ends once such pool is exhausted. For correct control over available actions, players are often either required to remember their progress in the play or to manage a number of tokens. PC implementation may be done in a way that player is visually guided through the play (highlighting available options and remaining actions).

**Area Influence** Player is rewarded for control over certain areas. These rules are usually evaluated at enter point in game states without direct control of user, so this counting being automated makes gameplay much faster. It is quite common that players forget to account for everything while counting their points and that has often negative effect on game experience.

**Area Movement** Movement can be done over adjacent areas of varying size or shape. When characters or units have longer range of movement, considering all possible moves and strategizing over them makes every play unnecessarily long, whereas highlighting possible destinations makes it a matter of seconds.

**Auction/Bidding** Players bid in-game currency on game elements to gain profit from such elements in case of win. Secrecy around bidding process slows the game down and it is possible for a player to unintentionally reveal the bid before is bidding over.

---

<sup>5</sup>As source for list of these mechanics was used list available at [www.boardgamegeek.com/browse/boardgamemechanic](http://www.boardgamegeek.com/browse/boardgamemechanic).

**Card Drafting** Player picks card from limited subset into his own pool for later use.

Games including drafting of cards have often issues with space on the table, especially when there must be at all times 20 piles of cards with text on them in the reach of every player. This problem can be solved by revealing these cards only when player decides to interact with them and in matter that will not clog the whole screen.

**Deck Building** Player starts with predetermined set of cards which he expands over the course of the game. Games with deck building suffer from constant need to draw cards and shuffle the deck, which takes its toll not only on card, but on patience of players as well, especially when it is common to have player shuffling the deck in middle of the turn.

**Dice Rolling** State of game is decided by dice roll as the source of randomness. Games with larger dice pool have sometimes issue with amount of certain type of dice and necessity of re-rolling while remembering previous results.

**Grid Movement** Pawns are moved on grid in predefined directions (square, hexagonal, ...). With strategy games where more units are included, player must remember which units player already used and/or which may be used. In PC implementation can be active units highlighted along with their available actions.

**Modular Board** Game board changes over course of game by adding or removing tiles. Modifications of board have negative effect on layout of game components on the table, since it is not always clear where the next tile might appear and it may require shifting everything every once in a while. Digital environment tabletop can be defined to be virtually infinite.

**Player Elimination** Player can be eliminated from the game and stay as observer only. These games are unpopular for when a player is eliminated, he/she has nothing to do for the rest of the game but passively watch others.

**Point to Point Movement** Unlike with grid or area movement, adjacent are only the points connected by line. Players need to keep track where which path leads and they might forget to consider some options, but virtual graph representation does not care where the pieces are actually positioned.

**Role Playing** Player's character improves over time or according to specified actions. Improving players character often includes piling or shifting tokens on the game board and that is prone to errors by pushing the table or the board and misaligning the tokens leaving players clueless about what was the actual setting.

**Set Collection** Player gains special bonus for collecting certain set of items. Collecting these sets is usually part of the middle to late game which can leave player unaccustomed to gained abilities and forgetting to use them. When gained ability is highlighted, player is less prone to forget about it.

## 7 PHYSICAL ENVIRONMENT COMPARISON

**Trading** Players can exchange items between each other. For players to exchange items between each other, they first need to have a notion about other players' possessions, which often includes running around the table and slowing the game down.

**Variable Phase Order** Certain actions may be prohibited in given turn. This may cause some confusion for players counting on options that are currently unavailable, but this state of the game can be on PC properly highlighted so no player will go surprised.

**Variable Player Powers** Each player starts with different setup or conditions. New game is hard to follow even when all players do the same things over the course of round, and it gets more difficult once each player has different set of abilities.

### 7.2 General game engine types

Game engines could be classified by their complexity into three levels [6]:

**Low level** These engines are either made with one particular game in mind or extremely general only to help with creation of system for the particular game. They basically consist of set of libraries, for example physics, rendering or scene graph to remove the process of the reinventing of the identical wheel over and over again. Their advantage is that they do not change very often and are very well optimized at what they do. As low level might hint, they are usually written in native code and offer little to no adjustments past the point of changing predefined settings. In these engines are programmers often interacting directly with used system APIs <sup>6</sup>. Examples of these libraries are *OpenGL* <sup>7</sup>, *PhysX* <sup>8</sup>, *DirectX* <sup>9</sup> and many others.

**Middle level** These engines could be described as frameworks for certain type of games, consisting of all necessary parts well chained together. Programmers usually start a new layer of the actual game and do not change the core until final optimizations. Such is the compromise between freedom and comfort for game makers. One of these engines is *jMonkeyEngine (jME)* <sup>10</sup>, which offers *NetBeans IDE* <sup>11</sup> based editor with set of tools for virtual scenes creation and manipulation.

**High level** Engines built as high level usually consist of one or more scripting languages. Scripts for such engine may not even be handwritten, and can be completely created in point-and-click GUI. Such simplicity speeds up production process, requires

---

<sup>6</sup>Application Programming Interface for access to contents of libraries.

<sup>7</sup>[www.opengl.org](http://www.opengl.org)

<sup>8</sup>[www.geforce.com/hardware/technology/physx](http://www.geforce.com/hardware/technology/physx)

<sup>9</sup>[www.en.wikipedia.org/wiki/DirectX](http://www.en.wikipedia.org/wiki/DirectX)

<sup>10</sup>[www.jmonkeyengine.org](http://www.jmonkeyengine.org)

<sup>11</sup>[www.netbeans.org](http://www.netbeans.org)

less experienced developers and for certain genres it is not even all that much constraining. To the family of these engines belongs *Unity3D*<sup>12</sup> or *Unreal Engine 4*<sup>13</sup>.

### ■ 7.3 Existing solutions

In this section a few existing board game engines<sup>14</sup> will be listed and I shall discuss their approach. While comparing these solutions, different attributes need to be taken into account. For one, there is availability of rule enforcing by the implementation. When there is basically no rule enforcement by the system, players are required to know them themselves and have to keep track of everything. On the other hand, when are rules completely watched by the system, in-house-rules<sup>15</sup> common while playing physical board games are impossible to be added.

Another parameter is difficulty of use of the system. For instance, there are engines offering great freedom in creative process, but their complexity makes them viable option only for experienced programmers. This is of course not necessarily bad thing as it depends on intentions of authors of mentioned engines.

#### **Thoth Engine**<sup>16</sup>

Thoth Engine is a typical representative of Game Engines focused only on one type of games (card deck building). Engine only loads predefined placement for used cards from configuration file and provides few operations over cards (flip, shuffle, rotate, ...).

#### **Battlegrounds Gaming Engine**<sup>17</sup>

This game engine works basically as point-and-click virtual table top on which you can place objects of your choice and set some simple behavior. The main issue with this engine is that players cannot be forced to follow the rules. Board is controlled by drag-and-drop over existing objects or you can call some simple action over them, for example flip or shuffle. Such engine is suitable maybe for board games prototype testing but creating a game ready to be played out of the box is merely impossible.

#### **Vassal Board game Engine**<sup>18</sup>

Vassal uses various wizards, dialog setting windows and configuration files so no coding is required but user can, in case of some properties, load custom classes and assign them to created objects. Similarly as with Battlegrounds Game Engine, the player

---

<sup>12</sup>[www.unity3d.com](http://www.unity3d.com)

<sup>13</sup>Showcase of development tools for Unreal Engine 4 [www.youtube.com/watch?v=M0vfn1p92\\_8](http://www.youtube.com/watch?v=M0vfn1p92_8)

<sup>14</sup>As reference list served collection provided on [www.battlegroundsgames.com/links.html](http://www.battlegroundsgames.com/links.html)

<sup>15</sup>In-house-rule is a modification of original game rules to certain extent for either freshening up the game that has been played too many times, or simply fixing design flaws in game. These rules are often wide known, shared by community and in some cases even added to re-editions of game itself.

<sup>16</sup><http://digilander.libero.it/zak965/thoth/>

<sup>17</sup>[www.battlegroundsgames.com](http://www.battlegroundsgames.com)

<sup>18</sup>[www.vassalengine.org](http://www.vassalengine.org)

## 8 GENERAL REQUIREMENTS

cannot be forced to respect and follow the rules, although his options can be restricted in more precise manner by adding certain traits to specific objects, e.g. a card deck can be shuffled but there will not be the possibility of shuffling anything that is in the selection (as in case of Battlegrounds Game Engine).

### ZunTzu <sup>19</sup>

ZunTzu is very similar project to *Vassal*. It as well does not enforce game rules on players and behaves very much as virtual table intended for online play sessions. In comparison to *Vassal* is this engine much easier to set up and contains integrated voice conference system, so no third party software is necessary.

### FlexibleRules <sup>20</sup>

FlexibleRules engine uses for making games set of editors (graphics, logic, code, mappings, etc.) to configure behavior of created entities and how they react to each other. That means rules of the game can be specified and players are forced to follow them. All rules in this engine are defined using set of tables and user is basically required to create whole structure on paper and then rewrite it in the engine. Engine contains its own scripting language for more specific definition of actions, but I found that language unnecessarily chaotic.

## ■ 8 General requirements

In this section I shall list requirements that I consider as essential for "finished" engine, i.e. one that is ready to be deployed for usage by public.

### ■ 8.1 GUI

Aspects of user interface provided by engine to creator are to be straight out used or inherited to more complex components defined by user.

**Basic input components** Implementation of elementary input elements, e.g. button or text area used to interact with active state space.

**Manipulation components** Implementation of components for manipulating scene, such as sliders or zoom controllers intended for fitting more elements on the otherwise limited tabletop.

**Basic output components** Implementation of components for displaying data in text format or images.

---

<sup>19</sup>[www.zuntzu.com](http://www.zuntzu.com)

<sup>20</sup>[flexiblerules.fulviofrapolli.net](http://flexiblerules.fulviofrapolli.net)

**Input event dispatch** Creating a system for user input dispatching, such as catching keyboard events or mouse actions. These need to be delivered not only to state space but as well to components in scene since they might want to interact with inputs immediately.

**Components for painting** System of components allowing to interpret user input as brush strokes for acting game mechanics.

**Styles** Implement component layout styles for easy definition of scene where creator can separate visual definition from logical structure.

**Animations** Implementation of various animation for movement, appearance or special effects, e.g. components on visibility change animating themselves in or out of the window. These effects should be dependent on direct call with delta time every frame and completely separated from game logic.

**Special effects** Implementation of effects that can be applied on components or even applied with animation, e.g. focused button with applied glow effect with varying intensity.

## ■ 8.2 Game components

Strictly boardgames related components to make easier implementing mechanics listed in Section 7.1.

**Dice** Set of dice components and generators of randomness.

**Countdown** Countdown components such as timers or hourglasses.

**Chat window** Chat window component that behaves in write-commit manner with appropriate rule presets.

**Boards** Set of playing board components that can be defined from XML file and have possibilities of path finding.

## ■ 8.3 Resources

Aspects of gathering and storing resources for final game. Work with data should be as simple as possible and provide sufficient amount of error detection mechanisms.

**XML parsing** Unifying XML loading into objects using precise definition.

**Dynamic *lazy* loading** Loading required assets first for faster ready to play times and being able to fetch any data based on runtime changes of game.

**Dynamic register** Register allowing storing various types in tree-like structure that supports lists and maps defined on the fly and/or at startup.

**Scene building** Scene should be built from XML file definition that possesses the same abilities as scene with attached CSS file.

### ■ 8.4 Network

For it is engine for playing games online, one of the main aspects should be cooperation of different clients over network.

**Initializing connection** Tool for initializing connection with other players via client-server-client model.

**Synchronizing content** Simple way to synchronize content with defined players and keeping consistent state of the game on all ends.

### ■ 8.5 Game rules

For tabletop engine is not important to force playing by the rules, and it is one of biggest advantages for experimental development on these engines, but for standalone game it should be vital to keep player on the tracks using restrictions and hints.

**Allowed operations** Implementation of system of dynamic rules that can be modified without recompilation of code, for example using XML structured files.

**Effects definitions** Each action enabled by rules has some effect and that effect should be modifiable to certain extent without recompilation.

## ■ 9 Discussion

In this part were board games classified from point of view of mechanics that may be used during their creation and what impact these mechanics have on implementation of such games on computers. Along with those were classified game engines in general and their properties in relation to board games. This was then expanded to cover existing solutions that aimed to achieve similar goal as this thesis.

Lastly was done general overview of requirements on created engine which should serve as guide while constructing feature list in future development cycles.



## Part 3

# Design

This part briefly discusses the design of architecture of this system. Elaboration of the specific implementation is described in the next part.

## 10 General architecture

Architecture of this engine is mostly event driven application, with dynamic code loading and behaviour programming from XML definitions.

It is generally focused on flexibility when modifications of game logic are concerned. However, it is for the price of use simplicity but that is intended to be solved in the future by GUI builder. Such builder should basically remove process of user writing XML files, and instead generate them from interactions in the editor.

Not only is the engine event driven, it is not intended for real time games as well. That will save operation cost with keeping game state consistent at all times on all ends, since every player may operate only limited amount of data and update them after state transition, which is very well controlled.

## 11 Event driven applications

Event driven applications are those where main communication channel is certain kind of event queue which distributes received events amongst designated listeners. This kind of approach is usually used for applications that revolve around user interaction with the whole system idle until event from outer source is received.

### 11.1 Advantages

Advantages of event driven applications.

**Decoupling** Event driven applications may be also callback driven, which means that the event has piece of code attached to it, and the code is executed during evaluation of the event, or at some point afterwards. This brings certain amount of flexibility to the system, since unit evaluating the event does not have to have any knowledge about evaluated event, and thus it is decoupled from source of the event.

**Extendability** Extendability of application that is event driven comes simple because once every major component in the system has instance of event handler and is

## 11 EVENT DRIVEN APPLICATIONS

subscribed to event processing. That allows newly added component to reach any other through events without interfering with application core. We can take this even further and move some components on different machines in separate code and nothing will change.

**Replay** It is not of importance what way event took to reach handler, space or time-wise. This way, we can have easily stored batch of events in a file and send them in the system as if they were produced by actually working component. That makes unit testing of event driven application easier for user input can be recorded in form of events and then replayed on testing machine after every change of build to verify that application still works.

### 11.2 Disadvantages

Disadvantages of event driven applications.

**Debugging** Event driven applications are more complicated to debug at times, because flow of the code being executed is interrupted in the event queue and it is hard to follow using classic debugging mechanics. This is even more complicated with introduction of anonymous callback execution where it is difficult to predict what will code exactly do.

**Efficiency** Distribution of events alternates between simple and efficient, and it is not easy to achieve both. For instance if we have only one handler and every listener receives every event, it will be simple to use but a lot of unnecessary checks will be done. We can work around that issue by having components subscribe to only those events, that they can possibly care about. However, that introduces complexity to the code and increases coupling because then the types of events in the handler have to be recognised.

**Generated data** Amount of data and objects in general created in event driven application can reach significant amounts that may slow the whole system down. If some system event occurs often enough, it may clog system with unnecessary supporting data while simple direct call on target instance would cost insignificant amount of resources.

**Naming** Messages within the system need to be described in one way or the other. This description is then used to recognize purpose of processed event in target component. Names describing events need to be for each purpose unique, which cannot be forced without centralized collection. We have then either option to increase coupling of application or rely on user of the system to chose new names that will not collide with so far defined name space.

## ■ 11.3 Application

For engine I decided to design are not all of the properties of event driven systems relevant. I will briefly mention here which ones are relevant and why.

**Decoupling** Using callback driven structures enables possibility to decouple for instance graphics scene from used technology and allow changing it without interfering with games already created. This proves very useful in case of 3D game engines where there is plethora of available technologies and advances in their development would make switching plausible option. That does not pay as much for 2D games and even though 2D graphics accelerated on graphics card would be probably faster, it is not necessary, at least so far. (engine performs relatively well and there is a lot of space for optimization)

**Extendability** There is present a single event handler in the architecture that will touch every event being sent. It is essential for this handler to not care about received events beyond filtering, which is defined externally and with emphasis on generality, which means events can be added reasonably freely.

**Replay** Replay-ability is viable option, since all events are strictly kept externalizable so they can be easily saved, and thanks to their timestamps, it is easy to reproduce everything that happened in the game at exact time it happened.

**Debugging** In this engine debugging is somewhat complicated because there is a lot of user defined behaviour. Also, it is not simple to keep track of the execution. Because of that, there is a custom logger which receives a report about nearly every unusual state so the user has as many clues for resolving issue as possible.

**Efficiency** Efficiency is not problematic in this engine, since the amount of generated events is quite small and those that are generated are already processed by event filter that is automatically applied on game logic.

**Generated data** Amount of data generated per second is not unbearable, but pooling of objects is definitely an option for later optimizations.

### 11.4 Modularity

Even though design of this engine is not as robust as it could be, modularity should be one of the properties it has, or at least to some extent. Modularity in definition by Eberly [2] follows five criteria:

1. Decomposability. Design allows decomposition of problem into subproblems whose solution may be pursued separately.
2. Composability. Design allows combination of modules into new system.
3. Understandability. Modules can be understood separately or together with other modules.
4. Continuity. Small specification change requires change of one or few modules. Changes do not affect general architecture of the system.
5. Protection. Abnormal conditions that occur in a module stay in that module.

These criteria then lead to principles that should be followed to ensure modularity.

1. Modules must correspond to syntactic units in the used language.
2. Every module should communicate with as few other modules as possible to reduce coupling.
3. When two modules communicate, they should exchange as little information as possible.
4. If two modules communicate, it should be obvious from their definition.
5. All information about the module should be private unless specifically declared public.

Not all these guidelines are followed during design of this engine, but they should be a general goal for all future modifications and all components of the system should get eventually refactored into a fully modular state. This is a long-term goal which is not worth fulfilling on changes and modules that may not last or are undergoing heavy changes.

## 12 Architecture layout

In this section, the whole architecture layout is described, without extra detail on specific blocks.

As you can see in Graph 1, the whole architecture could be split into four separate blocks - persistent, core, dynamic and event handler. Visible connections in the graph do not necessarily mean there are not other connections between different blocks, as they are more of a strongest connection, or intended logic connection.

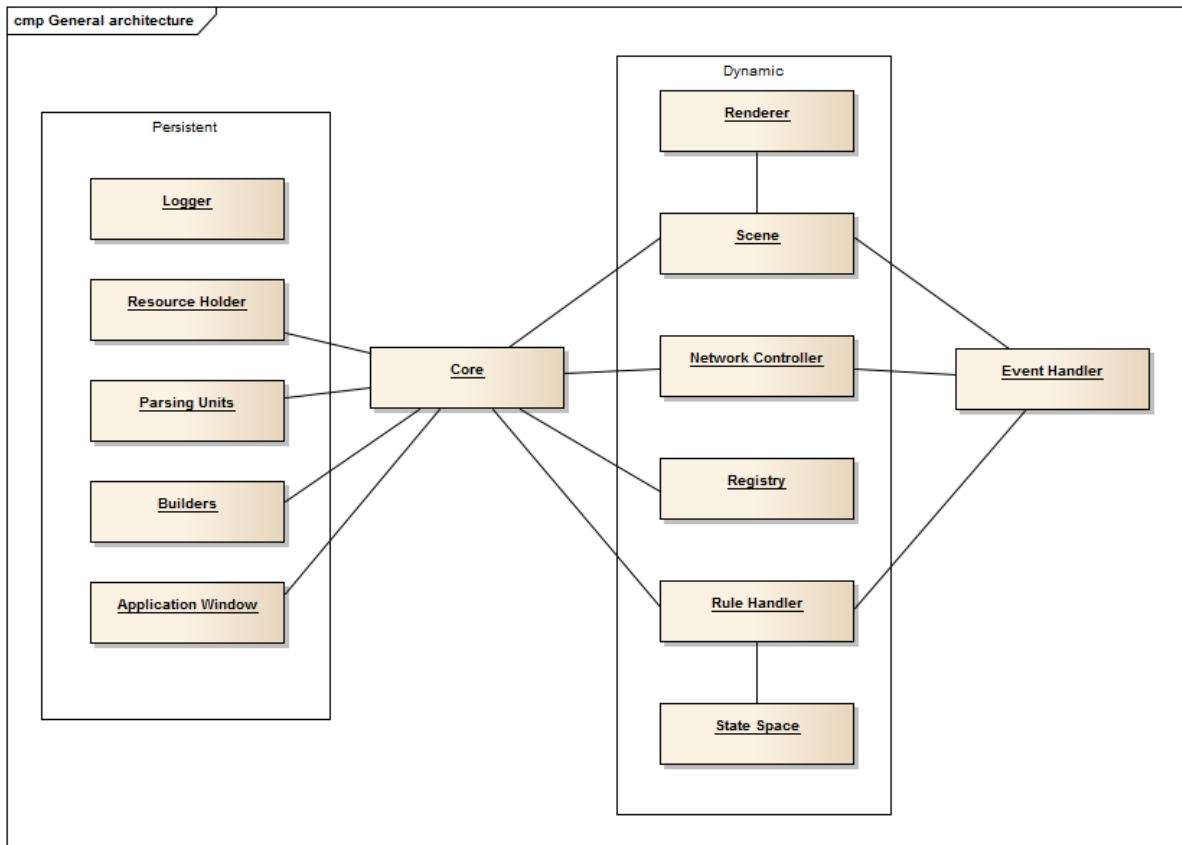


Figure 1: General architecture layout

**Core** Core block is to a certain extent the main class of the game that will initialize everything and then die. But that is almost everything that is required of it at the moment. There is need for this module to hand references to every other module based on their connections and start all separate threads.

Core may differ based on type of application, but for the most part, it will be the same process and thus there is no need for defining it as object with specific properties, because there would be simply no one to handle it anyways.

**Persistent** There are blocks in this section that are once created by core and then act as libraries or factory classes for dynamic parts of the engine. Generally everything that has connection to core may have in some way access to these, if they are not by themselves static already.

Blocks in persistent sections may have inner state but the difference between registry block and resource holder block is that inner state of the resource holder is not heavily accentuated by anyone. If we completely replace it for different holder, for example one using soft reference cache instead of hard reference one, it will go unnoticed.

**Dynamic** Dynamic modules have inner state that is, to a certain degree, of importance. This not only means other modules knowing about them will expect them to maintain specific state, properties, but as well that they may *live their own life* and need to be accessed in synchronized manner.

There should be certain separation between modules in this section but only from the point of view what user is supposed to be modifying. It is agreeable to have everything decoupled, but not if it means generating huge amounts of unnecessary calls over structures that will most likely never change, and if so, it will not be done by user of the engine.

**Event handler** Even though the event handler would nicely fit into dynamic section, I decided to keep it separated. The one important reason for it is that event handler should be completely decoupled from rest of the engine so it can be freely used by any added module as communication middle man without any ties to rest of the engine.

Job of event handler in itself is very simple. Accept event by synchronous call, enqueue event and hand it to all registered listeners with regards to its timestamp. You can see in Diagram 2 how interaction from user is passed to event handler.

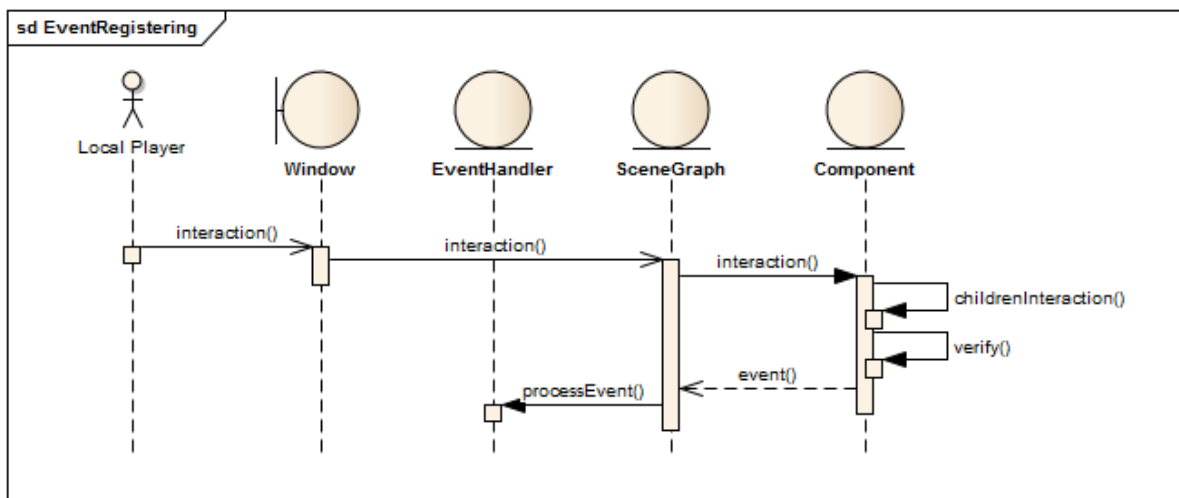


Figure 2: GUI input processing

## 12.1 Scene design

Scene is a module containing acyclic oriented graph of components and nodes that are tightly wired into the engine. As you can see in Graph 3 basic scene layout is quite simple. Scene is possibly tied to a window that generates user interaction events for it. Then it contains a root node that is able to process and translate actions from outside depending on implementation of scene nodes. This root then contains certain hierarchy

of group nodes that are extended by components to different degree of complexity to fit their specific purpose.

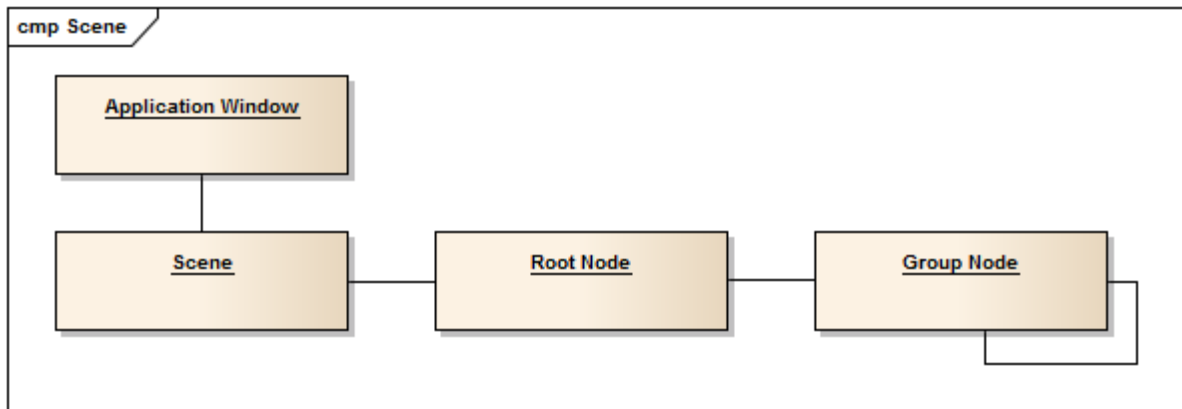


Figure 3: General scene layout

It is important to note that not only leaf nodes may contain rendered graphics. This may cause issues, as Eberly [4] describes, with manipulation of scene nodes. We cannot by default manipulate parents graphics without influencing children. There is possibility to avoid that by using grouping nodes that do not have graphics for manipulation of children with parent and in case parents graphics should be manipulated independently, it may be done so by accessing parents graphics representation node as it is shown in Figure 4.

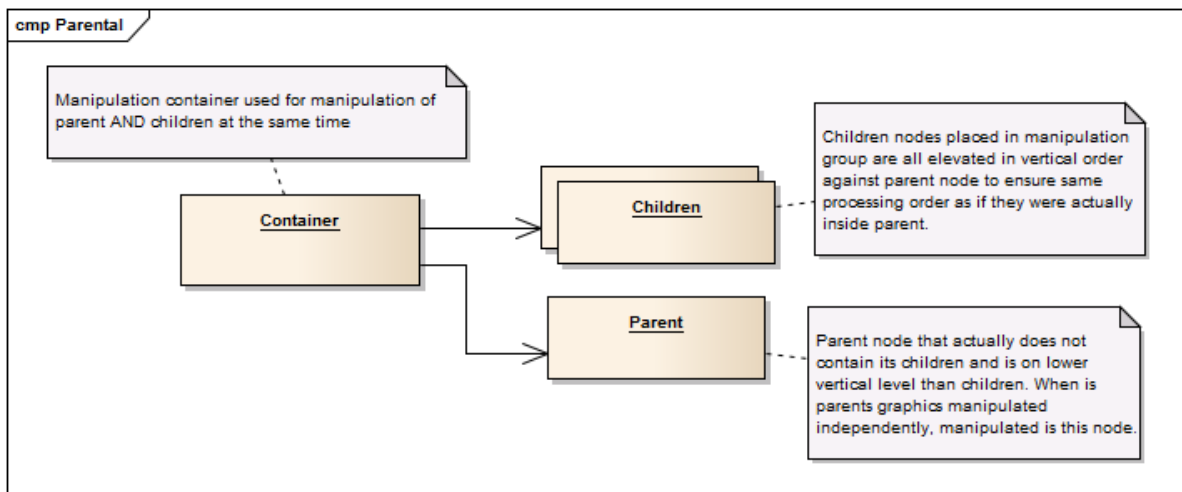


Figure 4: Manipulating parent without influence on children

This construct needs to be applied by user but it is possible to have it in the system. That is not important issue since such manipulation is not exactly common use case.

## 12.2 Logic design

Logic of the engine is contained in separate branch starting with module *RuleHandler*. In Graph 5 is shown the order of containment of specific logic elements. As top level element, there is a state space that could be related to definition of state machine states without its connections. Each state is then composed of rulesets that further contain specific logic, rules or actions, grouped in different ways to control the flow of logic evaluation.

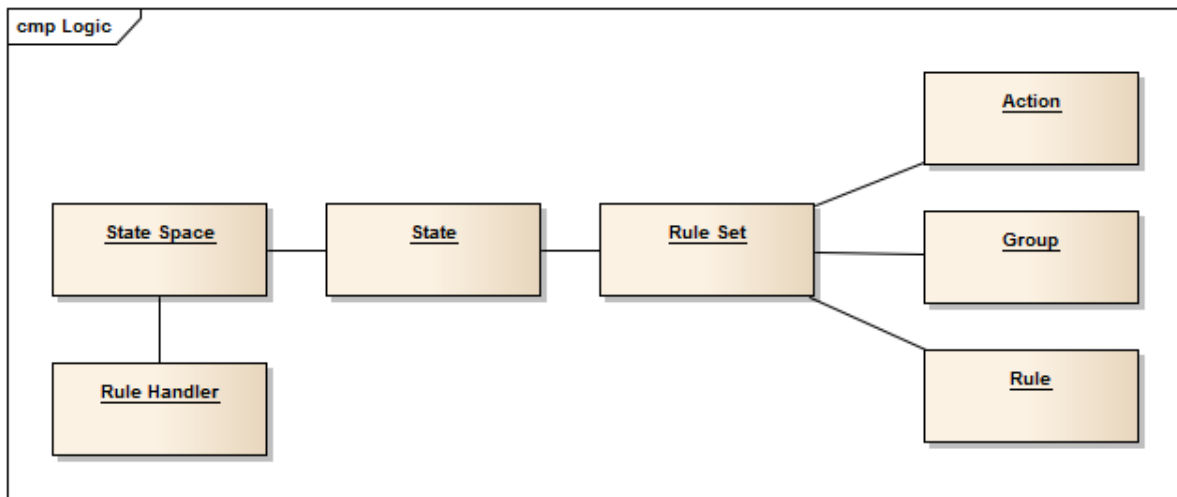


Figure 5: General logic layout

## 12.3 Data processing

Data loading from is done from generalized cached resource handler that has access to all builders and manages requests directly from path to object without the hassle with age of file and its modifications.

Data in the engine are of three different types in general. First, there are XML definitions that have their counterparts within the system strictly defined by XSD <sup>21</sup>. Second, there are styles structured similarly to CSS with specialized parser. Third is the imagery used for rendering process. All these types need to have their own defined access because of different used builder, loader or parser, which may make extension of loading process unnecessary complicated. That is done very rarely, though.

Since imagery is accelerated in rendering process, it is crucial to apply correct memory management on it. As Eberly [3] highlighted, it is important to keep used imagery in AGP memory that has better connection with VRAM and thus significantly increases

<sup>21</sup>XSD is language for XML file structure description. Further description can be found in wikipedia article [www.en.wikipedia.org/wiki/XML\\_Schema\\_\(W3C\)](http://www.en.wikipedia.org/wiki/XML_Schema_(W3C))



performance. This is better achieved by centralized caching unit that forces imagery to be accelerated.

## 12.4 Network synchronization

Diagram 6 shows the connection between server and client which is supposed to demonstrate that server is using for specific connections same implementation as client. That means the communication is always client-client, just server is here to accept multiple connections and create new client instance for each.

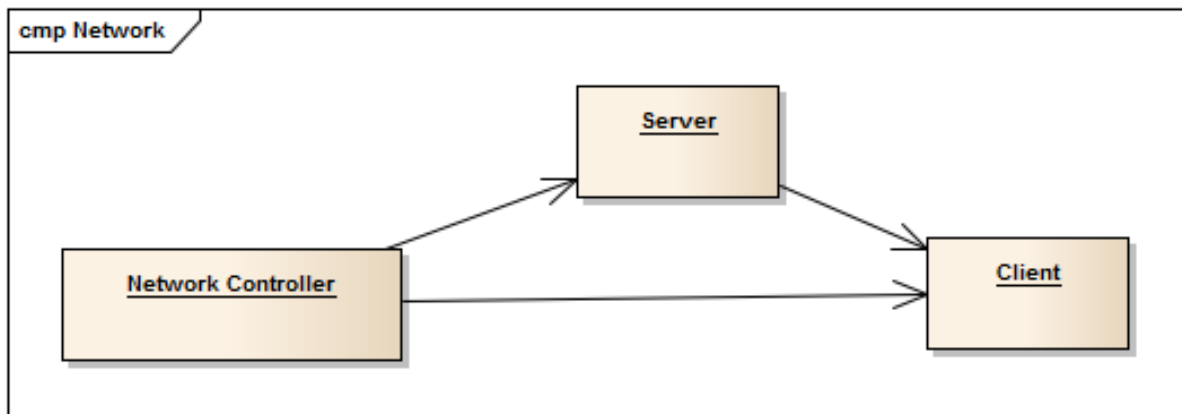


Figure 6: General logic layout

Since nobody has instance of created server and clients, except for event handler which does not know about them by design, communication with them must be done via events. Therefore server and clients have to check every received event for command that changes their behaviour.

## 12.5 Security

As Bartle stressed [1], client should *NEVER* be trusted with any game critical computation, and it should *NEVER* be given information that player on client is not capable of knowing. This is mainly because of possibility of any program being reverse engineered and modified for advantage of specific group of players. This applies especially to *java* programs, since compiled java code is for convenience of *JIT* in understandable form — no optimizations that we can see in C++ for example. Compiled source for *JVM*<sup>22</sup> can be simply decompiled back into human readable code that is nearly identical to what author wrote. In case of game logic and rules is this even worse, since they are not even compiled and anyone can edit them at will.

<sup>22</sup>Java Virtual Machine

It should be as complicated as possible to crack the game, and for that there are some important tools. For *JVM* byte code is designed *ProGuard* <sup>23</sup>, which will amongst other things obfuscate <sup>24</sup> given compiled java code, so it is much harder to reverse engineer. In case of game logic, there should be created binary wrapper for all the logic, registry and data in state of a simple encryption. This way will any cracker has to first reverse engineer the code and then crack logic definitions.

Since this scenario is bound to happen eventually, if the game gets successful enough, it is necessary to keep clients from data they are not supposed to have. This needs to be handled by game designers and basically cannot be prevented from engine by default.

## ■ 13 Discussion

General architecture design was described and all decisions were to certain degree justified. This design overview should be perceived as guide to look at final implementation from correct perspective. Programmer oblivious to intentions behind some design choices could easily *break* the whole architecture and severely complicate future development.

The goal of this design is to guide programmer to right places when modifying the engine so time spent reading code is reduced to the minimum. Specific implementation parts are described in depth in Part 4.

---

<sup>23</sup>[www.proguard.sourceforge.net](http://www.proguard.sourceforge.net)

<sup>24</sup>Obfuscation in programming generally means modification of code in way that does not influence execution but reduces its readability by humans to make reverse engineering more expensive process.

## Part 4

# Implementation

This part is about specific implementation of the engine and brief description of how some of the parts are used. More detailed reference manual is located in appendix.

## 14 Scene

Scene is the main part of the visual representation of the game. It contains various components in hierarchic structure that have generally abilities to render themselves, react to user inputs and allow game logic to modify it. These responsibilities are split amongst abstract graph nodes in line of inheritance of usable components. Whole scene structure could be split in abstract node part, component part, root node that is not available to user and scene object itself.

Scene object as such contains final *RootNode* and in that node are inserted further scene elements. Each *Scene* object, if it is to be displayed, should be bound to some *ApplicationWindow* on which it will reside and to some *Renderer* responsible for regular draw calls. Since scene is accessed from different threads simultaneously, it needs to have lock system 14.6 for prevention of concurrent modification errors.

One of the main features of scene is listening to events from event handler and passing them synchronously into scene graph with response queue attached. This queue serves for limiting amount of places that can generate events. This way all response events coming from scene must go through scene object and may be controlled there if needed be.

### 14.1 Graph

As you can see in Diagram 7, scene is composed out of instances of *SceneNode* classes, but actually available scene components for user are those extending *GroupNode*. All the nodes in the hierarchy prior to that are simply for separation of responsibilities and better code readability.

**Abstract section** Abstract section of the graph is composed of four nodes — *SceneNode* being predecessor for everything in the scene, *PaintableNode* maintaining properties and requirements related to rendering, *StyleableNode* for modifications of nodes using styles and *GroupNode* for building trees out of scene nodes. Even though it is not intended, user can extend any of those and build some functionality around it. That node should still be able to be placed in some group node, however, it will lack all the

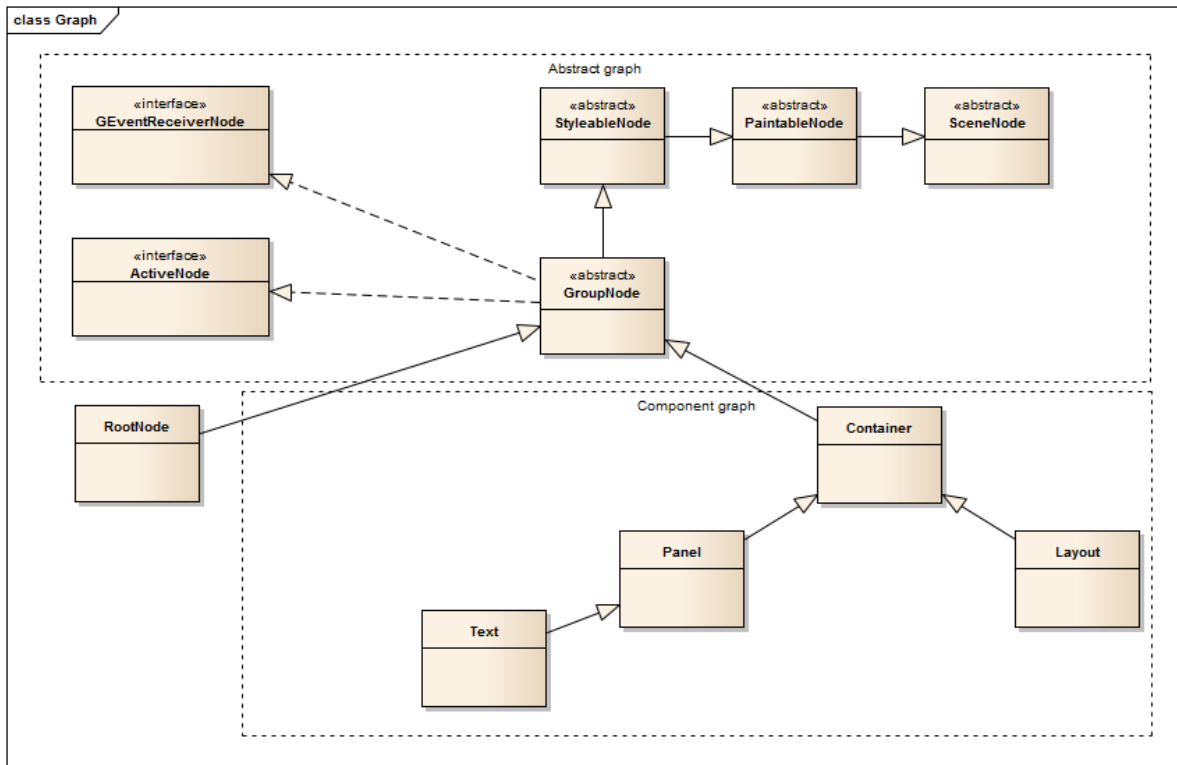


Figure 7: Inheritance schema of implemented nodes and components

basic properties of mentioned abstract nodes. More in detail in appendix Section B.2.

**Component section** Component section of the graph contains also four elements — *Container* for basic positioning, *Panel* for simple styling of background imagery, *Text* for displaying of texts and *Layout* for cooperative positioning of multiple components on the same level. This, contrary to abstract nodes, is list that should be extensively expanded during later development of the engine depending on requirements. These components can be defined by user in the XML scene graph, unlike nodes from abstract section. Each component has more detailed description in appendix Section B.3.

**Root node** Root node is special instance of group node having extra capabilities when managing changes in the graph. For example, when new node is added to styled scene, it needs to get all applied styles that were set to the scene. Root node has all of those styles stored, so added node initiates recursive call from its parent all the way up to root recording nodes on the way and then having all scene styles matched on that path. User defining multiple nodes of this kind would break the scene and it would not be entirely visible, since such state is unexpected, and thus errors caused by it are not adjusted to lead to its resolution.

## ■ 14.2 Selection

Selection of nodes in scene is heavily inspired by CSS, but it still has some minor differences in use, and lacks the general amount of features CSS selectors offer. When element in HTML scene is operated by javascript to change its appearance based on user defined state, it is usually done by adding or removing of class that carries the style difference. In case of this engine, the class is defined statically and cannot be changed at runtime, but states that an element can enter are modifiable at construction time and those can be entered from logic of the game.

Process of applying styles based on defined selectors follows very simple rules. Selector is built out of selection chain and then it is iteratively applied on nodes in the scene while checking last unfulfilled member. For more details about specific available members see selectors appendix Section D.1.2.

When a selector member is checked on scene *StytleableNode*, it can result in two states. First, member passes the check as valid, which means we are on the right track, selector is progressed to next member if there is one, or applied if validated member was the last one. Second, member does not pass the check and selector is not progressed. In both cases is selector passed to child nodes for further evaluation in non-progressed state. Pass in both cases is because of starting point of selector may not be root of the scene but any element.

Important thing to note is that selection chain is defining only elements that have to appear on the way through scene. Using selector *#root Panel* will that way match on all panels inside of the scene. There is no direct child selection option at the moment and thus class specifications have to be used for these occasions.

For selection attribute style ID is added requirement that each ID has to be unique across whole scene for optimization purposes. There is fixed style ID *#root* attached to scene *RootNode* and thus cannot be used inside of the scene for any other node.

## ■ 14.3 Styles

Scene styles are very similar to CSS in all ways imaginable. These styles can be defined directly in XML scene definition as attributes of elements in the scene, but mainly, they should be defined separately in stylesheet definition using scene selectors. This is the same kind of separation of presentation, behaviour and structure that was propagated back in the day for CSS and HTML. The main idea stays the same, but some properties are still better mixed up - same as in CSS and HTML.

**Style in scene** Mixing presentation in scene definition is still possibility despite the fact it is not the best practice, simply because of definition of small scene fragments that would generate unnecessary stylesheet files just for one or two attributes.

**Behaviour in style** Style definitions can have, like in CSS, defined behaviour states like *hover* or *visited*. I took it a bit further and allowed user to define any additional state which can be changed either in implementation of component or inside logic of the game. There are of course defined default states that can be styled out of the box without the need of additional specification.

Every component may have set any amount of properties and it does not really matter whether is that component using them or not. Every component will look upon only those properties that are predefined in it, and everything else is ignored. By design, no property should be mandatory. For more information about specific properties see available properties in appendix Section D.2.

### ■ 14.4 Style properties

Properties are internally stored inside of map of general objects and are transformed on first use into correct class. They may be as well stored in correct class right away, but it is not done automatically. There are some predefined access types:

**Integer Double Boolean** Type access will look for appropriate instance in the map, but if it is not found, will try to take string instance and parse it to target type. When parsing is done, property is replaced in style and next time will be retrieved right away.

**String** Strings are basic type and everything automatically loaded will be in the beginning stored as a string.

**Color** Color access similarly to previous types is first retrieved as instance, and in case of fail, one of the following representations is attempted to be parsed. Color in classic hexadecimal definition *#RRGGBB*, expanded version *#AARRGGBB* where *AA* is hexadecimal value for alpha channel, or word name of color from predefined testing palette *white—black—red—green—blue—yellow*. Alpha channel is inverted so when it is by default not set, visibility is on maximum.

**Relative integer** When a property has units attached to it, it can be retrieved as *propertyInPixels* which will relate stored property according to its units to given value.

### ■ 14.5 Scene states

Nodes in scene can be in different states at different occasions, but they are functionally divided into two possible spaces. These spaces are *Logic* and *Interaction*. They are managed the same way, but they are defined differently and system reacts to them differently as well.

**Logic states** Logic states are defined by inner logic of the scene and work based on inner implementation of the specific node or component. They still may be set from outside using game logic, or even require it, but they have generally special treatment and purpose outside of specific game. As an example of those are *hover*, *click*, *disabled* and *invisible*. In case of *click* and *hover*, the difference is set by the manner component is entering and leaving those states. On the other hand, *disabled* and *invisible* are always set from user logic, but they influence the scene beyond just changing applied styles.

**Interaction states** Interaction states are all user defined states that are entered and exited from game logic. They are defined from the scene, so styleable components treat them purely as a map of state names with styles attached and without any other behaviour. Entering such state in logic of game will just add its style to currently displayed one.

**Priorities** There are priorities in place to keep behaviour of these states consistent and predictable. For one, all logic states are prioritized over interaction states. That means possible overlaps are overwritten by values in states in logic space during the process of recalculating applied style. The other priority is within state definition, where states are evaluated in order they were defined. Keep in mind that order of definition of logic states is determined by hierarchy constructor calls, where super is called before body of constructor, so states defined in inheriting classes will always have higher priority.

## ■ 14.6 Locks

As mentioned before, scene being accessed by different threads at the same time. Namely, render thread while putting current state of scene on the screen and event handler while distributing interaction events through the game. Number of threads to be synchronized is mostly reduced thanks to event handler use, since application window events and network events are dropped there and evaluated asynchronously.

Every time some action is done over scene, it should be locked in *try* block using appropriate methods over scene, and unlocked in *finally* block to ensure consistency of the lock. Unlike with classical lock, scene can be locked by the same thread multiple times without blocking itself because of owner of the lock being checked. Some core methods that pose danger in concurrent access are already protected by lock, but user modifying code of the engine should be aware that it might be necessary to add them.

As for performance, early profiling does not show any slowdown over scene locking, and if that problem arises, it is still possible to modify lock system to read-write locks. This system would allow user to lock scene for read and thus not block renderer. First thread locking scene for write would block new read locks, wait for current ones to finish and then proceed with its job. This would save insignificant amount of wait time, as most of the current locks is distribution of events that do nothing to scene or do

selections.

## ■ 15 Events

Event processing in this engine is done using single threaded event handler which receives events from various event generators that contain its reference. I chose this approach to avoid as much of synchronization problems as possible, and when this one thread would get blocked by expensive operation, there is always possibility to create separate thread and report result using another event. This approach is demonstrated in network event, where connection that does not happen immediately blocks event processing in the scene, so new thread reporting result using message event is created.

For more information on specific events and their implementation, please, see Section C.1.

## ■ 16 Logic

In this section is discussed how game logic implementation processes defined state space.

### ■ 16.1 Rule Handler

Logic of the game is processed based solely on events in the game. That means no logic is executed when there are no inputs from user. It is possible to add time event generator that will keep game at least a bit in sync with real time, but that makes sense only in order of seconds. Any faster processing would be just inaccurate and it would flood system with mostly dropped events. In case you would think about implementing something faster, consider different engine altogether, since that sounds more like real-time application.

Rule handler holds resources that might be needed for actions and rules contained within the state space. Those are accessed by instance every evaluation element holds. Besides all resources, it also holds used state space with game logic and has registered all states from it. Upon load of state space are all of those states stored in local map and those within *init* definition are entered.

When rule handler receives an event, it is passed to each active ruleset within each active state. This execution may be interrupted by state exiting itself during its processing. That is common behaviour and it will result in runtime exception being thrown, caught and resolved into skip of remaining rulesets from that state. In case state is being entered, its setup is postponed until all rulesets from executed state are processed.



## ■ 16.2 State space

Whole state space consists of states that are not connected to each other in any way. One might expect some kind of transition edges, but that is not the case. State space in this engine is designed in way so game can be in any number of states at once. Entering and exiting of registered state is solely controlled by logic in active states as a response to some event.

State space is consisting out of three parts. First, setup rulesets that are evaluated on state entry. Second, active rulesets that are evaluated on received event while state is active. And at last teardown rulesets that are evaluated when state is being exited. Any of these collections may be empty. It is important to have correct setup and teardown of states. First reason would be debugging, as when you reload scene in middle of the game, you need each active state to setup back into consistent shape. Second, and probably more important reason, for better configuration readability. It will be easier to read when you on entering to state setup everything necessary and when leaving clean up after yourself.

## ■ 16.3 Rulesets

As mentioned before, each state is made out of rulesets. Ruleset is collection of rules and actions that are stored in tree structure in root group element. This element is set to *and* operation, as we expect user to want execute contained element in order, all of them and in case first fails terminate.

Each ruleset contains its own variables map which could be compared to local variables of a scope in java. This variable pool is cleared every time before execution of ruleset and its contents are shared amongst all rules and actions within. That allows them to pass values one to another.

## ■ 16.4 Action vs Rule

There is almost no difference in implementation of action and rule. The main distinction of these two interfaces is name of the called method, which has also no practical implication. Reason there are two interfaces for the same thing is to remind user of the engine that rulesets should be constructed as condition-consequence pairs or groups and not a wild mix of actions and rules. Basically, user should create ruleset, insert some rules at the beginning, some actions at the end and possibly wrap all this into group. Of course more complicated operations cannot fulfill this pattern, but the idea should remain.

## ■ 16.5 Evaluation element definition

To preserve generality, all evaluation elements are set up using set of *param* objects and it is up to specific dynamically loaded element to pick what it wants from loaded set.

Defined action might look like:

```
<action class="StoreData">
  <param name="location" access="reg" value="game.data[%].%"/>
  <param name="location:wildcard" access="var" value="id"/>
  <param name="location:wildcard" access="reg" value="setting.usedMap"/>
  <param name="data" access="const" value="iteration%"/>
  <param name="data:wildcard" access="var" value="counter"/>
</action>
```

**Name** In this example, you can see definition of action that will be loading class *StoreDataAction*. This action takes two parameters, *location* and *data*. These names appear in *param* name attribute and serve for specification to what inner input should be acquired data assigned.

**Access** As you can see, every *param* has its own access defined. It should be one of three recognized access types, *reg* for registry, *var* for variables of ruleset scope and *const* for values directly specified. These access types decide how is value interpreted. In case of registry access is value interpreted as address to item in registry. For variable access will be value used as key in variable map. Lastly for constant is value attribute interpreted as final value, but it can also tell the element to load some predefined object. For example *LoadEventRule* used constant definition as signal to use input event.

**Value** As mentioned before, value depends mostly on setting of access type. If it contains percent character, wildcard is expected for that parameter, or more wildcards for every percent character.

**Wildcard** Wildcard definition is *param* with name of input or output it belongs to and with suffix *:wildcard*. This will make this *param* to evaluate as first and replace with its result first occurrence of percent character in target *param* value attribute. This way can be constructed not only dynamic paths to registry, but variable strings for text inputs. If we take example above and assume following: registry contains *setting.usedMap* with value *foo*, variable space contains variable *id* with value *5* and variable *counter* with value *7*; defined action will store string *iteration7* on address *game.data[5].foo* in registry.

More on implemented rules and actions align with their exact specification in appendix Section E.2 and Section E.3

## 17 Registry

Registry module is for storage of application data across all game states. It could be related to global static variables in java, but in this case it is more like global game state storage and not an abomination that should be frowned upon.

### 17.1 Registry keychain

Registry keychains are here for accessing specific parts of registers using object-like tree hierarchy. Every registry item is stored in its parent either in list or in map for different ways of access it.

When addressing items in registry, every level is separated by dot, with exception of list address that is right after map key, where dot is not necessary. Last defined element will be the destination or source of the value.

### 17.2 Value storage

Every *RegistryItem* may have a value of undefined type assigned to it and that value is accessed by calling for keychain of its wrapping Item. It is important to call read only on values that have been previously set, otherwise most elements will fail on undefined data.

### 17.3 Map

In registry item may be assigned a map *String:RegistryItem*. Accessing elements of that map is done by separating key on way to specific depth. For example *parent.child* is accessing element that is stored in root map under key *parent* and from map of that element *child* that is stored under key *child*.

### 17.4 List

Registry item may have a list of registry items inside addressed by standard indexing like with arrays. For example *element.child.[5].bottom* will access key *element*, then item with key *child* from which item on fifth position in list and at last element with key *bottom*. It is valid for one level of list to leave out the dot, but not for more than that one level. Example: valid *element[14].child*, invalid *element[14][16].child*. Corrected invalid case would be *element[14].[16].child* or *element.[14].[16].child*

**List indexing** List being indexed as array has couple of extra options that can be used for item selection.

**First** First flag that serves for indexing of first item of list and works basically like indexing with *element.[0]*. Use of first follows example *element.[f]*. Its purpose

is just to increase readability of definition. This marker must be first thing in indexing brackets!

**Last** Last flag works analogically to first flag. Use of last follows example *element.[l]*. Its purpose is to remove the necessity to load size of the list into variable space, decrement it and then use wildcard to paste position into address. This marker must be first thing in indexing brackets!

**Add** Add flag denotes in case of structure manipulation that item should be added and not replaced. Usage of add follows example *element.[-a]* where dash stands for position definition by number, first or last flag (*[4a][fa][la]*).

**Replace** Replace flag stands for element replacement. Usage is similar to add with difference that it is required for replaced element to actually exist. Example is *element.[-r]* where dash stands for number, first or last flag.

## 17.5 Example

Correctly defined XML file with registry definition could look like this:

```
<root>
  <map key="settings">
    <map key="resolutions" value="16:9">
      <list value="720p"/>
      <list index="1" value="1080p"/>
    </map>
  </map>
  <map key="round" value="0"/>
</root>
```

Every registry record has to start with root element. This is for unification and clarity that is necessary when merging registry together. As you can see, there are used *map* and *list* tags. The main idea is, as much as confusing it might seem, that registry item name is where that item belongs, and not what it is. That way defining *list* element means it should be put in parents list. For list, when index is not defined, it is automatically assumed item should be added at the end of the list.

## 17.6 Merging

Merging of two registry  $A.merge(B)$  records is done by systematic comparison of contained items, adding of items that are in  $B$  but not in  $A$  and overwriting elements that are in both. when overlapping items are found, it is not just replacement of items themselves, but only value is replaced and comparison is moved on children, where is done the very same process.

## 18 Rendering

Renderer starts rendering once it has defined component it is drawing on and scene that should be drawn onto it. Along with those values it needs to have updated dimensions, which are kept in consistent state with component by listening to *WindowGEvents* and loading new dimensions out of them.

Rendering is done in separate thread because its read only operation and can possibly work with scene along with other threads that are responsible for different parts of the engine. Renderer is configured to strive to follow strictly 60 fps, which is more than enough for mostly static game. The accuracy of precisely 16.6Hz is achieved by using sleep for break times larger than 1 millisecond where sleeping period is reduced by 500 nanoseconds. This will result into processor waking renderer up somewhere around target time and the rest for precise timing can be busy waited <sup>25</sup>.

### 18.1 2D graphics acceleration

2D graphics does not have to be demanding at all, but only when it is computed on graphics card or done in some low level language with good optimizations. Unfortunately in case of this engine, it is neither.

For classic rendering process is used *awt* <sup>26</sup> *BufferedImage* but that is not enough for rendering complicated scene 60 times a second in full HD resolution. First, buffered image is held in RAM which is way too slow for quick operations and constant flow of textures and imagery. Second, CPU is unsuitable for filling large surfaces of image just out of principle <sup>27</sup>. This number grows with every panel that has background color or image and it can get beyond capabilities of low budget CPU to handle this amount of changes in 16 milliseconds. For that very reason at least basic operations, such as *blit* <sup>28</sup> routine.

**Volatile Image** *VolatileImage* <sup>29</sup> is *awt* image representation that may be depending on the platform loaded in VRAM <sup>30</sup>, which performs for graphics operations significantly better. This image needs to be created specifically for component it will reside on and more importantly, it needs to have verification of lost content in place. This image, unlike *BufferedImage*, may at any point of rendering lose its contents and rendering

---

<sup>25</sup>Busy wait usually refers to looped execution that has no other purpose than stalling the processor. No work is being than but system resources are being consumed.

<sup>26</sup>Abstract Window Toolkit from java is heavy weight API for rendering GUI and 2D graphics.

<sup>27</sup>If we take full HD resolution 1920x1080, we have to visit 2,073,600 pixels per layer. This basically means we have on single thread execution running on 3.5GHz CPU about 28 ticks for each pixel per frame for all layers

<sup>28</sup>Term originating from BITmap BLock Transfer standing for copying of rectangular areas between bitmaps.

<sup>29</sup><https://docs.oracle.com/javase/8/docs/api/java/awt/image/VolatileImage.html>

<sup>30</sup>VRAM stands for general video RAM without specification of further technology.

process needs to start again. For that reason is whole frame rendered in volatile image outside of component and when its contents are verified, it is rendered on component using classical draw image call that has guaranteed success.

**Accelerated graphics** Once on VRAM, blit operations can be hardware accelerated using *OpenGL* support in java. This does not require any special treatment except for use of *Graphics2D* class for rendering instead of usual *Graphics* class. The only thing needed for enabling accelerated graphics is to set system property *sun.java2d.opengl* to *true*. There are some graphics cards that will not pass through hardware check for accelerated graphics and to solve that there is a bit of a ugly hack, where whole HW check is disabled by setting environment property *J2D\_D3D\_NO\_HWCHECK* to *false*.

### 18.2 Updates

In any real-time game engine would be present update call for keeping all components of the game in same point in time. In this update is usually used delta time from last update, so all processes can take that into account. In this engine, there is this call as well, but its purpose is solely for animations within scene. This update is called before each render call to progress all animations in their proper place. This call should not be used for anything related to logic of the game!

### 18.3 Application window

Renderer is tied with an *ApplicationWindow* instance, which is general wrapper for *JFrame* representing window and some canvas component you can paint on. It also contains correction style to compensate window decoration on screen. This style basically makes drawing surface smaller for renderer so game logic does not have to take care of that.

To this canvas component of the application window is registered mouse event generator that takes *awt* events and translates them into events of the game. On context frame are registered window and keyboard generators for translating of resize and key press events. This translation of events may be used when porting the game on different platforms, for example with touch screens available, since received tap can be translated as mouse click without even touching game logic and maintaining the same functionality.

## 19 Loading

Loading is done through resource holder that is maintaining all loaded results in hard reference cache <sup>31</sup>. Every loaded file type has its own access methods that deal with all the hassle around and user can just call load with path attached. This customization of

---

<sup>31</sup>Hard reference cache will always contain stored values, compared to soft cache that may delete some content that is least likely to be used.

calls makes extension of holder more complicated, but dealing with builders and parsers every time file is loaded is unnecessary.

## ■ 19.1 XML

Loaded XML files are parsed using JAXB bindings. These work based on XSD schema that defines expected structure of XML file. This definition is used to generate java class hierarchy in which are all the data from loaded XML inserted. Instantiated structures filled with data are then processed by appropriate builders and stored in game structures.

There are four ways how to approach conversion between java XML classes and system.

First, there is a designated builder that knows everything about both parts of the conversion and will instantiate and transfer all data from one side to another. This makes loading centralized and decoupled from system itself, but it is much more complicated to extend behaviour on either end.

Second, where designated builder knows what classes to instantiate, but all the setup is done inside of new instance using given counterpart. This approach makes builder much simpler, but the complexity is just shifted further in the system. Adding new properties or objects is easy, since user just adds instantiation in builder and handling of conversion is responsibility of created instance which knows very well what to do. In that case, we have centralized everything about functionality in system class and builder is more or less oblivious to it.

Third would be leaving even the instantiation on system classes, where builder only creates, or gets root of the structure and hands the definition to it. This root then know what to instantiate and how. Any child will do the same and so on. This makes builder pretty much decoupled from whole structure and can be generalized using some interface to the point whole application has only one builder for everything. In this approach, there is a lot of knowledge expected of user and extension of the system is completely in his hands.

Fourth could be described as simple handing XML structures into the system as they are and anyone who likes takes data from it. This makes the XML structure deeply rooted into the system and makes any change complicated, since removal or change of property can cause errors all over the system.

In this engine are used three listed approaches. Namely first for loading of registry structures where user expansions are simply not intended at all. Second approach for scene construction, since user should be able to fairly easily add components. First and fourth for logic loading, where logic itself does not change at all, and parameters for

actions are handed as they are, since those will not change either and structure carrying same information would look pretty much the same anyways.

## ■ 19.2 Styles

Styles are loaded using in house parser, since not all features do match with CSS, and expanding some existing parser would prove more difficult than creating new one. Definition of this style is further described in appendix Section D.

In this example is shown general structure of styles definition file:

```
#root * Panel:hover, #menu{
    background-color: black;
}

//include from different file
#include "../data/include.sts";

/* This definition is commented completely
.class Text{
    font-size: 80%;
}
*/
```

## ■ 20 Network

Network controller is as simple as possible. There are two entities that user can instantiate and they will then work separately without any further interaction with their instances from system.

Every platform that is connected to someone will have set client ID that is unique between all clients connected to same server, which can be used for game logic decision making. Once this ID is received in event handler, it is attached to all passing events as source, but only in case these events do not already have an ID.

### ■ 20.1 Server

When server is created, an attempt to start socket on one of the preferred ports is made, and if it is successful, server will sit on the socket waiting to get started. After its thread is started, server subscribes itself to event handler calls. Before entering the main server loop is sent a message with client ID set to zero.

In main loop is server listening on created socket for incoming connections and for each that occurs is done initialization procedure. This procedure consist of creating new client thread that will directly communicate with its connected counterpart, then



number for that connection is generated and sent to that client so it knows its ID. At last is sent a message containing information about new client being connected, so all currently connected clients may act on it.

## ■ 20.2 Client

Client class may be used not only inside server, but as well separately for initiating of connections to servers. These are done similarly on preferred ports, but hostname is required as well. This hostname must be resolvable into java *Inet4Address*<sup>32</sup>. When is client connecting to server, all preferred ports are tested with one second timeout until one of them passes or everything fails.

When connection is established, client has to be started, which involves subscription to event handler — same as with server start. Communication data streams have to be created, since they are necessary for event transfer over network.

Once inside its main loop is client blocked on read of object from input stream, and once an object is received, it is converted into event and passed into event handler for rest of the engine. Network flag of all accepted events is turned off so it is not sent again when that event appears in client again from event handler on this side of communication.

When event handler is sending an event through client, it must be first marked as network event. In case this platform did not yet receive its ID, client will busy wait for it to be set, since sending events without correct platform ID would cause opposing clients event handler to consider sent events as its own. This busy wait blocking should be very short because client ID is one of the first things that appear in communication.

## ■ 20.3 Issues

There are three main issues with current implementation of network communication. First one would be that disconnected clients are not watched, so when a client dies, server has no way to find out about it and let everyone else know. This would be deliberately game breaking, because when everyone is waiting for disconnected player to make a move, but that player is not only no longer there, but cannot even reconnect. This never happened during testing, but it is issue that needs to be addressed and resolved, since it may happen on unstable connections.

Other issue is with sending events over the network. When events are distributed, they are always sent to everyone on the other side. For server is such behaviour relatively fine, but client needs to have every message redirected by server for it to reach everyone. It would be convenient for client to have option to pick target of event being distributed.

---

<sup>32</sup><http://download.java.net/jdk7/archive/b123/docs/api/java/net/Inet4Address.html>

Last issue is that there is no guarantee that events will arrive in destination in order they were sent. This can prove quite problematic, because if server sends message for transition to different state on client and then data to be processed in that state, it can happen that they will arrive before client enters new state and will be dropped. This is currently in implemented games solved by two sided communication where server sends command for transition and waits with sending of data for response from new state.

## ■ 21 Tic-Tac-Toe

As first example game I chose hot seat *Tic-Tac-Toe* (further referred as *TTT*), since it is one of the most basic games imaginable and thus can be used as tutorial game for new users of the engine.

### ■ 21.1 Rules

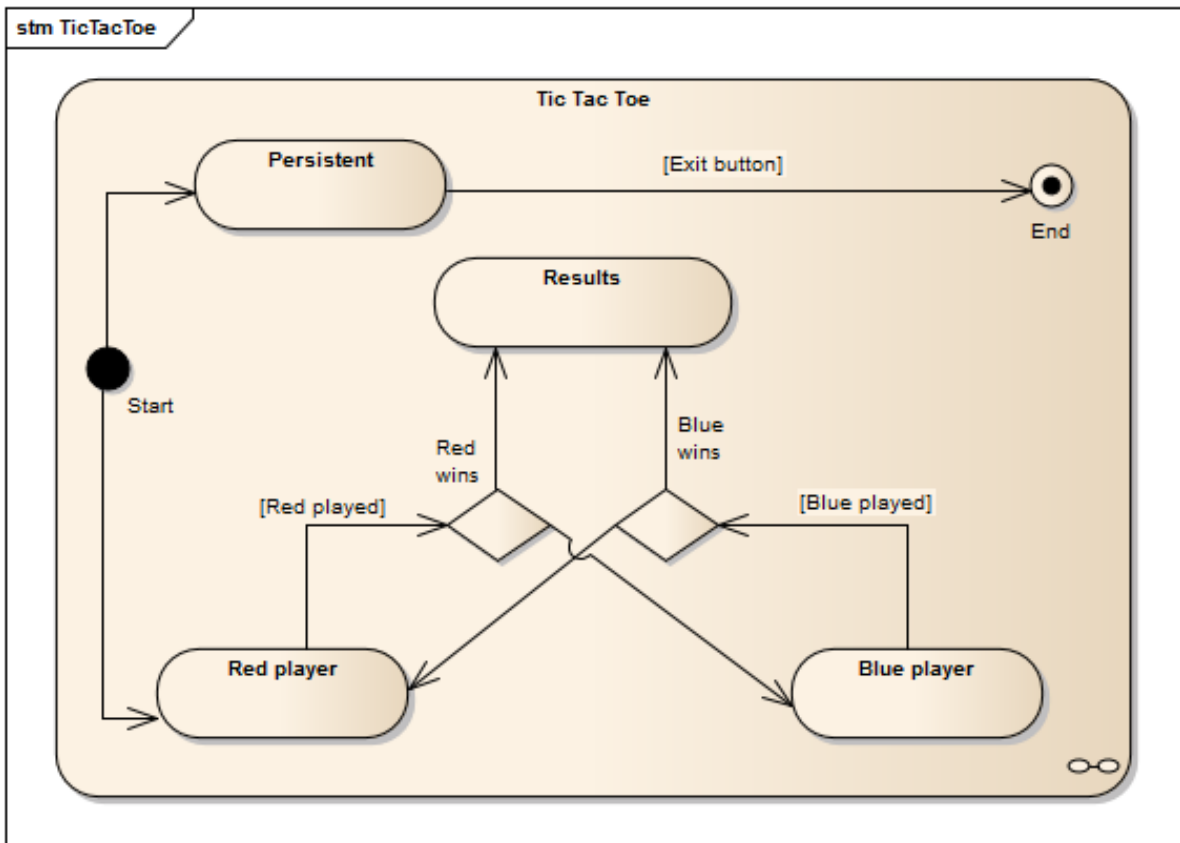
Rule of the game are fairly simple. Players are taking turns on putting their mark on unused fields in three by three array, and first player that has three of his marks next to each other vertically, horizontally or diagonally wins. In case all fields are marked and no player is a winner, game results in a draw.

### ■ 21.2 State machine

Logic of the game could be presented in many ways depending on how much information we want to represent. Example how state machine for *TTT* could look like is in Diagram 8. As you can see, there are total four states, *persistent*, *red player*, *blue player* and *results*. We could go a bit further and split *results* state into three separate states *red won*, *blue won* and *draw*, but that would be waste of space and it would make the diagram harder to read. The point here is, that these states can not only be split in diagram, but as well in the game logic. We can define state *results* that will behave depending on winner marked in registry, or we can have different state for each possible winning condition that will have static behaviour.

Merging states is exactly what I did while implementing the game. In actual game, there are only two states. First, *main* state that does loading and exit button behaviour, and second, *game*, that does everything else. That may sound like it is doing a lot, but it actually contains only two rulesets. Setup ruleset that initializes visibility in game scene and cleans registry records, and active ruleset that reacts to mouse click on tile, will mark tile as one belonging to current player, swap players, check winning conditions and possibly displays winner.

In this case has *game* state "only" about 100 lines of definitions which may sound like a lot for such simple thing, but keep in mind how many things user does not have to take care of. For beginner would be probably better to split the game in as many states as possible, but in the end, it has its benefits to concentrate it back into minimum of

Figure 8: State machine for *TTT*

states, once everything is up and running.

### ■ 21.3 Scene

For this game are in scene needed four main components. First is exit button, that will allow us to turn off the game in full-screen mode. Second is some sort of marker telling to players who is now playing. Third would be grid three by three for game tiles and finally text message saying who won the game.

**Exit button** Since Text component inherits from Panel, it receives mouse click events. This way we can just define Text component in scene, position it, attach style ID to it and button is set up. It is not very intuitive to call Text component as button, but after all, everything that extends Panel can be button, so why restrict ourselves.

**Current player** Marker determining current player can be simply a panel with extra interaction states attached, which we can then style for red and blue player. In that case, when in logic is changed state of that Panel to one or the other player, its color changes as well.

**Grid** Grid is not at the moment implemented in the engine, but we have some option how to help ourselves while creating one. In implementation of grid this small was simpler option just to create vertical layout containing three horizontal layouts where cells have dimensions *30%* of parent. These cells have defined same interaction states as current player marker, they will be just set only once in the logic.

**Winner** Label announcing winner is just another positioned Text component by default visible — it will appear after game ends. Text of this component does not matter since it will be filled from logic.

Resulting scene can be visible in Figure 9.

## 21.4 Registry

Because of state aggregation, we will use registry a bit more than in case of expanded state space. We need to remember current player and tiles with owner values they contain at the moment. Tiles are at runtime stored in linearized 2D array under key *game.tiles* so it can be passed in appropriate rule for evaluation of winning conditions.

```
<map key="game">
  <map key="tiles"/>
  <map key="player">
    <map key="current"/>
    <map key="red" value="blue"/>
    <map key="blue" value="red"/>
  </map>
  <map key="mainScene" value="./data/tictac/scene.xml"/>
  <map key="mainStyle" value="./data/tictac/style.sts"/>
  <map key="mainState" value="./data/tictac/states.xml"/>
</map>
```

Switching of players is with use of helper values done using single action:

```
<action class="StoreData">
  <param name="data" access="reg" value="game.player.%"/>
  <param name="data:wildcard" access="reg" value="game.player.current"/>
  <param name="location" access="reg" value="game.player.current"/>
</action>
```

Value for current player is string containing his color name because in that way can be winner announced by pasting player ID in predefined message.

## 21.5 Walkthrough

When game is started and scene<sup>33</sup> loaded, we can immediately see red player is currently playing, as shown in Figure 9. Moving cursors over tiles highlights them as available for first move.

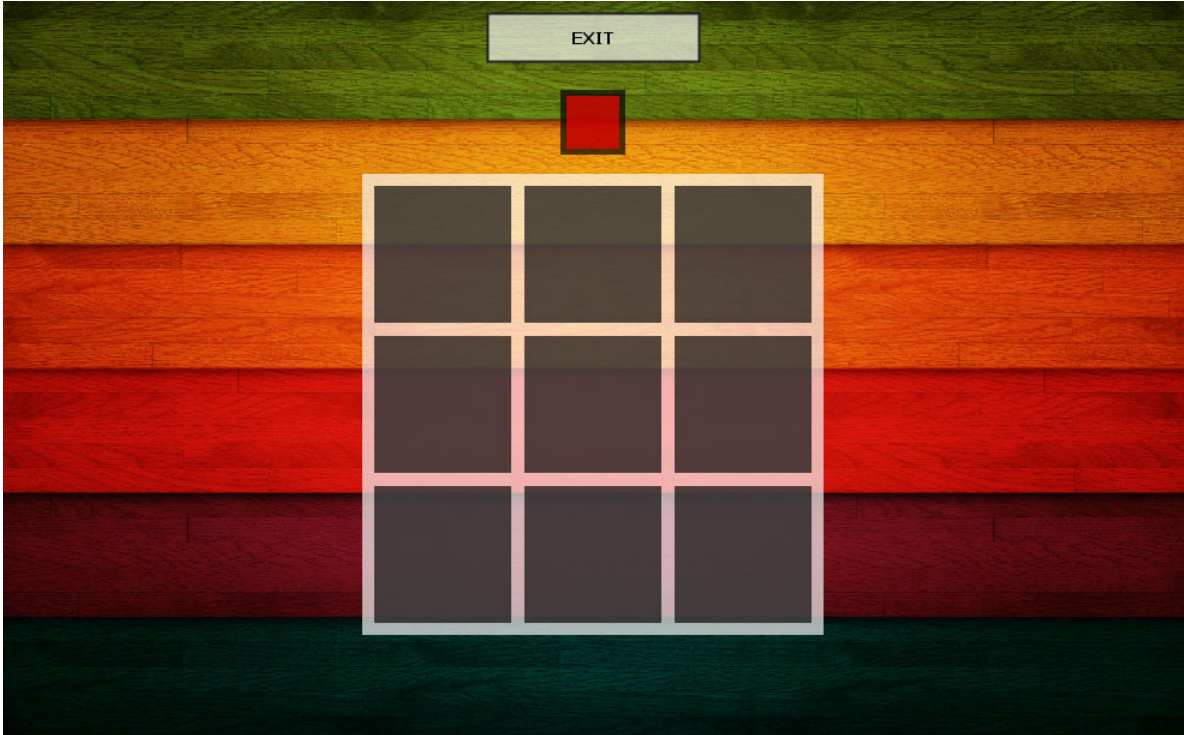


Figure 9: *TTT* — New game

In Figure 10 is visible scene after clicking on a tile. Clicked tile turns red as current player's state was set to it, and it is also disabled, so it cannot be played again. Current player marker changed its color indicating now that it is blue's turn.

Now after several turns, as shown in Figure 11, red player won and it is written in a message below the board. All tiles are also disabled so they stay in their current state and the game is basically frozen, leaving only one option — the exit button. But because the state space has *debug* state enabled, we can hit *F5* to restart the game. That means the whole scene is cleared and the setup stage of all active states is called again. Because state setup is constructed correctly, it puts the game in a consistent state and it can be played again.

<sup>33</sup>Background image used for this game is from page <http://www.iwallhd.com/wallpaper/1600x1000/wallpapers-texture-wood-simple-hd-free-in-for.html> under GNU GPL

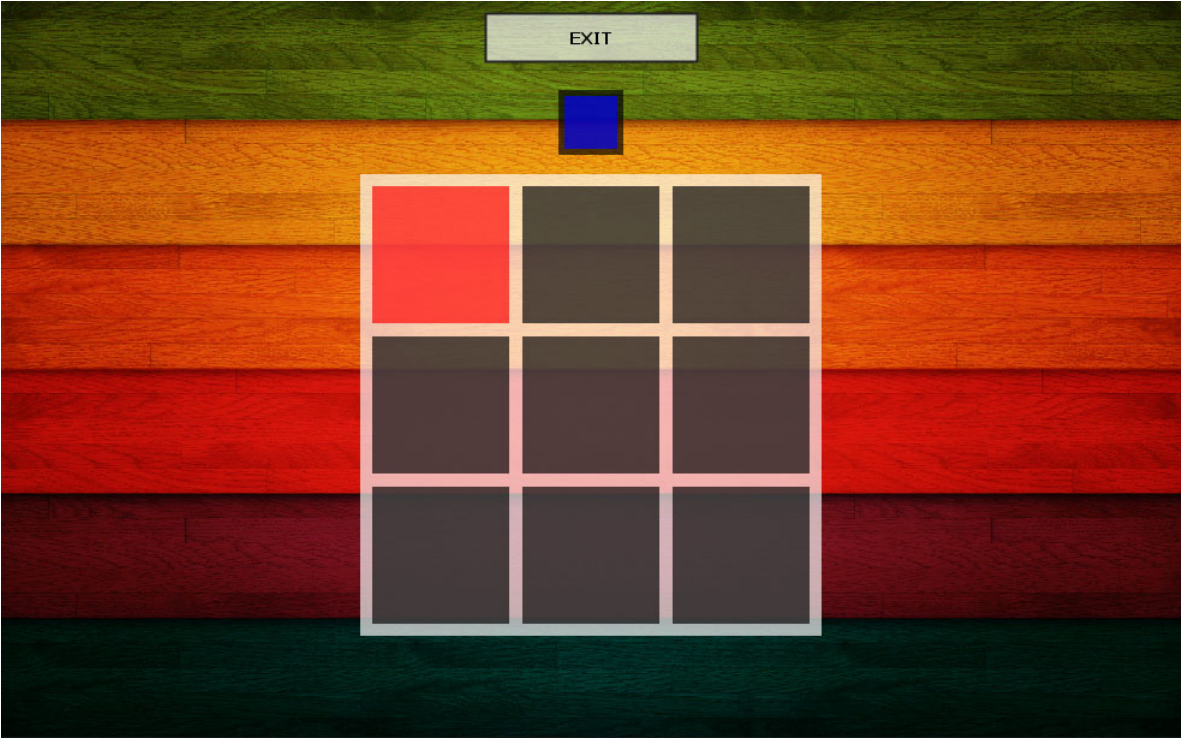


Figure 10: *TTT* — First move

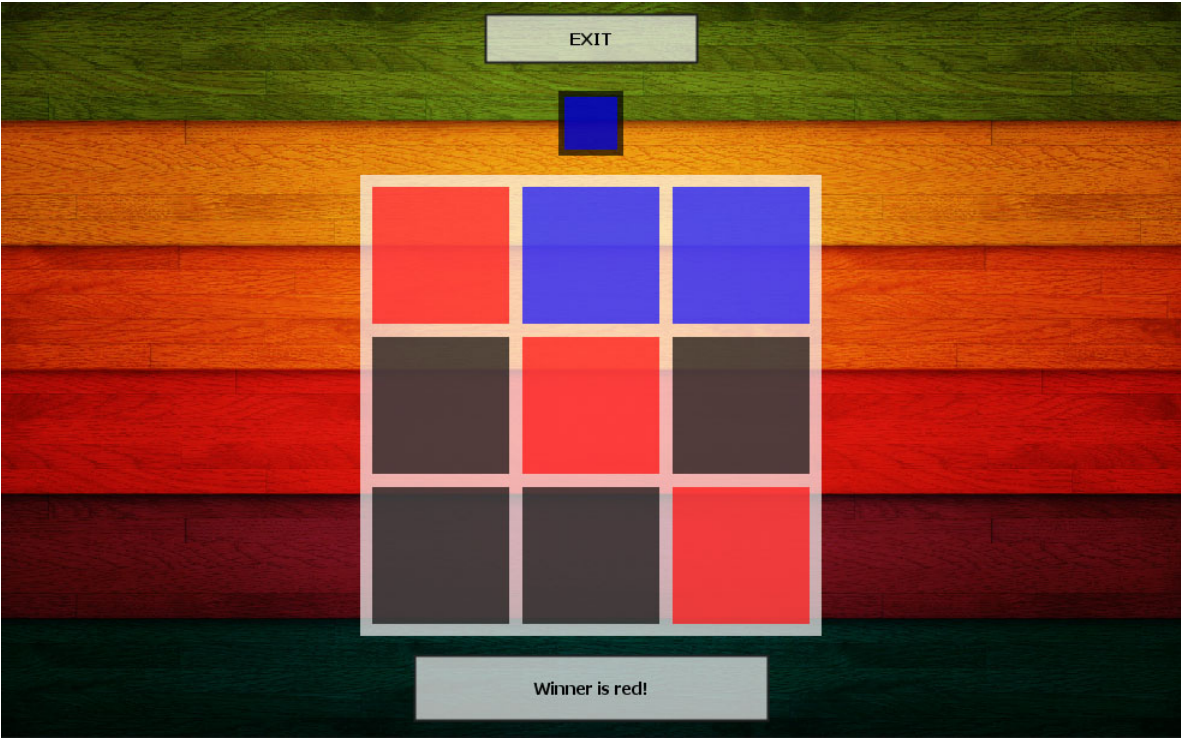


Figure 11: *TTT* — End of the game

## ■ 22 Cards against humanity

For second example is chosen simple open license game called *Cards Against Humanity*<sup>34</sup> (further referred as *CAH*). This game is based on matching sentences containing blank spots with funniest fillings. Point is to reveal dirty-mind-ness of players. I picked this game because of lack of licensed imagery and simplicity of rules that allows a lot of space for modifications.

### ■ 22.1 Rules

This game in general can be basically for any amount of players larger than two, but because of time demands added per player is set maximum of players to five. During a round can have these players one of two roles, either *czar* or regular player (further referred only as *player*). Game round consists of two main stages, *player* picking answer from their hand and *czar* picking the funniest answer from what *players* played. Owner of picked card gets all *points* in the play, always at least one, and at the end of the game, player with most *points* wins.

**Stage 1** New question card is drawn out of the deck of all unused questions and presented to all players. This card is visible for everyone for rest of the round. This question card may contain either question, or fill in the blank sentence.

**Stage 2** All *players* draw answer cards up to 10 and pick the answer that is the funniest in their opinion. They remember their own card so they can score points for it if it is picked by *czar*. These cards are put in one pile. In case a *player* has more cards he believes might win, he can spend one *point*, if he has any, and play one extra card to boost chances to be picked. Winning player gets all gambled points.

**Stage 3** *Czar* takes pile of played answers, shuffles it, presents this set of cards to all other *players* and picks the funniest answer there is. Picked *player* gets all points currently in play and *czar* role is shifted to next player.

Original rules can be found on attached disc.

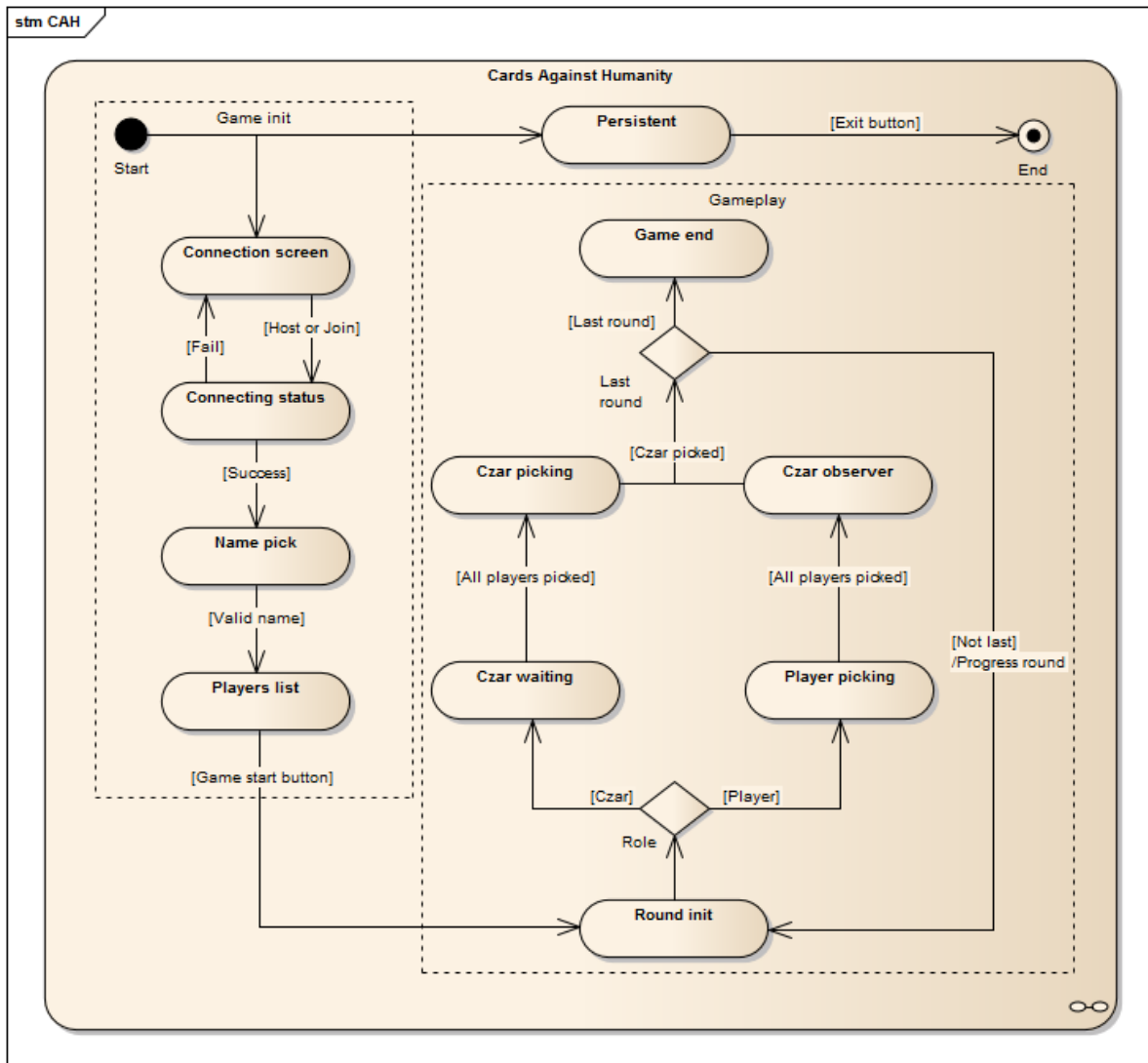
### ■ 22.2 State machine

Logic of *CAH* game implementation is mostly done according to state machine in Diagram 12. Similarly to *TTT*, *CAH* has persistent state for exit from the game and set of dynamic states that shift along the course of play. These dynamic states can be divided in two sections, initialization and gameplay.

**Initialization** In initialization phase is just established connection to a host, player

---

<sup>34</sup>[www.cardsagainsthumanity.com](http://www.cardsagainsthumanity.com)

Figure 12: State machine for *CAH*

picks name that is unique amongst other players and then is everyone just waiting for host to start the game. Host can start the game only if there are no players currently picking name and if minimum number of players joined the game. By hitting the start button will host initiate construction of remaining support structures, such as hands and decks. At last are *clients* notified that they should shift to next phase.

**Gameplay** Once all *clients* enter gameplay phase, they are cycling states for stage one, two and three. Along with *client* is there on *server* counterpart dealing with network messages related to that stage. This repeats for certain number of rounds, where round is one change of *czar*. This number can be defined in registry and could be subjected to time preferences of playing group.

See documented implementation for further details.



## ■ 22.3 Scene

Scene of *CAH* uses similar constructions as *TTT*, but some are worth mentioning. As first construct notice extensive use of visibility changes in the scene. It is a way to avoid expensive operations of inserting new nodes in scene and making game more responsive. That is possible because nodes do not interact with each other unless they are in Layout component, so they can be overlapping each other in the graph. The other is display of cards on limited amount of space. Every card has its own layer it is drawn on. This layer is ascending from left to right, and when a card is hovered, its layer changes to value above all others.

## ■ 22.4 Registry

Amount of data stored in registry here is significantly higher compared to *TTT*, but that is to be expected. Worth mentioning is concept of language definition and card decks.

**Language definition** In registry is key *lang* that contains under other keys specific texts used inside of the game. These are loaded into appropriate text components on state entry. Purpose of this definition is to have language dependent content separate where it can be easily replaced. Paths to different translation registry records may be defined in registry main registry record and then used to dynamically merge and replace all the texts inside of the application using without changing game logic.

**Card decks** With *CardDeckAction* was added possibility to create list of indexes that point into different array in registry and that way operate deck of cards without even touching original definition. Created deck can have the indexes simply removed to prevent cards from being reused and shuffled to ensure their random order.

## ■ 22.5 Walkthrough

On game start will player see connection Screen 13 with option to join existing game, host new game or exit the game altogether. Exit option is available for course of whole play. As the prompt says, if player wants to join existing, he should write host IP address to input below. Since there is only thing player can write into, focus is automatic. In this input field are allowed only numbers and dots, and contained text cannot be longer than 15 characters, which is longest *IPv4* address possible. On each change is input evaluated whether it is valid IP address and if so, *JOIN* button is enabled. Second option on the screen is enabled only if game succeeded in finding some network interface that is up and connected to some network. If it is that case, found IP address is displayed so less experienced players can share it through other means with their playmates.

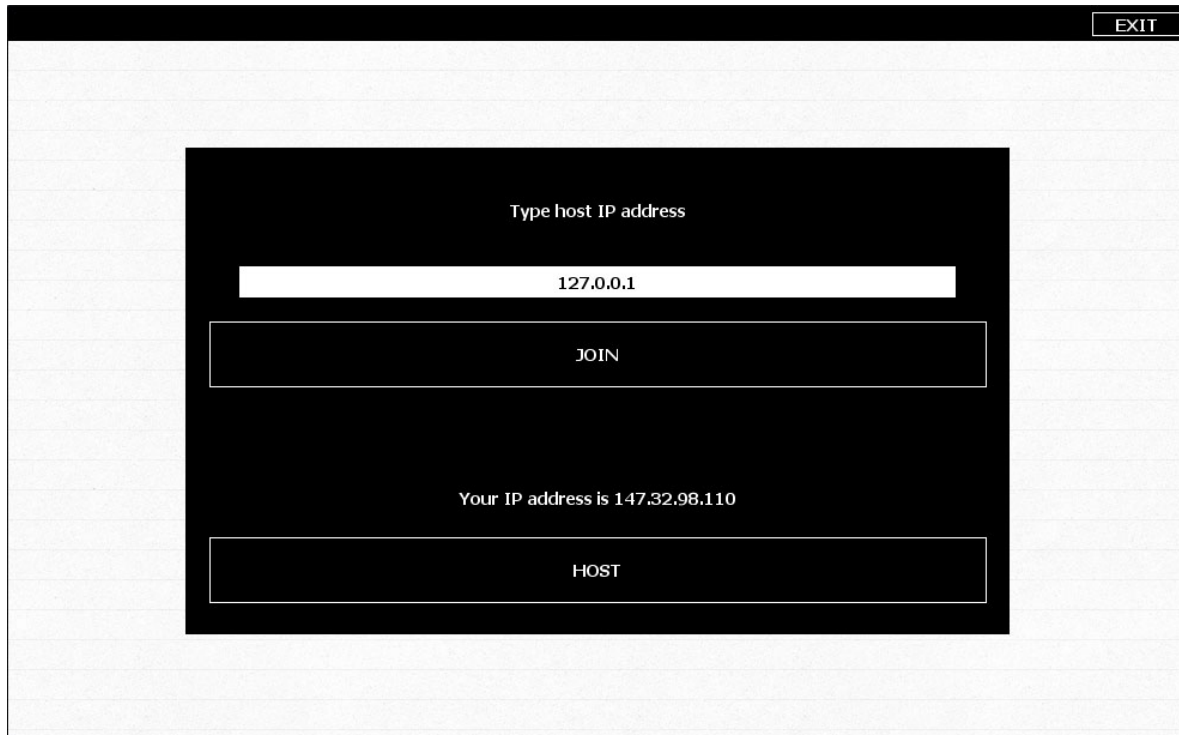


Figure 13: CAH — Connection screen

Once player chooses one of the connection options, new Screen 14 appears with text *Connecting...* indicating network controller is attempting to start connection or server on currently set preferred ports. In case of incorrect host may this screen stay on for several seconds as timeout for connection attempt is one second. In case of failed connection is player redirected back to connection screen with error message reporting something bad happened. On the other hand, if is connection successful, player is sent to Screen 15 for picking of name.

Name picking screen is here for simple management of names. Since names have to be out of principle unique within one play, they need to be first verified on server. On this screen can player type name consisting of letters and numbers. When name is non-empty, *CONFIRM* button is enabled, and once pressed a validation message is sent to server that will verify sent name and allow player to proceed to player list Screen 17. If is name taken, error is displayed instead and player stays on the same screen.

When host of the game is waiting in players list Screen 16, he has extra overview on top what regular client sees. There is extra *START* button which is enabled 18 when no player is picking name and when number of connected players is larger than two. Until player picks the name, there is nothing to display, so instead there is *Picking* gray text that serves as a notifier for host that he is waiting for something in particular and that game is not broken.

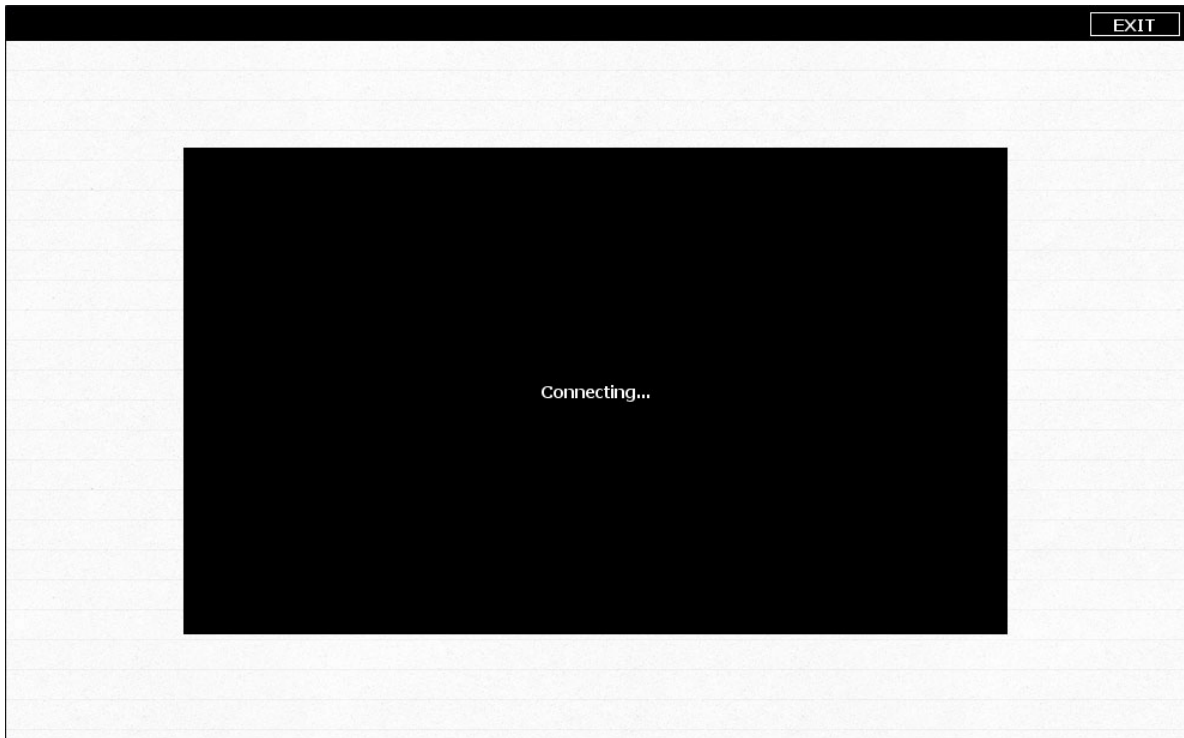


Figure 14: *CAH* — Connection status

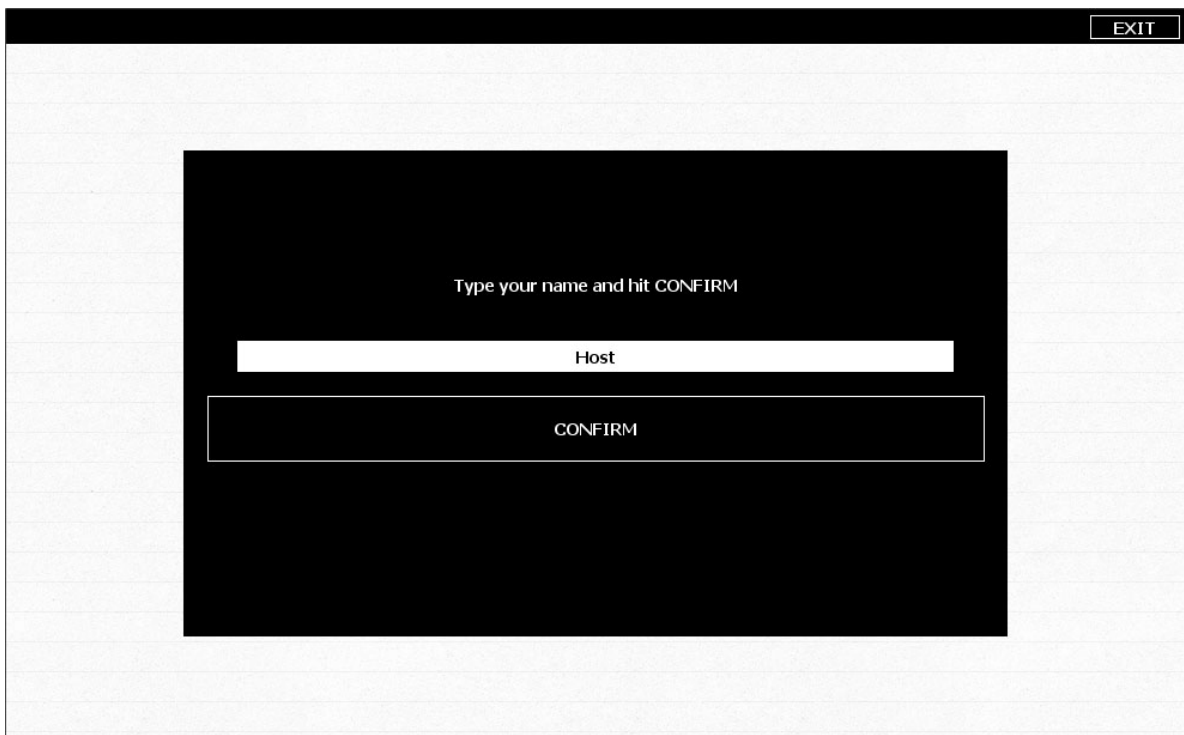


Figure 15: *CAH* — Player picking name

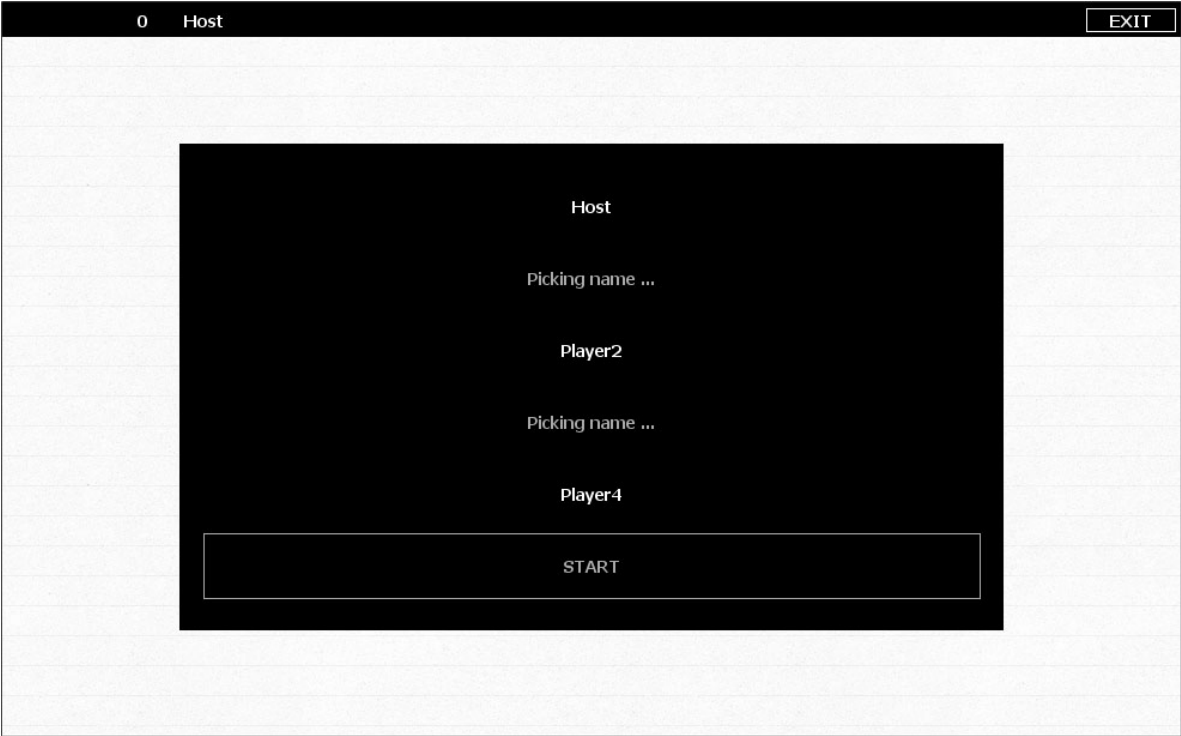


Figure 16: CAH — Host waiting for players to connect

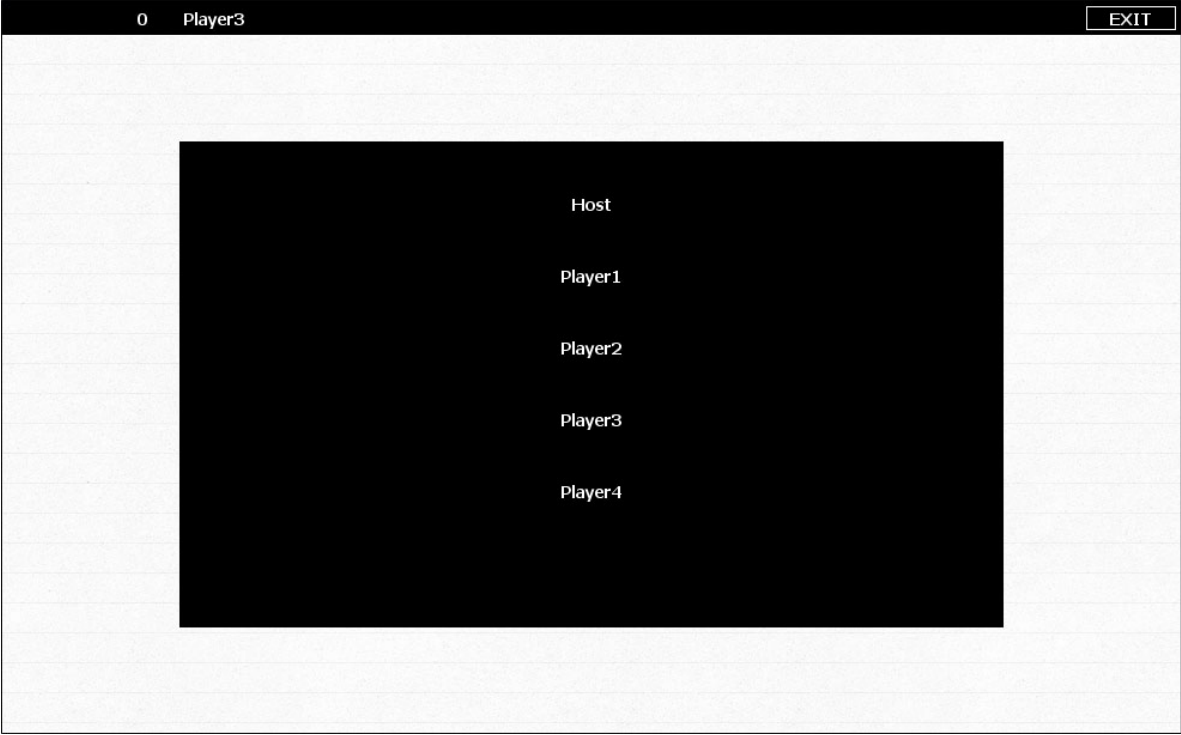


Figure 17: CAH — Client being ready for game to start

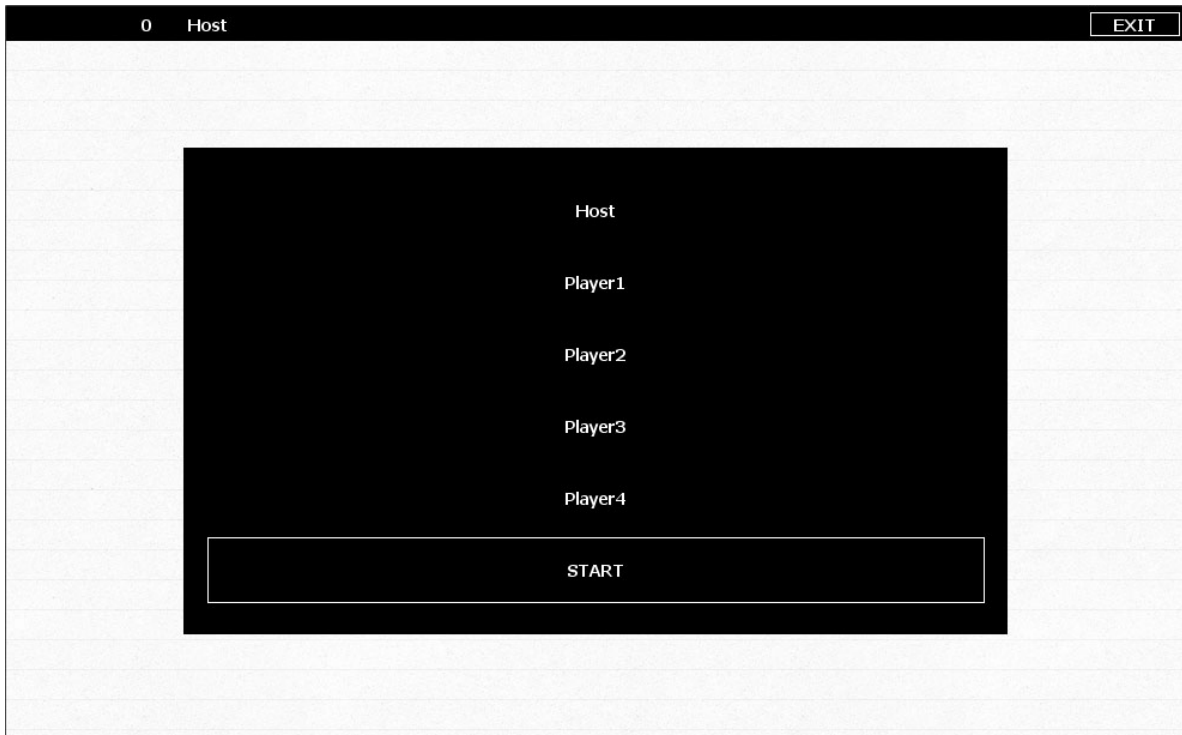


Figure 18: *CAH* — Host being ready for game to start

Once game starts, *czar* is directed to his waiting Screen 21 and all other players to Screen 19 for picking an answer card. On entry to picking state is sent card request message from every empty slot in *players* hand and server returns new card from its local deck. Returned value is ID to card collections that are in everyone's registry. For the time *player* did not select anything is displayed a hint that a card should be picked. Once it is picked 20, new hint advises *player* to confirm that choice by hitting ready button. After hitting ready is *player* waiting for others to play and when they do so, transition to *czar* observer Screen 23 is made.

When all *players* finish their picking, all picked cards are shuffled and presented to everyone. Only *czar* can in this stage select cards 22, but selection that was made is visible 23 for every *player* so everyone sees WHAT exactly was the funniest thing. When *czar* hits ready after picking a card, scores for *players* are recalculated, distributed and game is progressed by picking new *czar* and question card.

When player has some points, option to hit button *GAMBLE* instead of *READY* is available 24. This will result in card being played without ending *players* turn, so one more card can be picked. Score is subtracted immediately so *player* can see the impact of the action when it is done 25. This has of course impact on overall number of cards when *czar* is picking. That means in *czar* picking screen can be with maximum amount of players and everyone gambling eight cards.

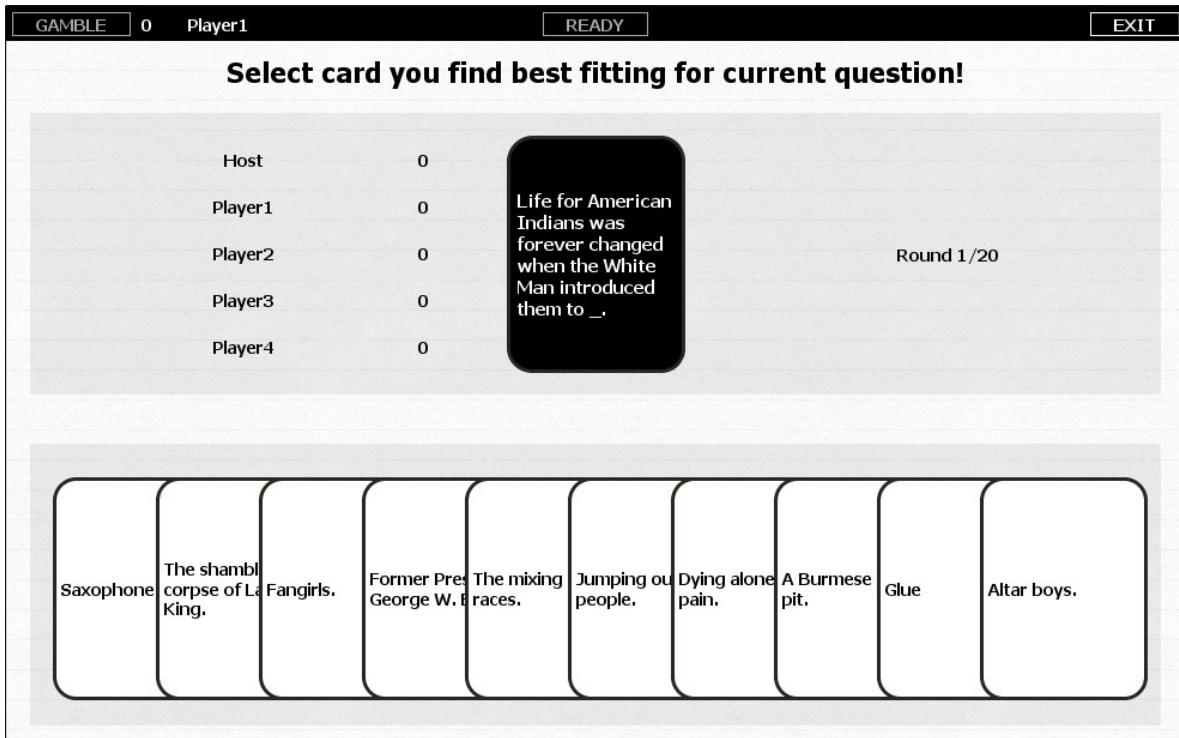


Figure 19: CAH — Player picking an answer card

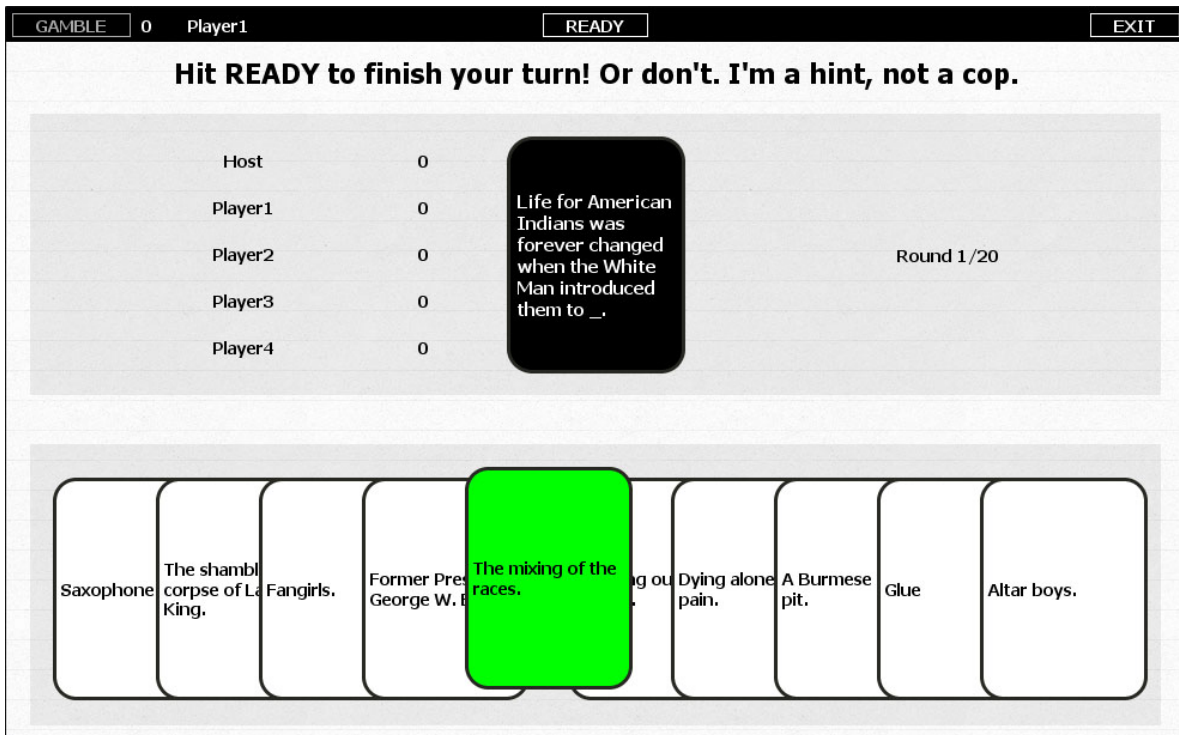


Figure 20: CAH — Player picked an answer card

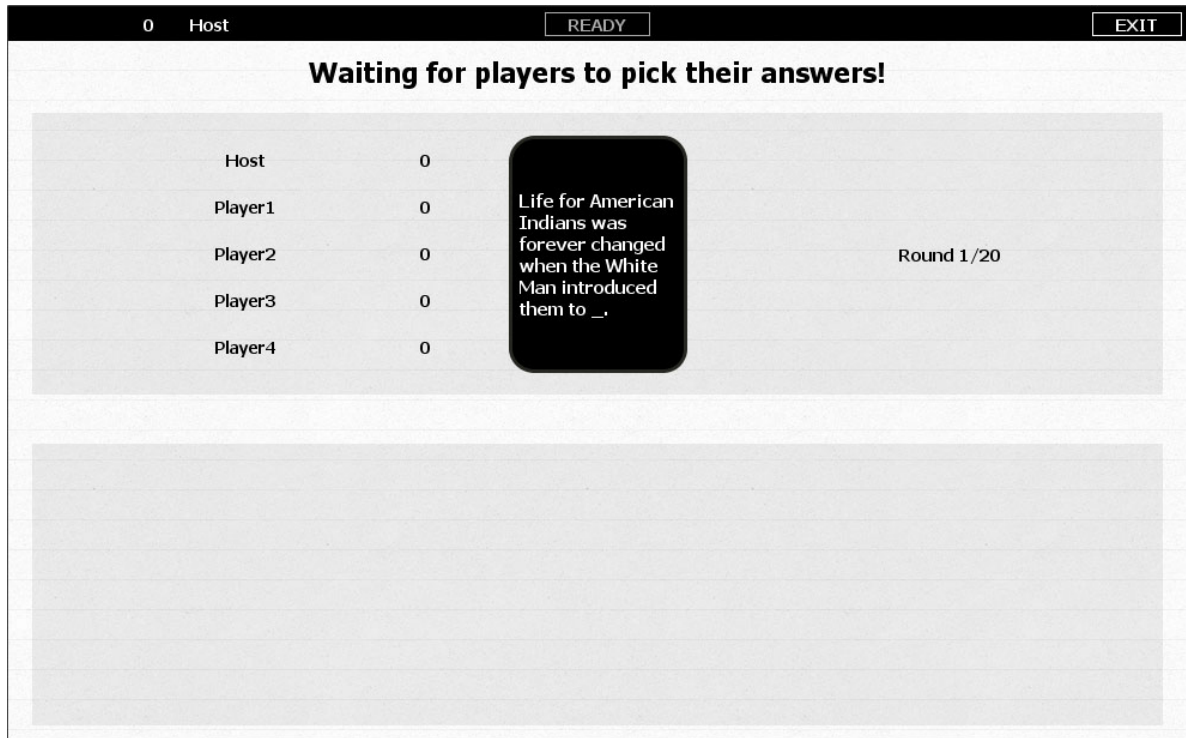


Figure 21: CAH — Czar waiting

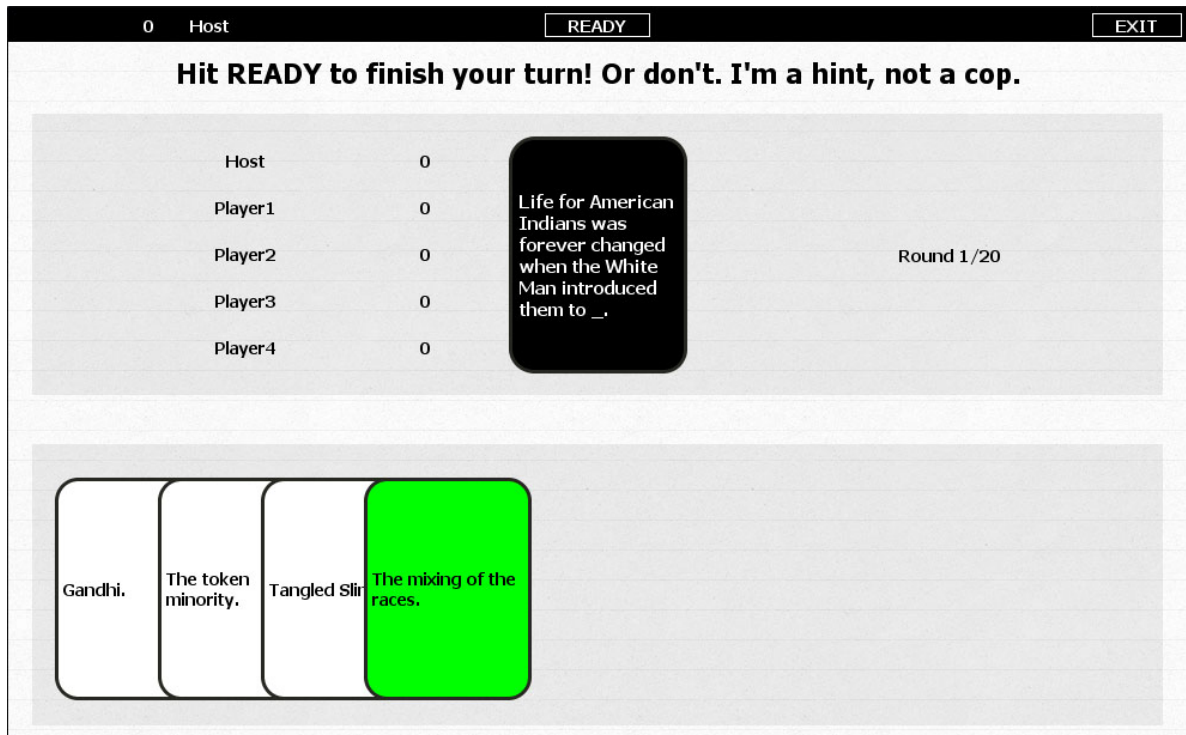


Figure 22: CAH — Czar picked winner

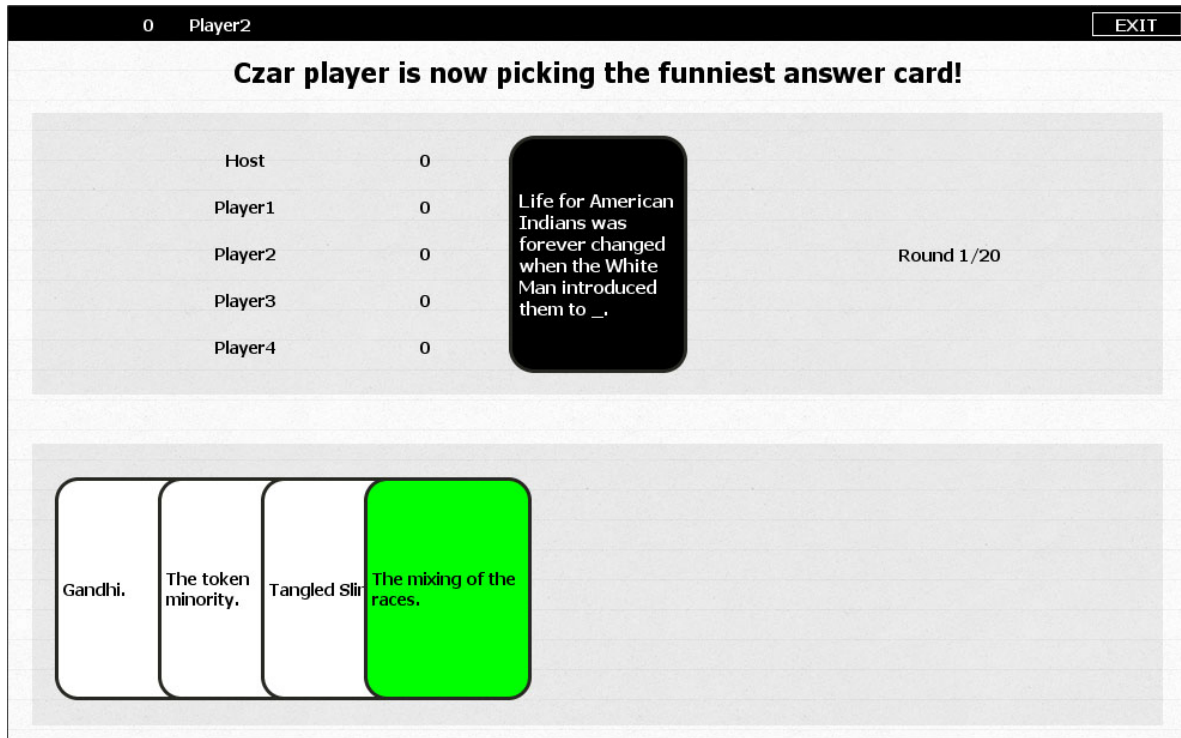


Figure 23: CAH — Player observed what czar picked

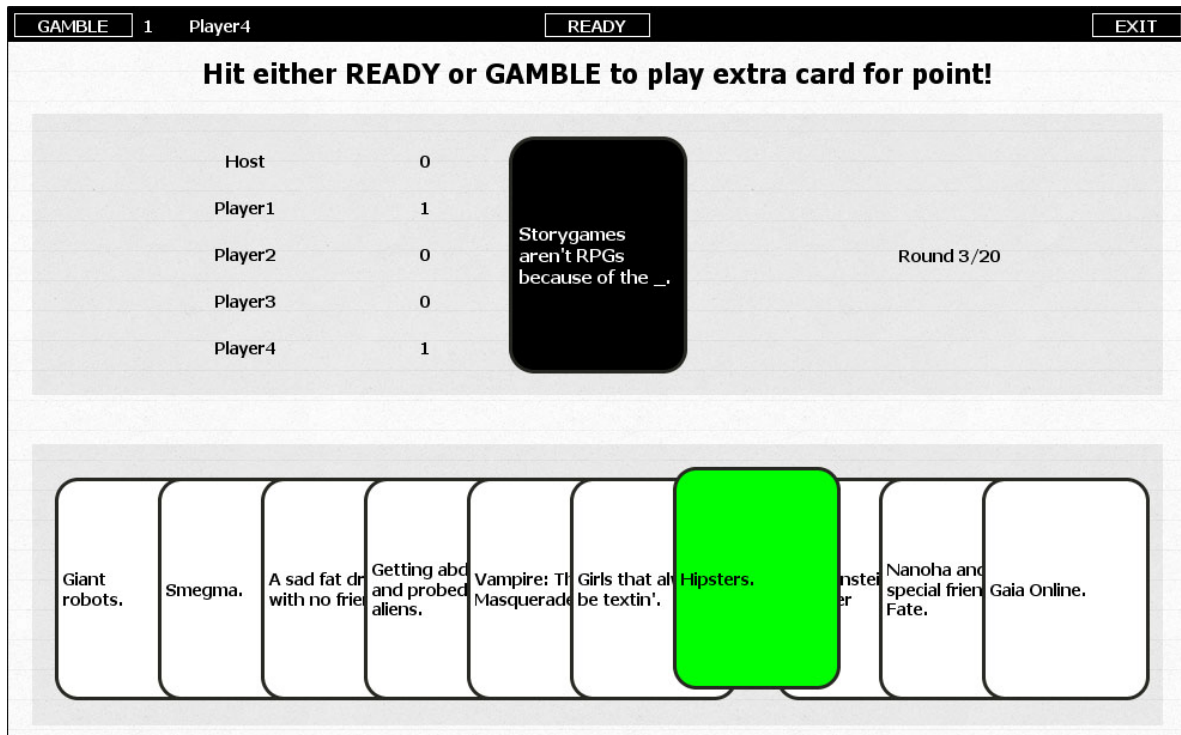


Figure 24: CAH — Player has gambling enabled



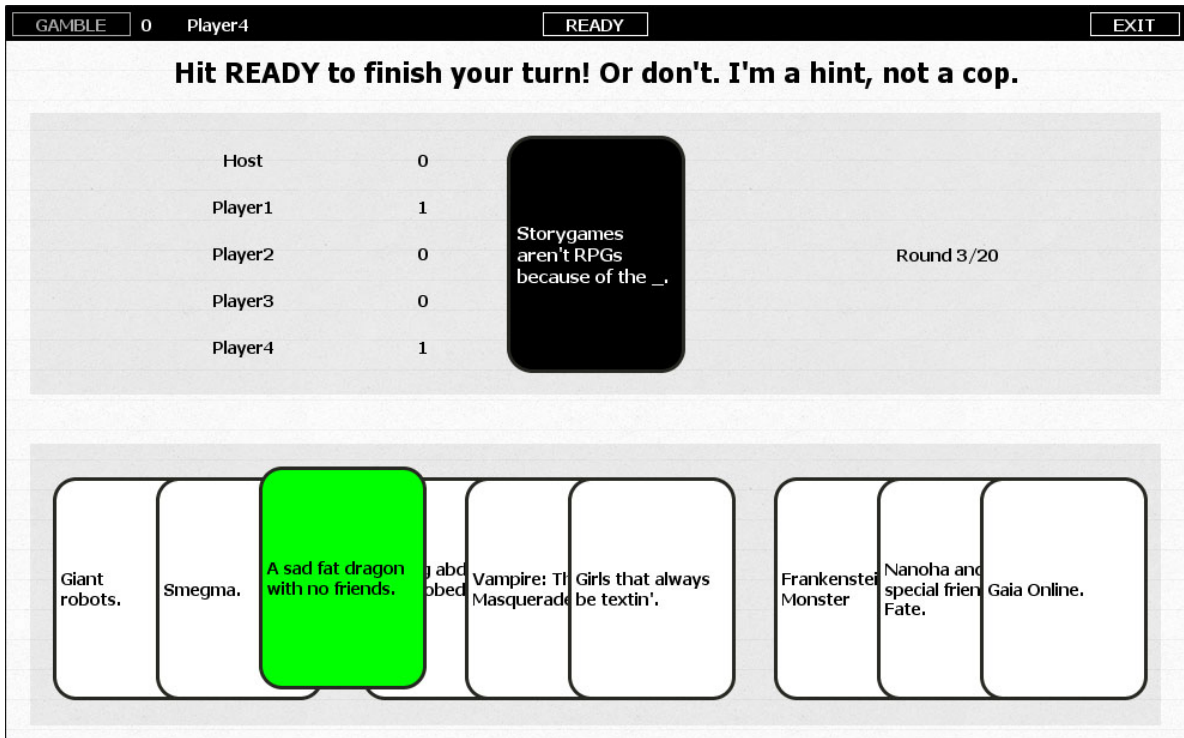


Figure 25: CAH — Player gambled extra card

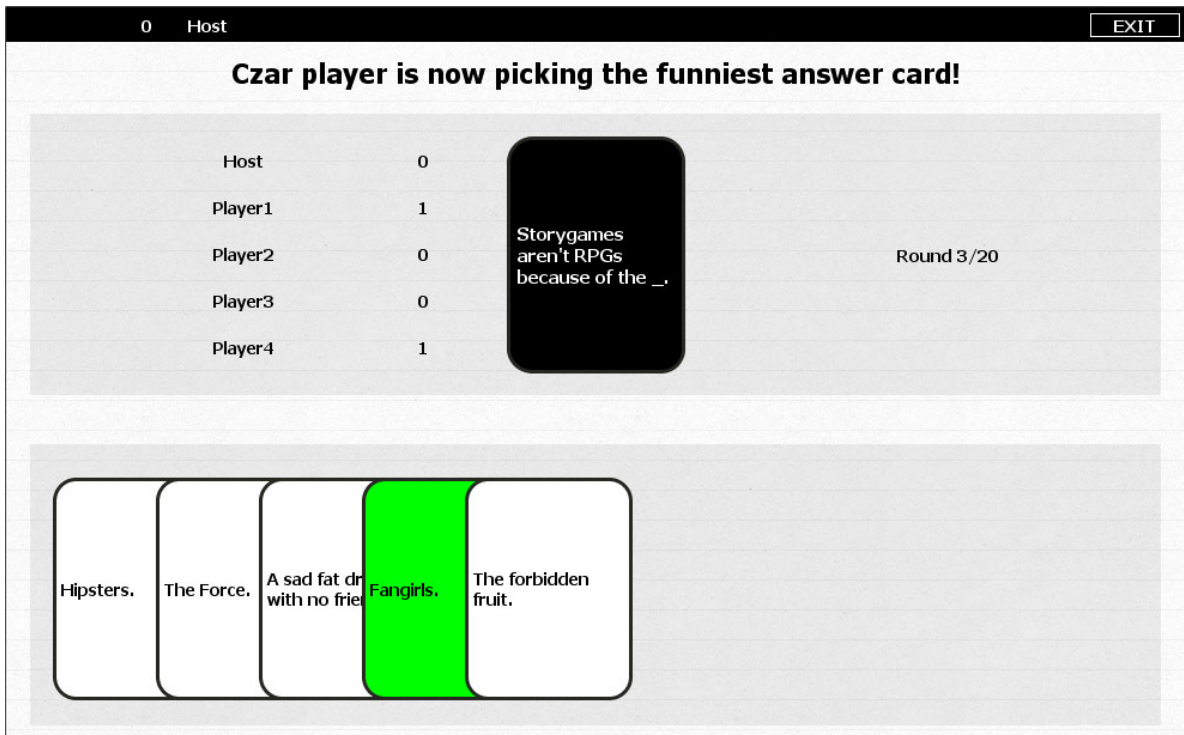


Figure 26: CAH — Increased number of cards for czar

When round counter reaches its maximum, instead of transition to next round screen with score and without cards is displayed.

### 23 Discussion

This part serves as extensive guide through implemented engine. Most implementation properties of basic building blocks mentioned in Part 3 were described to a degree allowing reader to fully understand architecture and use of the engine. This part is then complemented by contents of relevant sections from appendix.

Part of implementation process during creation of this engine was building of working example games. For this reason were briefly introduced two examples of games implemented using this engine with complete walkthrough of screens that appear in those games.

## Part 5

# Usability tests

In this part testing of created system and game implemented in it is discussed.

## 24 Testing in general

Testing of the system and created games is vital for spotting of major flaws in design. This procedure should be done several times during development process of the system. Different iterations of testing have different content based on what we already have and how in depth we want to test at that point. Early stages of development are covered with experts which evaluate whether design fulfills basic recommendations. In later stages of testing it is necessary to involve real users of the system since experts work around assumptions, and those may significantly differ from reality. Testing with users should resemble the way final application will be actually used as much as possible.

## 25 Testing with programmers

The system needs to be tested with programmer to find out whether all processes in it are intuitive. In this section, there is described a test with programmers where they were asked to create *Tic-Tac-Toe* game.

### 25.1 Method

As a testing method, I chose a method somewhat similar to laboratory testing of UI designs. These are done with real life users of the system. They should point to problems that are either utterly hidden to creators of tested environment, or assumed problems that need to be verified before major, and often expensive, changes are made.

This testing method usually requires at least two people that are familiar with the tested environment, moderator and logger. While moderator is with participant and directs progress through tasks, logger observing the test makes notes about everything what is happening. When logger does not know the tested environment, he is not able to spot potential problems and must log everything. In this case can be logger replaced by recording of the whole session and evaluated later. In case moderator is not familiar with tested system, he is not able to help participant in case of severe blocks and testing cannot proceed without help from outside. Since I am the only person developing the system, there are just two options available. First, using recording device and evaluate the session after it ends, or second, do logging while moderating. Since the nature of

this test is an early stage one, I did not consider the precise logging necessary, so all results from testing process are notes from moderator.

While testing UI design, test participants are presented with a set of tasks they are to achieve and there is usually one or two ways how to reach these tasks. Testers then observe participant trying to achieve set task and note where were problems and what might have caused them. Participant can get stuck somewhere along the way, and that points us to critical issues with the design.

Since processes being tested are much longer and complicated by themselves, at least in comparison to UI design testing, each task has allocated more time and it is expected that participant will consult documentation of system. That way not only system itself is tested, but its documentation as well.

For the duration of the test participant is asked to follow *think aloud protocol*, which basically means participant should vocalize all though processes related to current task. This helps moderator or logger to spot more problems than by silent observation. Participants tend to forget about this, so it is moderators responsibility to remind them when it happens.

At the beginning is every participant presented with simple *Hello World* template of application that is supposed to ease the beginning. This template should not disrupt the test results, since such template is intended to be available along with the engine, so using it follows intended use case.

### ■ 25.2 Screening

During screening process is decided for what kind of user is the tested system intended. There may be multiple different types of users and all should to be selected with respect to their proportional representation in target demography.

**Java** First evaluated characteristic of participant is experience with *Java* programming language and participants general knowledge of it. This requirement roots in necessity of creating of custom *Actions* and *Rules* for complicated operations and user completely oblivious to java programming would have difficulties while using the engine that would not be directly related to engine itself.

**Web technologies** Second observed characteristic is participants knowledge of web development technologies, CSS and HTML. It should not be necessary for user of the engine to have experience with these, but since parts of it are strongly inspired by them, user being to some extent should be performing better at all graphics related tasks.

**General programming** The last characteristic is general programming level of participant. In this case, it does not depend on the language, but it is really important to have some basic analytical thinking and understanding how programs work. This requirement is related to logic programming, where user is constructing basically sequence of functional calls with conditional execution, and understanding that is necessary for any user of the system. This engine is not intended for non-programmers.

In the final group of participants was a representation of all observed types. Total number of participants was only three even though recommended number is four to six. This is because the number of discovered problems with each next participants follows logarithmic curve and this engine is basically still in early stages of development, so we care only about the most critical issues.

### ■ 25.2.1 Participant 1

This participant represents user with average experiences from general programming. From web development has more experience than others and is able to do all parts of creative process. This participant is in further text referred as *P1*.

**Java** three years of use of java in school, several android projects

**Web** several websites

**Programming** participant has experience with java, php, C++ and C#

### ■ 25.2.2 Participant 2

Participant number two is experienced programmer that worked a lot in java but has no background for web development. As basic requirements fulfills knowledge of XML file structure. This participant is in further text referred as *P2*.

**Java** five years of professional java programming

**Web** no experiences with web technologies and closest to HTML is with XML configurations

**Programming** participant has experience with java, matlab and C++

### ■ 25.2.3 Participant 3

This participant represents user with reduced programming skills, medium web development experiences and not completely oblivious to java but very close to it. This participant is in further text referred as *P3*.

**Java** little experience, is familiar with he language but did not use it too extensively

**Web** maintenance of several websites, no experience from creative process

**Programming** participant has low programming experience, more of system administration, is familiar with bash, C++, python and java.

### ■ 25.3 Tasks

All tasks represent together creative process of making simple game. Tasks made by participants are split in following section.

**Environment setup** First, a working template needs to be set up. This part is here because all participants are testing the system on their own computers, as real life use environment would look like, and setting up working template may not be all that obvious as it might seem. In this phase, user is allowed to use the Internet for solving of platform specific issues and moderator is allowed to cooperate after fair number of tries.

**Analysis** Second, participant needs to do analysis of the problem in terms of what are the requirements, what are basic building blocks of the program, etc. This part is not evaluated and is directed by the moderator. It serves for setting the participant on the right track as quickly as possible. In practice, this part of game development would be done outside of the engine anyways, so it is better to speed up the testing process. It is important for the moderator not to influence the participant during this phase with, for example, advices about design that would make easier development in tested engine. Most of this moderation should be done by asking using open questions instead of giving advices.

**Scene setup** Third, scene needs to be created and tested before logic can be applied to it. It is in separate phase, because once all scene components are in place, there is not too many changes once logic part is entered and mixing these two would only support confusion of participant while trying to figure out what is the problem. Moderator is in this phase not allowed to give any advice until participant gives his fair try to fulfilling the task alone.

**Logic construction** The fourth phase is for participant to start adding behaviour to created scene using designed state machine from second phase. Moderation in this phase should be very similar to the one in scene setup.

**Cleanup** The last phase serves as final touches and optimization of game that is already working but could work better with minor adjustments. This phase is not determined by specific tasks, but it tests participants satisfaction with created game and what is the thought process of improvements.

All tasks are described in following list with additional commentary that was not in list presented to participants.

### Environment setup

1. Install program *NetBeans 8.0.2* with *Java Development Kit (JDK) 8u?*. This part would not be normally required, since the system is not dependent on NetBeans, but *XSD* definition builds are already defined in NetBeans project and it would have to be defined by hand in any different IDE. I assume that the fact user has some different favorite IDE implies knowledge about it sufficient to set up everything anew.
2. Install *Java Runtime Environment (JRE) 8u?*. This step is required for running created game outside of NetBeans IDE. The game can be developed in the engine without IDE altogether, but application must be run in console so so creator can see possible log messages.
3. View all template files in data folder. This step is intended for participant to get familiar with structure of basic game so he know where to look when these parts are mentioned.
4. Run the game with unchanged *Hello World* template files. In this step, participant will verify that everything is working correctly.
5. Go through all debug options in *debug* state and make sure you understand them properly.

### Analysis

6. Construct on paper state diagram of hot seat *Tic-Tac-Toe* game. In this step is participant forced to realize what should inner working of the state machine look like, but as well what components will have to be used.

### Scene setup

7. Insert in the background of the scene *Panel* component with background image that will always cover whole scene but will maintain aspect ratio of the image. In this operation, participant should look up *aspect-ratio*, *min-width*, *min-height* properties along with obvious *background-image* and positioning properties.
8. Split the scene horizontally in ratio *10/90* using two containers. These components should not be placed in background *Panel*. This task is supposed to remind participant that components may overlap in the scene without influencing each other.
9. Place *Text* component in upper *Container*, set its width to *20%* of its parent and center it horizontally. In this task should user learn how to do relative positioning to parent.

10. Assign class "*button*" to this *Text* component and style it in the way that it has some default background color, then different color on mouse hover and mouse click. Participant should demonstrate how intuitive it is to define styles for interaction states of components.
11. Place new *Panel* in lower *Container* that will be centered, have maximum dimensions on 70% and will stay square. This task is here to ensure participant understood how property *aspect-ratio* works.
12. Construct in new *Panel* a grid three by three of *Panels* positioned using *Layout* components. Each tile *Panel* is 30% square with defined background color. Participant should be able to tackle layouts within each other and be able to write more complicated selectors than direct IDs.
13. Create two custom states *red* and *blue*, assign them to each tile panel and style them to match color of their name. Here is demonstrated the creation of custom interaction state. At the end of the step should be participant asked whether result of his actions is correct and explain the answer. That is because interaction states are set from within the logic and at this moment is participants work not visible.

### Logic construction

14. When player clicks on tile from created grid, change its interaction state to *red* or *blue*. With change of this interaction state put the *Panel* into its logic state *disabled*. In this task should participant define first rule and two actions. Node change does not require any selections or wildcards, so here should be presented how intuitive it is to defined parameters for simple actions and rules.
15. Assign text "*EXIT*" to earlier created *Text* component. This text should be assigned from registry record at state setup phase. This should demonstrate how comfortable are users of engine with setting content on state entry instead of direct access in scene.
16. Make new active ruleset that will on mouse click on *EXIT* button shut down the application.
17. Change tile coloring ruleset so every time tile is clicked, it is colored opposite to previous color, starting with red. Use registry value for storing current state. This task is for introduction of use of registry for storage of current state.
18. In a loop initialize one dimensional array of nine elements in registry. Use this array to store tile that has been clicked on and what color it has been colored to. This task should hint the participant how rules will be evaluated.
19. Create custom *Rule* that takes registry address to one dimensional array and returns winner or fails evaluation. Purpose of this task is to verify that new evaluation element creation guide is clear and intuitive.



20. Use created rule on array of tile values and print winner in log using *Logger* action. This should show the participant that created rule works and also present means of debugging logic definitions.

### Cleanup

21. Modify any part of the game to your satisfaction. This may include adding new components, rules and styles. Do not forget to comment on everything you want to do and why. Moderator should inform you in case that is not possible in the engine. Consider applying something from optimization Section F in appendix.

**Time length** Each participant has for all tasks allocated four hours of time split into two segments divided by break. This is unusually long testing time, but it is necessary since complicated creative process is being tested. Splitting the session into two separate days might affect results since participant would have time to think about the game creation and would return biased.

## 25.4 Post test questionnaire

After the testing session was every participant asked following set of questions. These questions are for open feedback from participant, because they often do not vocalize all their thoughts during the test and *post-test* helps to get that extra information out of them.<sup>35</sup>

1. How did you like the engine?

**P1** In the end I did like it.

**P2** It still needs some work, but can I see potential in it.

**P3** I liked the simplification of GUI creation compared to java, in which it is a nightmare.

2. Did use of the engine feel intuitive?

**P1** Yes, it did, but only until the point where rules came into game.

**P2** No, it did not. The scene creation is more like making of web page which is very different from my field of work. It took me a moment to get used to the way all definitions work. On the other hand, it was not deliberately counter-intuitive, and once I was able to switch into functional programming mode for game logic, understanding of the process was not as big of a problem. Also layout component was not exactly clearly described.

**P3** Yes, but documentation could be better structured. Looking for pieces of information required reading of blocks of text.

3. What did you miss the most?

---

<sup>35</sup>Answers of participants were translated for use in this thesis.

**P1** I could use some larger set of tutorials. Documentation is good, but working examples would make it much easier to digest.

**P2** Example codes.

**P3** Some example application that has demonstrated all capabilities of the engine.

4. What was most difficult for you?

**P1** To understand how logic in state space is defined.

**P2** Construction of game grid out of layout components. It was bearable for  $3 \times 3$  grid, but I cannot imagine doing something like this for chess. Indexing in that grid was a nightmare.

**P3** Getting the custom rule to work. It kept breaking on different places and in the end I was forced to go and look how is implemented different rule for comparison.

5. Would you use this engine to make some game?

**P1** When will the engine get more time in development, yes.

**P2** Probably not, I am not too interested in board games creation.

**P3** I might give it a try, but the engine still has long way to go before it is prepared for common use.

6. If you were to implement this game in your favorite language, framework, engine, how would you compare the difficulty?

**P1** This engine would be definitely easier option.

**P2** Definition of scene would be easier in some OOP language, like java, but game logic itself with connection to scene would be much simple in this engine.

**P3** This engine would be much simpler, because it saves me from all the hassle with GUI definitions.

7. Anything you would like to mention about the engine?

**P1** Editor with for construction of definitions and scene would be helpful.

**P2** In some areas is the engine easy to use, but for the cost of more difficulties in other areas. Once something is simple in java, it could stay simple in engine that is built on java.

**P3** I can imagine logic getting very long and chaotic for more complicated games. The way definitions are made should be simplified.

## ■ 25.5 Found problems

In this section is list of selected problems with number of task they appeared in and participants that encountered that problem. Only the major usability problems are listed here since the minor ones were either fixed right after the session or are not worth being taken care of since they may disappear or change during later development of the engine.

- 3 All participants have problem understanding how registry records work. The fact name of the item defines where is value stored and not what object it creates seems confusing.
- 4 Issue for *P3* with running the project despite all correct settings (class 52 error). Problem disappears after change of *JDK* platform name.
- 7 *P1* confuses pivot setting option for *gravity* option in android GUI specification and expects centering to work without specifying *left* and *top* offsets.
- 7 *P2* expects *pivot: center;* to behave like *pivot: center center;*.
- 8 *P3* forgets to add *top* positioning for lower container. Complains about missing feature displaying borders of components on hotkey.
- 8 *P2* placed both containers in background panel so they peek out of the screen when window does not have square shape.
- 9 *P2* used for width definition property *min-width* combined with *max-width* instead of *width*.
- 10 *P2* and *P3* are having difficulties finding how interaction state styling works.
- 12 All participants missed the information that components inside of *Layout* must have defined dimensions in way so they fit inside their parent.
- 12 *P1* complains that *layout-type* cannot be set from styles definition.
- 12 *P3* is having problems with relative dimensions in inner layouts (what is row and what is col).
- 13 *P2* complains that new interaction states cannot be registered from styles.
- 14 *P2* is trying to use selectors instead of node instance.
- 14 Participants are having problems understanding how output parameters work.
- 14 *P3* is having problems understanding how rulesets work.
- 17 *P1* and *P2* are immediately trying to use numbers for players instead of strings.

- 18 All participants are having problems with figuring out how to convert style IDs to array indexes. *DictionaryTranslation* is not clear enough name.
- 19 P3 is having problems with exceptions in custom code. (related to java skill)

### ■ 25.6 Problems analysis

First of all, most of issues with understanding how things in the engine work can be solved with a set of tutorials for all features that are available. That may prove time consuming and makes no sense until is engine ready for release. Because of these reasons seems like a good idea to postpone examples creation for later stages of development. Because of testing, it is just obvious that large amount of example codes is absolutely vital for engine to have at least a little chance to success.

While positioning components, invisible lines and borders could be indeed displayed for debugging purposes. That could be achieved simply by adding styled component on top of everything that is visible only when a key is pressed. This component would consist of several debugging informations and on click on component would custom debug action transfer information from selected component to debugging component. This solution is very simple and could be achieved without modifications of core of the engine.

Complaints about larger grids being initialized in complicated way are in place, because that procedure is indeed a bit complicated, but it can be done dynamically at runtime using *InsertScene* action. It was not in example tasks because of increase of complexity. This can be solved either by a macro grid component or better integration of dynamic contents of the scene.

It proved to be a hard requirement on users of the engine to have at least basic knowledge of web development. This can only be solved by adding short course about CSS and HTML to the beginning of documentation for the engine and explain related parts using examples.

Extracting array indexes from style IDs is serious problem that needs to be addressed in some way. Currently used *DictionaryTranslation* does not scale very well and it is not practical when things are being changed a lot.

Structure of documentation in general needs some improvements, since it is quite hard for user who does not know what is he exactly looking for to look around.

### ■ 25.7 Conclusion

Testing with programmers revealed some problematic areas of the engine that need to be addressed, but since all participants finished their tasks within designated time span, I can only conclude the engine is not an utterly hopeless case.

Firstly, all found implementation problems should be addressed before further development. Issue with example codes, as mentioned, should be dealt with much later. It is important to keep creating games around features being added for verifying they are actually suitable for intended purposes. This approach would not only result in significant amount of example games, but also verification of usability.

## ■ 26 Testing with players

Test of crated game is not directly related to the engine, but it is good to verify that game created in it is sufficient and all found problems can be solved using engines features.

### ■ 26.1 Method

Testing method with players does not have to be as detailed or extensive as with programmers, because result of this testing is addressing how example game was implemented, which is not focus of this work. What I expect to get from this test is a list of features players were missing so I can verify whether it is possible to add them and at what cost. Fact that a button was hard to find or that its hover color was not as bright might be important if created game was to be shipped, but from engine developers point of view, it is a useless piece of information.

For these reasons will test proceed as simple play session of *CAH* over network, and after session finishes, each player will fill a short questionnaire.

There is no screening done for this test, because there is no specific profile of board games player and to cover whole demography of all possible players would have to be the test repeated quite extensively.

### ■ 26.2 Results

This questionnaire was filled by four participants after session of *CAH*. Game was in state presented in walkthrough 22.5 described in implementation part.

1. Did you know Cards Against Humanity before?

*P1* I knew it but did not have the chance to play it before.

*P2* No.

*P3* No, you were the one to show me the light.

*P4* No.

2. Did you enjoy played session? If not, why?

*P1* Yes. Partially because I won.

*P2* Yes, it was hilarious.

*P3* I enjoyed it, it was hilarious at times.

*P4* Yes, I did enjoy it.

3. What were you missing in the game?

*P1* I missed some notification sounds when game progresses to next phase. Replay option would be nice.

*P2* Winner of the round is not highlighted in any way.

*P3* Sound and light notification when the game moved on.

*P4* Sound on game progress would be nice. I would like to take screen shot of funny answers without having to clip print screen in MS paint.

4. Was always clear what are you supposed to do?

*P1* I got a bit confused when I was *czar* for the first time, but then it was alright.

*P2* No, ready and gamble buttons should be in the middle, not so secluded and I always got mildly confused when I entered *czar* role not having any cards.

*P3* I was a bit perplexed the first time I was the *czar*, it could have been a little more clear. Otherwise, the rest went smoothly.

*P4* I did not get how to gamble at first and had to ask.

5. How would you describe pace of the game?

*P1* Pretty good.

*P2* Waiting for other players was sometimes too long. After *czar* picked winning card, it was not visible for long enough to notice the result and who won.

*P3* Calm and peaceful. Sometimes a bit too slow, depending on how much the other people wanted to be funny or how long their cards were.

*P4* Pace was quite adequate.

6. Will you play the game again with someone else?

*P1* Might be.

*P2* Maybe, but it is difficult to gather all players.

*P3* Yes, I will. It is darkly and guiltily funny.

*P4* If I will get the chance, I would like to.

### 26.3 Found problems

Testing with players revealed few usability problems. Some can be solved by modification of game, but others will need modification of core.

Complaints about pace being slow can be solved by adding seconds timer that will force players to play their cards faster. This modification would speed up all rounds so whole game would be shorter. With this modification would be good idea to increase number of rounds one game lasts — currently set 20 corresponds to approximately 30 minutes of play. This timer could be added directly into the core, because it is feature that can be required in other games as well, but it is not necessary. There are two approaches that could be added, both utilizing custom action with separate threads. First, action creating thread that generates message events in regular intervals. This option is a bit heavy and would be complicated to control. The other option is to have action create thread that will sleep for specified time, send one message event and then die. Timer with this action would then work based on next activation reacting to previous one with optional condition for stopping the timer. This one is a bit worse, since it is not as precise. From time message event is sent until it is received and new timer is started elapses some delay which can be variable and it would be difficult to account for it. Of course action could attach previous timestamp to the event so timer can actually account for that, but that might be unnecessary for something as simple.

Adding the notification sounds could prove to be a bit complicated, because the engine does not have support for sounds in current state, and adding sound controller would require creation of sound controller module and its integration in core.

Confusion around being czar for the first time would be easily solved by placing "don't panic" message on the board that is where cards were before.

Highlighting the winner could be done by using interaction state styling and timer mentioned above. This timer would hold highlighted winner and card for a period of time and then proceed to the next round, instead of instantaneous progress.

Gamble button not being clear is problem of difference between disabled and enabled buttons. New enabled button style could make it green so it will draw attention of player immediately when it is available.

## ■ 26.4 Conclusion

Results of the testing did point out some usability problems, but none of them were critical for the game being playable or enjoyable. If game was supposed to be shipped, most of problems could be resolved without modifications of core.



## Part 6

# Conclusion

The aim of this thesis has been to simplify board games development process. Upon successful fulfilling of this pursuit, people with limited programming skills ought to be able to create a board game in digital environment. It was not intended to create complete engine as much as to create a sufficient framework in which features based on needs of its users can be added. Creating a *complete* set of tools requires years of work and testing which does not correspond with extent of bachelor thesis.

In the background part the current state of engines available for board games implementation was discussed. These existing engines were found to a different degree insufficient for easy board games creation. Possible requirements were then listed for reference as for what finished engine should contain. These are of course only a brief summary and might change over time and with amount of games being implemented in it.

The Design part contained description of event driven architecture and reasons for choosing it for implementation of engine in this thesis. Also, the essence of specific building blocks of engine architecture was described. This outline does not completely cover the matter, and certain blocks might radically change in further development.

Implementation of the engine is described in detail not only as overview for the reader of this thesis but also as a reference guide for any user of the engine. Furthermore, two implemented example games were discussed as a demonstration of previously illustrated system. One game, *Tic-Tac-Toe* serves as example of *Hello World* game for quick overview of capabilities of the engine, and second, more complicated game *Cards Against Humanity*, is supposed to demonstrate that sophisticated games are possible to be made with the engine.

The final part contains details on the testing of the engine with programmers. The testing proved that the engine is usable for creation of a simple game within hours after introduction to it. The second test was done over created game and showed that it meets criteria set by players for a pleasant board game experience.

The created engine is not complete but it already resembles the goal I set to achieve. I believe it is possible to fulfill it in every respect with more development. This does not only mean the engine is only usable for creating of board games. It can also be adapted for any kind of application that will benefit from event driven core and simple scene construction.



## ■ 27 Further development

This engine is not a complete product that should be used by creators on daily basis but, as testing proved, it is on the right track. In future development, the set of actions should be expanded, new components and generally features made available to users but that would not be the main goal. Results from testing show that consulting of documentation is not fast nor comfortable for creators. That is the reason for the need of an interface that will allow the user to manipulate all existing definitions using graphical elements instead of writing them by hand. This would enable integration of documentation directly in the engine by showing available options whenever possible.

# Appendices



## A Getting started

This section describes how to setup working environment, how to orientate in it and brief description of files contained within template application.



### A.1 Installation

This engine was developed and tested under *JDK 8* and *JRE 8*. Please make sure you have installed these versions of java virtual machines. They are included on attached DVD or on official web of *oracle* <sup>36</sup>.

Although it is not required, I would advise you to use *Netbeans IDE* in version *8.0.2* or higher for including of project. Settings of *JAXB* builds are already in place there and you will avoid any other problems with project definition. Of course it should be possible just to copy source files to new project in different *IDE*.

In the code, as a user, you may be interested in three main areas. *Testing* package that contains main class, *Debugger* for printing out events in the system and *Logger* that is processing all errors that may occur. Next two places are packages *Action* and *Rule* in *Core.Evaluation* package. These contain basic and custom action and rule classes.

Created template application definitions are located in folder *data/template* and consists of *registry.xml*, *states.xml*, *scene.xml*, *style.sts* and folder with *main.xml*, *debug.xml* and *game.xml*.

**Registry** This file contains registry definition that starts with *root* element. Consider all current predefined values as mandatory and nothing bad should happen.

---

<sup>36</sup>[www.oracle.com](http://www.oracle.com)

**States** States file contains includes or direct definition of states in the application. There is also *init* element for identification of entry states. These are activated once state machine is loaded and their setup rulesets are evaluated.

**Scene** This file contains definition of scene similar to *HTML*. Scene files can always contain only one root element and any element in the scene is not allowed to use style ID *#root*.

**Style** File with styles resembles *CSS* definition and is connected to defined scene. Whenever are styles applied, they are applied to whole scene!

**Main** In this state defined initial loading process of the application. Put here all persistent rulesets that do not have ties to any game part in particular.

**Debug** This state contains debugging rulesets and should be disabled when application is being deployed.

**Game** This is initial game state which serves as entry playground for first applications. Do not hesitate to add more states once game gets more complicated.

After running the project, new window should appear in middle of the screen with text *HELLO WORLD* in middle. Console output should then show whether accelerated graphics support is enabled and if game state was successfully set up. When everything loads, message about platform start is logged as well.

## ■ B Scene graph

Scene graph in this engine is composed out of subclasses of *SceneNode* described later on, which are laid down in tree graph using grouping elements extending *GroupNode*. New nodes and components are added more statically from fixed definitions, and initialization of each element is rigid as well. That is mainly because of use case of creating new components is much less common compared to use case of using them. This way is the ease of creating new ones sacrificed for for simplicity and clarity of definition of scene using *XML*.

### ■ B.1 File specification

File with scene lays only few requirements on user, while most of them are common with *XML* file format.

Every scene file that is to be loaded in must have a single component at root level because of *JAXB* unmarshaller being limited in that way. Root element can be any component available.

In actual scene file may be defined only components and not scene nodes. Scene nodes are in *XSD* definition made abstract to prevent creation of unprepared nodes to be directly in the scene. One of the reasons for this is implementation property of *GroupNode* — not keeping dimensions and position in the scene. This way using directly *GroupNode* would disrupt whole scene layout without any apparent reason.

Tags for components that may contain other elements do not have to be defined as a pair when they do not contain any other elements.

### ■ B.2 Available nodes

Graph nodes are elements from which the scene of the engine is composed of, but graph nodes themselves are abstract and not to be used directly. For user are purposed components, which are discussed later.

#### ■ B.2.1 SceneNode

*SceneNode* is base class of all nodes in the scene. Its main purpose is to carry scene reference to all nodes in the scene so they can interact with it. Parent reference is carried as well for changes that might influence parents behaviour. Lastly is attached current logical state of the node, which is defined by each extending class.

Every scene node can have defined a layer property on which that very node resides in relation to its peers. This information is used to resolve rendering order and order in which are resolved inputs over components. However, by default is this value on zero.

#### ■ B.2.2 PaintableNode

Paintable node, as name suggests, is a node that may have some content to be painted in the scene. With that comes definition of position and dimensions on the screen, in two versions - parent version, which are values pushed by parent as available, and absolute version, which is calculated by node itself. These two are for unmodified node identical, but may differ later on.

With paintable node comes first defined logic state — invisible — which does actually nothing in this node since its rendering methods are not implemented either, but it can be called upon and tested. State name is invisible because of all logic states being by default in not-set state, so visibility is controlled by setting the state up.

#### ■ B.2.3 StyleableNode

This node is intended for work around visual styles of other styleable nodes. This node still does not render anything for its only function is keeping and managing style definition itself while not caring about the content.

*StyleableNode* is expanding scene graph state space by new set of interaction states which are for purely cosmetic changes and should not in later implementation affect

working of nodes in any way, shape or form. This expansion does influence logic states which gain in this node ability to have styles attached to them the same way styles are attached to interaction states, so for example disabled logic state may have attached style properties making button gray.

Similarly to *CSS*, styleable nodes have their own style ID and style class which serves for selecting of nodes within scene using selectors similar to *CSS* (yes, this part is heavily inspired by *CSS*).

### ■ B.2.4 GroupNode

Group node is node purposed completely for managing of grouping of other nodes and passing all calls to children correctly. It uses layer information from paintable node to process render and event propagation in correct order.

## ■ B.3 Available components

This section shortly describes currently implemented components that can be used within the scene. For further details, please, consult documentation of specific properties.

### ■ B.3.1 Container

*Container* is base positioning component in the scene supporting similar properties as *HTML div* element with that difference it will not render anything. *Container* itself does not care about any of its children and does not affect them in any way. It is intended for frame positioning only. It can be used as placeholder in *Layout*, since it is not rendered, or for logical grouping of other components, because as long as no properties are set to it, it behaves as parent, with respect to parents padding of course.

Besides positioning, container serves for size management as well. For that there are properties for dimensions settings, minimal dimensions and especially important aspect ratio setting for preventing imagery from deforming without having to define its dimensions by hand.

It is derived directly from *GroupNode* adding these properties: *width* D.2.6, *height* D.2.6, *margin* D.2.7, *margin-(left|right|bottom|top)* D.2.7, *padding* D.2.8, *padding-(left|right|bottom|top)* D.2.8, *pivot* D.2.9, *(bottom|top|left|right)* D.2.10, *(min|max)-(width|height)* D.2.11, *aspect-ratio* D.2.12.

### ■ B.3.2 Panel

*Panel* is base display component that extends properties of *Container* to background patterns, border images and lines along with cutoff for displayed content. This component is not only visible to user, but as well has capabilities to work with mouse events

## B SCENE GRAPH

happening over it, and that in two modes, either masked, which will accept all mouse events that are over pixels not completely transparent, or non-masked, that accepts all events over area defined by *Container* bounds.

With ability to recognize user input from mouse, this component has two predefined logic states, *hover* and *click*, that are activated when their respective action occurs over the component. This makes the system seem more responsive, because state change does not have to go through game logic to verify change. These states are very similar to interaction states in *CSS*.

*Panel* is directly derived from *Container* adding these properties: *background-color* D.2.13, *background-image* D.2.14, *background-repeat* D.2.15, *border-size* D.2.17, *border-color* D.2.18, *border-radius* D.2.19, *border-position* D.2.20, *border-image* D.2.22, *border-image-(bottom|top)-(left|right)* D.2.22, *border-mode* D.2.21, *cursor* D.2.23, *background-image-alpha* D.2.16, *solid* D.2.24, *mask* D.2.25.

### ■ B.3.3 Text

*Text* is component for displaying styled text on top of a *Panel* from which is text component derived. Along with all inherited properties, *Text* component has also capability to adjust its size based on its content instead of completely filling the parent. *Text* supports line wrapping to specified or received width, but that does not influence preferred dimensions call.

It is derived from *Panel* to remove necessity of wrapping the text in different *Panel* and for keeping mouse event generation on related text. Following properties are added: *font-color* D.2.26, *font-name* D.2.27, *font-size* D.2.28, *font-style* D.2.29, *text-align* D.2.30, *line-stretch* D.2.32, *line-wrap* D.2.31.

### ■ B.3.4 Layout

*Layout* is basic positioning component for spreading children over given area, vertically or horizontally, with respect to their alignment. Main purpose of layout is to be able to spread child components evenly, which is especially complicated when we do not know how many of them will be there.

*Layout* is derived from *Container*, since it is purely for positioning of components. Following properties are added: *layout-type* D.2.34, *spacing* D.2.35.

## ■ B.4 Adding new components and nodes

When user wants to add new component, there is certain procedure that should be followed for full integration of added component to the scene, but since daily addition of components is not exactly common intended use of the engine, it is not all that much of a problem.

**Create new class** As a starting point would be creating new class in package *GUI.Component* and having the class extend one of the existing components or nodes depending on intended use. In case new component should be included in passing of events, it should extend the *GEventReceiverNode* interface and for delta time render updates (animated components for instance) *ActiveNode* interface.

**Pick style extension** Next needs to be picked what new properties will be added to that component, either defined from *XML* or from stylesheet definition. These properties should be properly formalized and their names added to set of constants in *GUI.Style.StyleConstants*.

**Expanding JAXB definition** In *XSD* binding for scene (*src/scene.xsd*) add new component by inserting structure

```
<xs:element name="newName" type="XNewName" substitutionGroup="node"/>
<xs:complexType name="XNewName">
  <xs:complexContent>
    <xs:extension base="XSuperClassComponent">
      <xs:attribute name="prop" type="xs:string" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

where *newName* is name that will be used in *XML* definition for constructing that component, *XNewName* will be name of inner class representing data from *XML* definition and *XSuperClassComponent* name of inner class this new component is based on (*XText*, *XContainer*, ...). After finishing this modification, rebuild the structure from modified *XSD* file.

**Adding component to builder** Component needs to be added to scene builder so it knows how to instantiate it. In file *Loading.SceneBuilder* add in method *build* line corresponding to just created class. Keep in mind that this addition **MUST** appear before superclass of created component.

**Overwrite construct** When scene builder instantiates new component, method *construct* is called on it for initializing all properties that may be defined from *XML* definition. Because of those overwrite in your component this method and load data from *XML* definition into appropriate fields. In case of style properties, you might want to set them as string properties into base style of superclass and have them evaluated with everything else when styles are applied.

**Apply style extension** When styles should be applied, method *applyStyle* is called on styleable nodes. Put any processing of style setting in extension of this method.

## C EVENTS

This procedure should ensure you have correctly added new component, but in any case, please view first current implementation to see working examples.

### ■ C Events

In this section is described general event and all currently implemented events in the engine.

All event classes use prefix *G* in front of event word to be easily distinguishable from *awt* events and other kinds residing in java.

#### ■ C.1 Available events

All available events are derived from abstract class *GEvent* and must be therefore externalizable for network purposes. They also share the same ID generator which means that even if two events are not of the same class, their ID is guaranteed to be different, however, that does not count for event received over network.

Listed names next to events in following sections do not contain under *name* actual name of property, but just keyword related to it! Look in implementation of specific event when operating with them in java code. Keep in mind that only properties related to recognized types are correct!

##### ■ C.1.1 Event

**Name** *GEvent* (abstract)

**Property** integer *client* that is distinguishing to what client event belongs to

**Property** boolean *network* that is on *true* whenever should be event distributed over network (default *false*)

**Property** long *timestamp* saying when was event created in milliseconds

**Property** long *id* containing value from generator assigned to *GEvent* class

**Description** This is abstract class used as mandatory predecessor for all events in the engine. It purely serves for encapsulation of mentioned properties.

##### ■ C.1.2 Window event

**Name** *WindowGEvent*

**Property** integer *type* from set of tuples *visibility:0*, *resize:1*, *movement:2* and *focus:3*.

**Property** boolean *shown* on true if window entered visible state



**Property** boolean *focused* on true if window gained focus

**Property** point *location* of the window on the screen

**Property** dimension *dimensions* of window

**Description** This event is for reporting of platform window related events. They may have some use in the game, but generally is window event for renderer only.

### ■ C.1.3 Mouse event

**Name** *MouseEvent*

**Property** integer *type* of event from set of tuples *moved:0*, *scrolled:1*, *pressed:2*, *released:3*, *dragged:4*, *clicked:5*, *entered:6* and *exited:7*.

**Property** point *position* on the window

**Property** double *scrolled* distance

**Property** integer *button* code *LMB:1*, *wheel:2* and *RMB:3*.

**Description** This event serves for translating of mouse event over current application window.

### ■ C.1.4 Message event

**Name** *MessageEvent*

**Property** string *name* of the event that is user defined

**Property** object *content* being carried with the message (may be *null*)

**Description** This event is for general purpose messages being sent across the engine or between clients over network. When this event is attempted to be marked as network, check is made on content being *Serializable*. In case this check does not pass, mark is not done and error is logged. This event may carry more than one objects (see Action E.3.14 and Rule E.2.1 for details).

### ■ C.1.5 Keyboard event

**Name** *KeyboardEvent*

**Property** integer *type* of the event from set of tuples *pressed:0*, *released:1* and *typed:2*.

**Property** integer *code* of key being pressed or released

## D SCENE STYLES

**Property** character *char* being typed

**Property** integer *location* of key being pressed or released on keyboard, such as numpad or standard

**Description** This event is for reporting of user interaction with keyboard.

### ■ C.1.6 Interaction event

**Name** *InteractionGEvent*

**Property** GEvent *event* source of the interaction

**Property** GEventReceiverNode *node* source being interacted upon

**Description** This event is there for components to be able to report their interaction with events that arrived in scene. Time of this event being returned is time of source event.

## ■ D Scene styles

This section covers information about style definition in general and should work as factual reference guide.

### ■ D.1 File specification

For purposes of defining styles for scene elements is present new file format *.sts* which serves for stylesheet definitions. These are in structure very similar to *CSS*, but recognized properties are purely users responsibility as well as their inner implementation.

Each *.sts* file may contain unknown amount of rulesets (*styles*) that are introduced by element chains (*selectors*) that define which elements should be influenced by that very *style*. Separate *selectors* may overlap in terms of elements being selected by them. That means that one element can gain properties settings from different rulesets. Conflicts are resolved by rule — last assigned value is valid value.

#### ■ D.1.1 Comments

As opposed to *CSS*, in *.sts* file may be placed two types of comments (single line, multi-line), so it is much easier to document style definitions.

**Single-line** Single-line comments are started with *//* and will guarantee that everything until the end of line will be ignored. Forward slash characters may be used inside string chain definition freely without being backslashed.

**Multi-line** Multi-line comments are enclosed in `/* */` tuple and everything within them is ignored. Both forward slash and asterisk can be used freely inside string chain definition without the need of being backslashed to not be considered a comment.

### ■ D.1.2 Selectors

Each row that is not part of style rule set or comment or include command has to be selector chain. Each selector chain is composed from selector *members* of different interpretations based on their prefixes and suffixes.

Only allowed characters for members names are basic *ASCII* letters and numbers, while no name can start with a number. This restriction comes from open possibilities of expansion of capabilities of styling and selections. This way can be ensured backwards compatibility along with expanding basic selector construction by designating specific characters to new functionalities. Following members are recognized:

**Class members** Class members are always introduced by dot and followed by at least one *ASCII* letter. Class members are to be used as general collections of properties for elements of similar purpose — elements of the same class may appear in the same scene graph multiple times.

**Identifier members** Identifier members are always introduced by hash tag followed by at least one *ASCII* letter. Identifier members are intended to be unique across scene graph, and thus be used as starting point for some local selection chains.

**Name members** Name members are not introduced by any special character and represent class name of java object that is representing target element. That being said, all naming rules on name members are identical with java class naming specification.

**All member** All member is composed of single asterisk character and will be treated as *equal-to-everything*. Intended to use as end selectors of selection chain or as mandatory layer in between of two members.

Selector may have defined at last member the state to which the style belongs by separating member name and state name with colon. That will ensure style belonging to the selector will be applied only when are concerned elements in specified state.

After each selection chain there **MUST** be pair of enclosing braces that may contain specific rules, but if empty style is encountered, user is warned about it. Between two selection members there **MUST** be at least one whitespace. Between braces and surrounding parts may be any amount of whitespaces so it is possible to have braces on the same line with selector but as well on the next one, if that is your coding style.

## D SCENE STYLES

There may be multiple selectors for single style divided by comma. Such notation is then treated as if there was same style several times with different selectors. This is supposed to prevent copy pasting of styles across the stylesheet, but keep in mind that last defined wins, so it may be still necessary for overwriting some changes.

### ■ D.1.3 Styles

In enclosing braces of selector may be defined any amount of property tuples that are composed of key and value. Keys are predefined words containing only lowercase letters and hyphens for word division. Each style definition **MUST** end with semicolon. Values can be one of following types:

**Number** Number type can define either signed integer value or signed double value.

This type is recognized by minus or number or dot being first character of value part. Number may optionally end with units definition: pixels as *px* that defines absolute on screen pixel value or percents relative to some other value depending on implementation of that very element.

**String** String type can define any string of characters, but they must be enclosed within pair of quotes. Use of additional quotes is possible, but each occurrence must be escaped by backslashes.

**Keyword** Keyword type is unenclosed string that may contain only lower case letters, dashes for separating words and spaces for cases when you want more keywords for one property. This keyword content will be stored trimmed <sup>37</sup> and splitting the string is job of component using it.

### ■ D.1.4 Includes

Include command is recognized when line starts with sequence *#include*. Then should follow path enclosed in pair of quotation marks. Path may be relative to project root or absolute, using forward slashes only. Include line must end with semicolon. Includes are systematically evaluated in loop with comments removal. Firstly are parsed out all comments and their content, then are resolved includes by pasting contents of files they are pointing to instead of include lines and then are checked comments again. Failed includes are reported and will not break the parsing of stylesheet.

## ■ D.2 Available properties

In this section are listed all available properties with their use, effect and components they are defined in. In case this list is not sufficient, please do explore actual implementation, which is heavily documented by itself.

---

<sup>37</sup>Trim is function that removes whitespaces before first non-whitespace character and after last.

In this list are similar properties aggregated by use of regular expression notation so actual names from example *padding-(left|right)* would be *padding-left*, *padding-right*. These properties are not defined separately because of their analogical use, so please take that into account while reading through them.

No property is mandatory for any currently implemented component and it should never be in any of the expansions. Any property should be additionally removable by using it with its name and value *none*.

### ■ D.2.1 Layer

**Owner** SceneNode B.2.1

**Name** *layer*

**Values** integer

**Default** *0*

**Description** This property serves for sorting of elements in the scene in different than insert order. When element has higher layer value than its peers, it will receive mouse events as first and will be rendered last. This setting does not translate outside of elements peers in single group. When all elements have their layer identical, they will be processed in their insert order. In case some of them have different layers, order of those with same one is not guaranteed since used sorting algorithm is not stable.

### ■ D.2.2 Visible

**Owner** PaintableNode B.2.2

**Name** *visible*

**Values** boolean

**Default** *true*

**Description** This property is for changing of visibility of related node. When node is not visible, it will not only be left out of rendering phase call, but it will also not receive mouse events that are happening over its surface. Visibility is not possible to be changed from stylesheet definition because it would cause constant blinking on every action happening over it, which is probably not required behaviour in any imaginable scenario.

### ■ D.2.3 Style ID

**Owner** StyleableNode B.2.3

**Name** *id*

**Values** string according to regex  $(A-Za-z)+(A-Za-z0-9)^*$

**Default** *not set*

**Description** This property is present for selecting nodes in the scene using defined ID keys. It is not possible to change style ID from style definition. It is strictly initialized inside *XML* file with scene. This ID must be unique across the scene at all times.

### ■ D.2.4 Style class

**Owner** StyleableNode B.2.3

**Name** *class*

**Values** strings according to regex  $(A-Za-z)+(A-Za-z0-9)^*$  separated by single space

**Default** *not set*

**Description** This property is similarly to style ID intended for scene graph selections, but with the difference that one node can have multiple defined classes and there can be multiple nodes with the same class. This is a tool for grouping of appearance and behaviour of nodes by their common abilities. Keep in mind that multiple classes are defined NOT by multiple definition of *class* attribute, but by putting more keywords separated by space inside single attribute definition. This property cannot be set from style definition.

### ■ D.2.5 Interaction states

**Owner** StyleableNode B.2.3

**Name** *states*

**Values** strings according to regex  $(A-Za-z)+(A-Za-z0-9)^*$  separated by single space

**Default** *not set*

**Description** This property serves for registering of recognized interaction states so they can be addressed in selectors. These states have no affect on logic of the game and are purely for visual changes. Styles that are stored under these states are applied in order they are defined inside of this property with the exception of states predefined in constructor of component or node. Generally logic states have higher priority during the application and will overwrite settings from interaction states. Keep in mind that multiple states are defined NOT by multiple definition of *states* attribute, but by putting more keywords separated by space inside single attribute definition. This property cannot be set from style definition.

### ■ D.2.6 Dimensions

**Owner** Container B.3.1

**Name** (*width|height*)

**Values** real numbers with possible suffix *px* (implicit) or *%* (relative)

**Default** *100%* for both

**Description** These properties serve for dimensions definition of component in scene. In case of dimensions in pixels is dimension on respective axis applied if not said otherwise in further implementation (outside of container) and dimension in percents is related to value given by parent as available space. In default setting will component fill all given space.

### ■ D.2.7 Margin

**Owner** Container B.3.1

**Name** *margin(-(left|right|bottom|top))?*

**Values** real numbers with possible suffix *px* (implicit) or *%* (relative)

**Default** *0* for all names

**Description** This property is similarly to *CSS* designated for spacing around the component. Space created around component by margin is not filled by content of that component, or partially in case of border setting center or outside. Contrary to *CSS* margin behaviour, margins of touching components on same level do not overlap but are added together, but only when they are inside of a layout component (components do not influence each other by themselves). Margin setting without side specification will change margin setting for all sides at once and then will be overwritten by possible specific definition. In case of margin setting in percents, resulting value for each side is always related to parents dimension in respective axis.

### ■ D.2.8 Padding

**Owner** Container B.3.1

**Name** *padding(-left|right|bottom|top)?*

**Values** real numbers with possible suffix *px* (implicit) or *%* (relative)

**Default** *0* for all names

**Description** This property is similar to margin, but with the difference that padding serves for offset of contained components. Padding setting without side specification will change padding setting for all sides at once and then will be overwritten by possible specific definition. In case of padding setting in percents, resulting value for each side is always related to parents dimension in respective axis.

### ■ D.2.9 Pivot

**Owner** Container B.3.1

**Name** *pivot*

**Values** two strings defining pivot on x and y axis using keywords (*left|center|right*) and (*top|center|bottom*) respectively.

**Default** *left top*

**Description** This property serves for shifting to what point on the component is the position related. When we define that position is certain distance from the right side, we might want as well to shift pivot to the right so the right side of the component is defined distance from right side of parent. Centering component in middle of parent is then done by setting *pivot: center center*; and left and top positions on *50%*.

### ■ D.2.10 Position

**Owner** Container B.3.1

**Name** (*left|right|bottom|top*)

**Values** real numbers with possible suffix *px* (implicit) or *%* (relative)

**Default** *not set*



**Description** This property is for positioning of element relatively to parents boundaries. Setting of position directly will override effect of margin, which is used when fixed positioning is not set. Relative positioning is calculated from parents dimensions on respective axis and is always related to pivot, not to boundary of the component, so *left: 100%*; will produce same result as *right: 0%*;

### ■ D.2.11 Min-max dimensions

**Owner** Container B.3.1

**Name** *(min|max)-(width|height)*

**Values** real numbers with possible suffix *px* (implicit) or *%* (relative)

**Default** *not set*

**Description** This property serves for defining edge case dimensions of components in the scene. This calculation is done after all other dimension influencing calculations, except aspect ratio, which may violate requested dimensions. Relative values are calculated out of parent dimensions on respective axis.

### ■ D.2.12 Aspect ratio

**Owner** Container B.3.1

**Name** *aspect-ratio*

**Values** real numbers preferably around *1* or *auto*

**Default** *not set*

**Description** This property is intended for keeping certain components in fixed aspect ratio, for example when there is imagery on components background and not keeping aspect ratio in line would deform the image. Aspect ratio is during its application battling minimum and maximum dimensions, but if component is not able to be sized to fulfill min-max requirements, they will be loosened for sakes of aspect ratio. Aspect ratio is one of the properties that may be set automatically when the assigned value is *auto*. This means extending component can overwrite call for aspect ratio generation and return different value than *1*. In case of panel, returned ratio is ratio of set background image, if there is one set.

### ■ D.2.13 Background color

**Owner** Panel B.3.2

**Name** *background-color*

**Values** color in classic hexadecimal definition *#RRGGBB*, expanded version *#AARRGGBB* where *AA* is hexadecimal value for alpha channel, or word name of color from predefined testing palette *white|black|red|green|blue|yellow*. Alpha channel is inverted so when it is by default not set, visibility is on maximum.

**Default** *not set*

**Description** This property is for coloring of the background of the component. Surface being colored is surface defined by final position and dimensions, but not surface covered by margin. Background color is rendered first of all background properties so it is covered by other imagery.

### ■ D.2.14 Background image

**Owner** Panel B.3.2

**Name** *background-image*

**Values** string path to image on the computer

**Default** *not set*

**Description** This property for spreading imagery from external file over component in manner defined by repeat property. This imagery is rendered over color as second and is covered by border line and border image. Path to the image may be relative to execution environment or absolute.

### ■ D.2.15 Background repeat

**Owner** Panel B.3.2

**Name** *background-repeat*

**Values** string containing two keywords (*repeat|stretch|fixed*) for x any y axis of image

**Default** *stretch stretch*

**Description** This property is specifying how background image should be used in three modes for each axis. First, repeat mode will repeat image across respective axis in its original size. Second, stretch mode will stretch image to the dimension of component on respective axis. Third, fixed mode will let the image sit on the component in its original size in upper left corner. Keep in mind that the smaller image is being repeated over big area, the longer it takes to render it because of large amount of separate calls.

#### ■ D.2.16 Background image alpha

**Owner** Panel B.3.2

**Name** *background-image-alpha*

**Values** real numbers in range  $]0,1[$  where 1 is fully visible.

**Default** 1

**Description** This property serves for reducing transparency of background image of a component without having to change image itself. Keep in mind that alpha is not inverted in this property, as opposed to other coloring definitions.

#### ■ D.2.17 Border size

**Owner** Panel B.3.2

**Name** *border-size*

**Values** real numbers with possible suffix *px* (implicit) or *%* (relative to maximum dimension)

**Default** 0

**Description** This property is for defining the size of border rendered around solid surface of component. Border size affects not only rendered single color border, but it also affects how is border image rendered. In border image is size of the border set in a way so all border images fit in the bounds, but adjusting it manually may result in border image cutoff.

■ D.2.18 **Border color**

**Owner** Panel B.3.2

**Name** *border-color*

**Values** color in classic hexadecimal definition *#RRGGBB*, expanded version *#AARRGGBB* where *AA* is hexadecimal value for alpha channel, or word name of color from predefined testing palette *white|black|red|green|blue|yellow*. Alpha channel is inverted so when it is by default not set, visibility is on maximum.

**Default** *not set*

**Description** This property defines what color is used while rendering border line.

■ D.2.19 **Border radius**

**Owner** Panel B.3.2

**Name** *border-radius*

**Values** real numbers with possible suffix *px* (implicit) or *%* (relative to maximum dimension)

**Default** *0*

**Description** This property defines circular radius around the edges of component. This radius is rendered on top of imagery which is for its sakes cut off so background imagery or color doesn't peek from behind of the component. Setting this radius to *50%* on square component will create a circle.

■ D.2.20 **Border position**

**Owner** Panel B.3.2

**Name** *border-position*

**Values** one of keywords (*inside|center|outside*)

**Default** *center*

**Description** This property serves to purpose of placing border in specific relation to edge of solid part of the component.

### ■ D.2.21 Border mode

**Owner** Panel B.3.2

**Name** *border-mode*

**Values** one of keywords (*dotted|dashed|dotdashed|solid*)

**Default** *solid*

**Description** This property specifies how is line of border stroked. Spacing in stroke patterns are derived from current border size.

### ■ D.2.22 Border image

**Owner** Panel B.3.2

**Name** *border-image(-(top|bottom|left|right))?* or *border-image-(top|bottom)-(left|right)*

**Values** string path to an image on the disk

**Default** *not set*

**Description** These properties are here for defining more complicated borders composed out of border images. There are three types of this definition. Firstly, common border image that is set to all sections, then there are sides with one specifier, and at last corners with two specifiers. Corner images are just placed in their position, when side images are repeated along their axis. Images are adjusting border size to fit this image border in cutoff area of background cache, so make sure you do not have border color defined when using border images.

### ■ D.2.23 Cursor

**Owner** Panel B.3.2

**Name** *cursor*

**Values** string path leading to file with image representing cursor

**Default** *not set*

**Description** Purpose of this property is to allow user to define specific cursors over components, especially with relation to states. Hover on component can display active cursor and click can animate cursor grab.

### ■ D.2.24 Solid

**Owner** Panel B.3.2

**Name** *solid*

**Values** boolean

**Default** *true*

**Description** This property causes related component to let through mouse events depending on its setting. When on *true*, no mouse events that are caught by this component are then silenced and will not land on anything else. That comes handy when covering normal interface with temporary menu, since with this option it is not necessary to disable underlying interface, it will not get mouse events at all.

### ■ D.2.25 Mask

**Owner** Panel B.3.2

**Name** *mask*

**Values** boolean

**Default** *true*

**Description** This property serves for masking out mouse events by contents of background image. It may have more complicated shape with transparent segments and mouse events will be caught only over non-transparent pixels. This does not apply to pixels generated by text! These are for this functionality ignored.

### ■ D.2.26 Font color

**Owner** Text B.3.3

**Name** *font-color*

**Values** color in classic hexadecimal definition *#RRGGBB*, expanded version *#AARRGGBB* where *AA* is hexadecimal value for alpha channel, or word name of color from predefined testing palette *white|black|red|green|blue|yellow*. Alpha channel is inverted so when it is by default not set, visibility is on maximum.

**Default** *black*

**Description** This property serves for setting up color of text related text component is rendered in.

#### ■ D.2.27 Font name

**Owner** Text B.3.3

**Name** *font-name*

**Values** name of font to be used

**Default** fixed font definition to *SERIF*

**Description** This property defines what font should be used for related text component. Defined font must be on the computer, it is not possible to define custom fonts from path at the moment, but used fonts may be included with application which will make them available for use.

#### ■ D.2.28 Font size

**Owner** Text B.3.3

**Name** *font-size*

**Values** real numbers with possible suffix *px* (implicit) or *%* (relative to height of parent)

**Default** *14*

**Description** This property is for setting the font size. Unfortunately it is not possible to have relative font size to size of related component since size of that component is partially decided by content and actual font size.

#### ■ D.2.29 Font style

**Owner** Text B.3.3

**Name** *font-style*

**Values** string containing keywords (*bold|italic*) separated by spaces

**Default** *not set* (plain)

**Description** This property is for adding options bold and italic to font style. Underline and crossed are not supported.

■ D.2.30 Text align

**Owner** Text B.3.3

**Name** *text-align*

**Values** string containing tuple of keywords (*left|center|right*) (*top|center|bottom*) separated by spaces

**Default** *center center*

**Description** This property is for aligning actual text inside of its component. Vertically is alignment done over whole text block, horizontally is aligned each line separately.

■ D.2.31 Line wrap

**Owner** Text B.3.3

**Name** *line-wrap*

**Values** boolean

**Default** *false*

**Description** This property is defining whether text should wrap automatically in case it does not fit inside of predetermined width of the component. This option should not be combined with line stretch, since that does not calculate with wrapped lines.

■ D.2.32 Line stretch

**Owner** Text B.3.3

**Name** *line-stretch*

**Values** boolean

**Default** *false*

**Description** This property defines whether size of the component should be adjusted to the requirements of contained text. This option will count the rows, measure their width and will come up with preferred dimension for the component. Then will system make an attempt to fulfill that preference while keeping all other restrictions on place. I that will not work out, text will overflow he component.



### ■ D.2.33 Text content

**Owner** Text B.3.3

**Name** *content*

**Values** string

**Default** *empty*

**Description** This property is for specifying contents of text component from XML file. This property cannot be set from style definition.

### ■ D.2.34 Layout type

**Owner** Layout B.3.4

**Name** *layout-type*

**Values** string keyword (*horizontal|vertical*)

**Default** *horizontal*

**Description** This property serves for definition of layout orientation, either vertical or horizontal. This property cannot be defined within style and is to be changed from XML file.

### ■ D.2.35 Layout spacing

**Owner** Layout B.3.4

**Name** *spacing*

**Values** string keyword (*gap|border|combine*)

**Default** *none (not set)*

**Description** This property is for setting spacing between child components inside of layout component. Components are first evaluated by their align and outer dimensions, and then is remaining space split by the defined value. For *border* is the space split evenly before first component and after last component. In case of *gap* is remaining space split in spaces (gaps) between components while border components are leaning towards sides. Lastly, *combine* will just split remaining space between both — spaces between components and around borders. This splitting is happening only along axis defined as orientation of layout! For spacing to work properly, all components must have defined sizes by themselves.

## ■ D.3 Adding new properties

Process of adding new property is similar to process of adding new component with some steps left out.

**Expanding JAXB definition** First we need to expand *JAXB* definition in *src/scene.xsd* file by adding attribute

```
<xs:element name="name" type="XClass" substitutionGroup="node"/>
<xs:complexType name="XClass">
  <xs:complexContent>
    <xs:extension base="XSuperClass">
      <xs:attribute name="prop" type="xs:string" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

inside of the related definition. Then we have to regenerate classes defined in this xsd and we are all set there.

**Specifying constant** Now we should to pick and store constant name in style constants *GUI.Style.StyleConstants* file. This constant should be prefixed by *STYLE\_* to be easily recognizable within the code.

**Expanding construct** Next we need to expand construct by loading added property in related component into base style or elsewhere by the use, so it can be assigned from *XML* structure.

**Apply style** As the last thing, we might need to expand apply style method in related component to load newly created property from style definition and process it in intended way.

This process is much simpler than adding new component and there is more than enough working examples to get inspired by.

## E Logic

Used state machine for logic is stored in *XML* file containing specific states and their content. This file can be defined only once and separate state spaces cannot be merged at this moment. Further in this section is described how to define logic for internal use.

### E.1 State space file specification

*XML* file containing state space could be summarized by simple example:

```
<statespace>
  <state name="stateName">
    <setup>
      <group operation="and">
        <rule class="RuleName">
          <param name="a" access="var" value="varName"/>
        </rule>
        <action class="ActionName">
          <param name="b" access="reg" value="reg.path[1]"/>
        </action>
      </group>
    </setup>
    <active>
      ...
    </active>
    <teardown>
      ...
    </teardown>
    <include path="./..." mode="setup$\vert$active$\vert$teardown"/>
  </state>
  <include path="./..."/>
  <init>
    <statename name="stateName"/>
  </init>
</statespace>
```

As example shows, state space consists of states and definition which states are entry ones. Then each state may contain three types of element collections that differ in time they are called. *Setup* is called on entry in the state, *active* is called on processing of subscribed event and *teardown* is called on exit from state.

Available includes are present to keep the definition logically separated and simple.

Keep in mind that during repeated parsing process are includes not checked and will be used from cache, even though they might have changed!

### ■ E.2 Available rules

In this section are described currently implemented rules. *Rule* class does not differ from *Action* class all that much, but they are separated for readability.

Outputs of rules do not have to be specified in *XML*, if they are not, their value will appear on their name in variable space. If inputs are not marked as mandatory, they do not have to be specified. All classes representing rules must have *Rule* as a suffix in name. This suffix is implicit and not used in this section.

#### ■ E.2.1 Check message

**Name** *CheckMessage*

**In** integer *sourceIn* client ID that is checked

**In** string *regex* used to validate message name

**Out** integer *sourceOut* client ID that was found in the event

**Out** object *messageContent* that was received

**Description** This rule will take given event and in case it is instance of it will attempt to match currently set *regex* on its name and store content of the message in variable space under *messageContent* key. In case there is no regex set, all messages will pass. Client ID attached to received message may be checked as well if *sourceIn* parameter is set. This rule has special behaviour tied with *SendMessage* action. Content inside this rule can be suffixed with any number of strings to attach multiple objects to the message. Common practice should be either using array indexes *messageContent[0]* or map keys *messageContent[data]*. These additional keys must be defined as outputs to be stored on defined place, it is done automatically only for unsuffixed key.

#### ■ E.2.2 Compare numbers

**Name** *CompareNumbers*

**In** mandatory double *num1* first compared

**In** mandatory double *num2* second compared

**In** mandatory string keyword *operation* from pool (*lt|b|leq|beq|eq|neq*) which represent lesser, bigger, lesser or equal, bigger or equal, equal, not equal respectively.

**Description** This rule serves as compare filter for two numbers.

### ■ E.2.3 Keyboard button released

**Name** *KeyboardReleased*

**In** integer *keyCode* representing code of checked key

**Description** Will verify whether given event is KeyboardGEvent, then whether its type is released and at last whether its *keyCode* is same as the one acquired using settings of this rule.

### ■ E.2.4 Keyboard character typed

**Name** *KeyboardTyped*

**In** integer *keyCode* representing code of checked key

**In** string *regex* used to validate received character (example use would be *[A-Za-z0-9]*)

**Out** string *keyChar* that was actually received

**Description** Will recognize keyboard typed event, store pressed key in output. If regex and/or key code are set, check on match is done.

### ■ E.2.5 Load user IP address

**Name** *LoadAddress*

**Out** string *myIP* that will contain loaded IP address in case of successful load

**Description** This rule will make an attempt to load IP address of currently running machine from any interface and address space. Addresses from public space have higher priority.

### ■ E.2.6 Load event contents

**Name** *LoadEvent*

**In** string *prefix* placed in front of defined variables

**In** mandatory object *source* that defines access to loaded event. If it is on *const*, trigger event is loaded

## E LOGIC

- Out** integer *id* of general event
- Out** long *timestamp* of general event
- Out** string *class* containing class name of the event
- Out** integer *clientId* that was attached to event as source
- Out** boolean *isNetworkEvent* value on true when event has network flag on
- Out** object *sourceEvent* of interaction event
- Out** object *sourceNode* of interaction event
- Out** integer *keyType* type of keyboard event
- Out** integer *keyCode* of key pressed of keyboard event
- Out** string *keyChar* of key typed of keyboard event
- Out** integer *keyLocation* of where was the key interacted with (numpad, main, ...)
- Out** integer *mouseType* of mouse event
- Out** point *mousePosition* where mouse event happened
- Out** integer *mouseScroll* amount
- Out** integer *mouseButton* code
- Out** integer *windowType* type of window event
- Out** boolean *windowShown* true if is window shown
- Out** boolean *windowFocused* true if is window focused
- Out** dimension *windowDimensions* that is set now
- Out** point *windowPosition* where window resides on screen
- Out** string *messageName* of message event
- Out** object *messageContent* content map of message event

**Description** This rule will load contents of any event into variable space with possible prefix to avoid collisions with other variables. Keep in mind that this rule does not care about the event type, so you have to check it yourself by looking at *class* before accessing any of the outputs. Try to restrain from using this rule as much as possible, because in its generality, it is very slow.

### ■ E.2.7 Load string length

**Name** *LoadStringLength*

**In** mandatory string *source* path to the source being measured

**Out** integer *length* measured value

**Description** This rule will try to load string from given *source* and then measure its length returning it in output *length*.

### ■ E.2.8 Mouse button released over component

**Name** *MouseReleased*

**In** string *class* that will be looked for in styleable components

**In** string *id* that will be looked for in styleable components

**In** integer *buttonCode* checked

**Out** object *nodeOut* that reported interaction event

**Out** string *classOut* collection of style classes of component reporting mouse click separated by space

**Out** integer *idOut* actual style id of component reporting mouse click

**Description** This rule evaluates interaction events over components and checks given information in them.

### ■ E.2.9 String equals

**Name** *StringEquals*

**In** mandatory string *str1* first compared

**In** mandatory string *str2* second compared

**Description** This rule will compare input strings and pass when they are equal. This rule can be replaced by *StringRegexMatch* rule, but it is left here for readability.

### ■ E.2.10 String regex match

**Name** *StringMatchRegex*

**In** mandatory string *target* string to be checked

**In** mandatory string *regex* to be used for match

**Description** This rule is for checking *target* string by defined *regex*. In case of regex match pass will pass this rule as well.

## ■ E.3 Available actions

In this section are described currently implemented actions. Action class does not differ from Rule class all that much, but they are separated for readability.

Outputs of actions do not have to be specified in XML, if they are not, their value will appear on their name in variable space. If inputs are not marked as mandatory, they do not have to be specified. All classes representing actions must have *Action* as a suffix in name. This suffix is implicit and not used in this section.

### ■ E.3.1 Apply stylesheet

**Name** *ApplyStylesheet*

**In** string *path* to loaded and applied stylesheet

**Description** This action serves for dynamic loading of stylesheets into current scene. There is no selector for applying of styles since they are always applied to all elements in the scene starting from root.

### ■ E.3.2 Change state in state space

**Name** *ChangeLogicState*

**In** string *name* specifying state being operated

**In** mandatory string *operation* with keyword operation name from (*enter|exit|add*)

**In** string *path* to state being possibly loaded



**Description** This action will operate with state in state space according to specified operation. For *enter* will be state of given name entered, which means its setup rulesets will be called and it will start receiving events. In case of *exit* will be state exited, teardown rulesets are called and

### ■ E.3.3 Change state of node

**Name** *ChangeNodeState*

**In** string *selector* used for node lookup

**In** object *node* for direct node modification

**In** mandatory string *name* of the state being operated

**In** mandatory string *operation* from keywords (*set|unset|register*)

**Description** This action serves for managing of generally states of nodes in the scene. Set or unset state operation do not distinguish between logical and interaction states and will operate in both spaces at once. This means that if names in logic and interaction spaces overlap, their modification will be simultaneous. From input parameters is mandatory to have filled either *node* or *selector*.

### ■ E.3.4 Change text of component

**Name** *ChangeTextComponent*

**In** mandatory string *selector* leading to modified components

**In** string *operation* from keywords (*replace|append|backspace*) (default *replace*)

**In** string *text* being operated (default empty string)

**In** boolean *report* true if message vent about this change should be generated (default false)

**Description** This action is for modification of text components in scene. Operation *append* and *backspace* are simply for quick modifications and for reducing of amount of performed operations in logic. Correct way to perform those would be to designate separate action operating strings in variable space or registry.

### ■ E.3.5 Clear node

**Name** *ClearNode*

**In** mandatory string *target* selector of node to be cleared

**Description** This action will clear all children from target node(s) and in case of target being *RootNode*, its derived styles will be removed as well.

### ■ E.3.6 Dictionary translation

**Name** *DictionaryTranslation*

**In** mandatory string *input* being translated

**In** mandatory string *patterns* being recognized separated by space

**In** mandatory string *translations* input is translated into separated by space

**Out** string *output* after translation

**Description** This action serves for translating between two sets of strings. These sets have to have equal length. Purpose of this action is to translate node style ids or keywords from user input into array indexes or map keys.

### ■ E.3.7 Dump registry

**Name** *DumpRegistry*

**Description** This action is purely for debugging purposes. It will print out currently used registry records in log message stream.

### ■ E.3.8 Dump states

**Name** *DumpStates*

**Description** This action is purely for debugging purposes. It will print currently active states in state logic of rule handler processing this action.

### ■ E.3.9 Insert scene

**Name** *InsertScene*

**In** mandatory string *path* to loaded scene file

**In** mandatory string *selector* to node that should have contents of scene file inserted

**Description** This action will load scene from given file and store it inside of node defined by selector. This scene is loaded as one instance so it can be inserted only into one node. Keep that in mind while constructing selector, as it must match only for one node! When more nodes are matched, first one is used and warning is reported.

### ■ E.3.10 Load registry

**Name** *LoadRegistry*

**In** mandatory string *path* to file containing loaded registry

**Description** This action serves for loading of registry hierarchy definition and merging it into currently used registry. Be aware that all existing records in current registry are overwritten by records in loaded in case of overlaps. These overlaps are checked in depth, so only leave items and values are changed.

### ■ E.3.11 Log message

**Name** *Logger*

**In** mandatory string *message* to be logged

**Description** This action is purely for debugging purposes. It will log given message in currently used logger as message record.

### ■ E.3.12 Modify counter

**Name** *ModifyCounter*

**In** string *operation* to be performed as keyword form (*inc|dec|set|mod*)

**In** mandatory param *target* containing the counter

**In** double *value* to be used by operation (default *1*)

**Description** This action is for performing simplest operations over numbers. Operation *inc* will do increment of current counter by value, *dec* is analogical to *inc* just for decrement. Operation *set* will replace counter value by set one and lastly *mod* computes modulo by *value*.

### ■ E.3.13 Network controller command

**Name** *Network*

**In** mandatory string *operation* to be performed as a keyword from (*startClient|startHost|stopAccepting*)

**In** string *ports* containing ports to be used by network controller as preferred separated by one space

**In** string *hostname* to be used for client connection

**Description** This action serves for initiating of simple network connection. Ports do not need to be specified, network controller already has a set of preferred ports. When connection is established, initiator of connection receives internal message saying *CONNECTION\_SUCCESS*. *CONNECTION\_FAILURE* is message received after failed connection attempt for any reason. Initiator of connection will receive additional message *CLIENT\_ID* with platform id this client has. All clients receive message *NEW\_CLIENT* to make them aware of new peer. Stopping server from accepting new connections is done by sending message *STOP\_ACCEPTING* in related event handler. Once connection is established, all message events marked as network are sent to all peers. Warning: TCP protocol will ensure all events eventually arrive to their destination, but their order may be different!

### ■ E.3.14 Send message

**Name** *SendMessage*

**In** boolean *network* true in case the message should be propagated to peers (default false)

**In** mandatory string *name* that will identify message

**In** object *content* being sent with message

**Description** This action serves for sending messages inside of the application and to peers using *network* setting. When attached content is being sent over network, it has to be serializable object, or otherwise the message will not acquire network flag. Content sent is map of elements which is constructed by using *content* as a prefix and any additional string is suffix to locate specific content in the message.

### ■ E.3.15 Shutdown application

**Name** *Shutdown*

**Description** This action will terminate whole application. Make sure you have finished all the work before calling it.

### ■ E.3.16 Store data

**Name** *StoreData*

**In** mandatory param *location* to have the data stored

**In** mandatory object *data* to be stored

**Description** This action will store given data on given location.

## ■ E.4 Grouping

Elements in game logic inside of their ruleset may be grouped by using pair tag *group*. This group will behave as if all contained actions were one element along with reporting success. Groups have one important property, and that is grouping operation, which allows you to set group in different mode of processing.

**and** Under *and* operation will group attempt to process all contained elements and will terminate on first failure reporting failure. Success is reported only if all elements were processed successfully.

**or** With *or* operation are contained elements processed until one is successful and that result returns success. If none were successful, failure is returned. This operation has readability shortcut *ifthenelse*, which isn't much of a shortcut, but it is more transparent what is related group supposed to do.

**not** Operation *not* is for negating result of first contained element. All remaining elements are dropped as not expected.

**any** This operation, *any* serves for processing where we do not care about the result all that much. All logs stay in place, but all elements will get attempt to get processed and event if they fail, group will report success.

## E LOGIC

Along with grouping operations, group gains one more important feature, which is for loop. This loop is specified using fixed parameters:

**In** real number *loopFrom* what value will be loop iterating (default 0)

**In** mandatory real number *loopTo* what value will be loop iterating (inclusive!)

**In** real number *loopStep* that is added at the end of each iteration (default 1)

**Out** real number *loopIndex* containing current value

Loop index can be of course, as all output parameters, redirected to different location so nested loops are an option. Do not forget that complicated operations should be implemented as custom actions or rules and these loops should be used only for simple element iterating.

### ■ E.5 Adding new elements

Process of adding new rules and actions is much simpler compared to addition of new components or properties simply because it is intended to be common way to solve problems during game creation. Whenever you feel like some process you are doing frequently or process that can be generalized and used often at different places, do consider creating element in java for it.

**Creating class** In appropriate package (*Core.Evaluation.(Action|Rule).Custom*) create new class of name of your choosing, but it must have suffix (*Action|Rule*) depending on package. This class must extend class *EvaluationElement* and implement interface, again, depending on package (*Action|Rule*).

**Overriding construct** Now you should decide what input and output parameters will be used, which ones will be mandatory and how they should be interpreted. All that is decided in *construct* method that is called when element is instantiated passing parameters from XML. Here before calling super set all output parameters so they can be automatically filled with results without user having to specify them, and after super defined mandatory parameters along with interpretation of constants.

**Defining event subscriptions** In implementation of *getRequiredEventSubscriptions* should be definition of what event types interest you in body method of the element. Keep in mind that the more events you will subscribe to, the more demanding will evaluation be.

**Implementing core method** Last you should implement element core methods *evaluate* or *execute* depending on the interface.

Be aware that this method should have all exceptions caught and logged, for if it is not, the exception is stopped way too late, which can mess up a lot of things and will be difficult to resolve.

## ■ F Optimization

In this section are described ways to improve performance of the engine by simply playing along with it while creating the game.

### ■ F.1 Scene

Although graphics of this engine is in current state strictly 2D, it still can be demanding for the computer, since it is not done on graphics card but on CPU with use of java graphics acceleration.

**Amount of graphics calls** There is limited amount of separate calls for every machine to handle, no matter how difficult these are and wasting system resources on calls that can be easily aggregated is simply not a good idea. That pays especially for background patterns, where the smaller background image is and the bigger component surface is, the longer recalculation of that surface takes. just making the pattern file larger (disk space and memory are not as big of a deal these days), will reduce calculation time significantly.

**Precomputing** If you have imagery with certain effect that does not change during runtime or with resolution of game window, precalculate it. Of course for that is during development of the game, when a lot of changes is made, simpler to define all the effects in styles, but when going through final optimizations, card with fixed dimensions and border with rounded edges will be better stored that way and loaded as one image.

**Load time calculations** Adding of children in scene nodes at runtime is generally slow, especially after styles have been already applied, since all previously applied styles have to be checked on match on every added child. That can be prevented by having whole scene loaded in one piece, all styles applied afterwards and use visibility control to change what is visible on the screen. Change of visibility does not invalidate cache states unlike all other state changes so it is not recalculated on appearing, and thus it is almost instantaneous.

**Amount of scene events** Events propagating through scene are clogging the event handler processing unit, and that counts especially for mouse move events. There are hundreds of those generated every second and are put to evaluation even though they are sure to be discarded everywhere. To avoid that, make sure you put all components in the scene that do not serve any active purpose (buttons, cards, ...) in disabled state with property *solid* on *true*. This will ensure that hover over component is not only

blocked before reaching underlying components, but component being interacted with will not take any response report actions.

**Style selector complexity** The longer and more complex selectors are, the more expensive it is to do selections with them. Because of that is used the fact that all style IDs should be unique across the scene and all styled nodes are keyed under their style IDs. So when selector containing style ID arrives, it is immediately shifted to its position and to related node. That being set, you should make shorter selectors starting with style IDs.

### ■ F.2 Logic

Logic of games made in this engine does not necessarily need to be real time, just because they are out of principle turn based, but they should still be responsive and excessive amounts of logic rules will bring the responsiveness down.

**Amount of rulesets** When you are creating game logic, you will probably tend to separate things that are not directly related out of tidiness. That is all nice during development, but once optimizing, what can be merged into one ruleset should be merged. When same check is done with every event that appears on logic handler multiple times as it is used for similar actions, it is slowing everything down in general. That is because entry checks need to be called every time, even if they obviously cannot pass, and this multiplied with amount of active rulesets.

**Amount of active states** Every state contains some amount of rulesets and those may be active in different instances of the state evaluation. This is unnecessary work, since all rulesets that cannot be obviously passed anymore should be left behind in state dedicated just to them. It is much better to split some states and make more transitions to keep just a small amount of rulesets active. This not only reduces load on logic evaluation, but it also makes debugging easier.

**Subscribed events** Every rule and action have some event subscription to events they care about. If you subscribe your custom rules and actions to all events, they will be evaluated A LOT, and that may cause general slowdown of the system. Not only make subscriptions to events you really need, but as well avoid using elements that already do subscribe to everything, for example *LoadEventRule*.

### ■ F.3 Network

Even though this engine does not work with real time interaction between players, it is still good idea to save as much as possible on network transfers, so the game can run fine even on mobile connection.



**Server data on server** Even though you might be tempted to do it, just don't. Do not send data that are not exactly necessary for client to have, and especially not in bulks. For example when you create a deck of cards available to every player to draw from on server, do not distribute it, and just let players ask for single card, what more, just for ID of a card that is out of the box on everyones computer already.

**Small packets** Do not send excessive amounts of data over to clients, even though it may be comfortable, since a lot of objects in the engine are already externalizable. If the data can be part of the game or recreated by client, let the client do so. For instance mentioned deck of cards is created with specific random seed. If server just generates seeds for players and distributes those, amount of data necessary for building those decks is reduced significantly.

## ■ G Contents of DVD

In this section are listed contents of attached DVD with brief description.

### **Example games/Cards Against Humanity**

This folder contains executable for example game *Cards Against Humanity* and all necessary files for game to be runnable. Note that this game needs *JRE 8* or higher to be able to run and that firewall exception may have to be added before running it.

### **Example games/Tic Tac Toe**

This folder contains executable for example game *Tic Tac Toe* and all necessary files for game to be runnable. Note that this game needs *JRE 8* or higher to be able to run.

### **Installation/jdk-8u45-windows-x64.exe**

Is installation file for *Java Development Kit* engine was tested on.

### **Installation/jre-8u45-windows-x64.exe**

Is installation file for *Java Runntime Environment* all games created in engine are verified on.

### **Installation/netbeans-8.0.2-windows.exe**

Is installation file for *NetBeans IDE* in which was created project for the game engine. Use this *IDE* if you do not want to deal with *JAXB* definitions setup.

### **Project**

This folder contains complete project for *NetBeans IDE* [?] along with all source files for the engine. Project also includes basic template application.

### **Thesis.pdf**

This file contains thesis you are reading right now in *PDF* format.

## References

- [1] Bartle, Richard A., *Designing Virtual Worlds*, Berkely: New Riders, 2004. p.108. Print.
- [2] Eberly, David H., *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*, 2nd ed., San Francisco: Morgan Kaufmann Publishers, 2007. pp.785-787. Print.
- [3] Eberly, David H., *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*, 2nd ed., San Francisco: Morgan Kaufmann Publishers, 2007. p.183. Print.
- [4] Eberly, David H., *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*, 2nd ed., San Francisco: Morgan Kaufmann Publishers, 2007. pp.223-228. Print.
- [5] Frapolli, Fulvio, *FlexibleRules: A Player Oriented Board Game Development Framework*, PhD. thesis. University of Fribourg, 2010. Print.
- [6] Ward, Jeff, *What is a Game Engine?*, 29.4.2008. Last accessed May 2015. [www.gamecareerguide.com/features/529/?page=2](http://www.gamecareerguide.com/features/529/?page=2)